Non-Speculative Reordering of Memory Operations with Strong Consistency

Alberto Ros

Universidad de Murcia

November 29th, 2017

OUTLINE

1. MEMORY CONSISTENCY AND PROGRAM ORDER
2. RELAXING PROGRAM ORDER WITH A STORE BUFFER
3. KEEPING PROGRAM ORDER VIA SPECULATION
4. A NON-SPECULATIVE SOLUTION: WRITERSBLOCK
5. EVALUATION RESULTS
6. CONCLUSIONS

Outline

1. Memory consistency and program order
2. Relaxing program order with a store buffer
3. Keeping program order via speculation
4. A non-speculative solution: WritersBlock
5. Evaluation results
6. Conclusions

Program order (P.O.)

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

Thread 1

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

Program order (P.O.)

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

Thread 1

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

- What happens if the core/memory changes this order?

Thread 1

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

PROGRAM ORDER (P.O.)

- 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 core/memory changes this order?

THREAD 1

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

THREAD 2

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

Possible results assuming program order

Initially X=0, Y=0

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

Possible results assuming program order

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 operations execute in program order

Relaxing program order (loads)

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 the order in which loads execute

Relaxing program order (loads)

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)

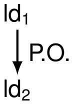
- (1,0) is possible by relaxing the order in which loads execute
    - The same result can be achieved by relaxing the stores

The memory consistency model

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

The memory consistency model

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



ld$_1$

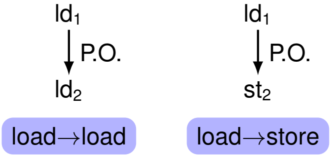
P.O.

ld$_2$

load→load

THE MEMORY CONSISTENCY MODEL

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



$ld_1 \xrightarrow{P.O.} ld_2$

load→load

$ld_1 \xrightarrow{P.O.} st_2$

load→store

THE MEMORY CONSISTENCY MODEL

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

$$\begin{tikzcd} \mathrm{ld}_1 \arrow[d, "\mathrm{P.O.}"] \\ \mathrm{ld}_2 \end{tikzcd}$$

load→load

$$\begin{tikzcd} \mathrm{ld}_1 \arrow[d, "\mathrm{P.O.}"] \\ \mathrm{st}_2 \end{tikzcd}$$

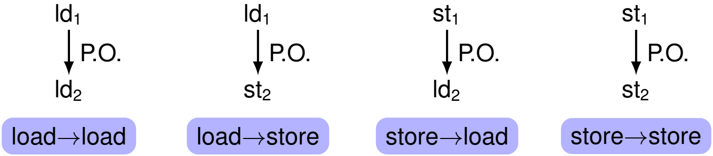
load→store

$$\begin{tikzcd} \mathrm{st}_1 \arrow[d, "\mathrm{P.O.}"] \\ \mathrm{ld}_2 \end{tikzcd}$$

store→load

The memory consistency model

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



$ld_1 \xrightarrow{P.O.} ld_2$

load→load

$ld_1 \xrightarrow{P.O.} st_2$

load→store

$st_1 \xrightarrow{P.O.} ld_2$

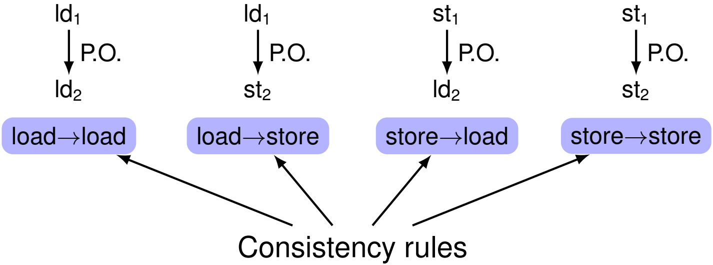
store→load

$st_1 \xrightarrow{P.O.} st_2$

store→store

The memory consistency model

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



$ld_1 \xrightarrow{\text{P.O.}} ld_2$: load→load

$ld_1 \xrightarrow{\text{P.O.}} st_2$: load→store

$st_1 \xrightarrow{\text{P.O.}} ld_2$: store→load

$st_1 \xrightarrow{\text{P.O.}} st_2$: store→store

Consistency rules

CORRECTNESS/PERFORMANCE ISSUE

- Correctness
  - The programmer intuition is program order

CORRECTNESS/PERFORMANCE ISSUE

- Correctness
  - The programmer intuition is program order
- Performance
  - Waiting for a memory operation to finish in order to start the execution of the next operation is very inefficient
  - Processors execute multiple memory operations simultaneously
    - Memory level parallelism

CORRECTNESS/PERFORMANCE ISSUE

- Correctness
  - The programmer intuition is program order
- Performance
  - Waiting for a memory operation to finish in order to start the execution of the next operation is very inefficient
  - Processors execute multiple memory operations simultaneously
    - Memory level parallelism
  - Operations can be reordered by the memory hierarchy, or even be issued out-of-order

CORRECTNESS/PERFORMANCE ISSUE

- Correctness
  - The programmer intuition is program order
- Performance
  - Waiting for a memory operation to finish in order to start the execution of the next operation is very inefficient
  - Processors execute multiple memory operations simultaneously
    - Memory level parallelism
  - Operations can be reordered by the memory hierarchy, or even be issued out-of-order
  - This is correct for single-core processors, but not in multicores

CORRECTNESS/PERFORMANCE ISSUE

- Correctness
  - The programmer intuition is program order
- Performance
  - Waiting for a memory operation to finish in order to start the execution of the next operation is very inefficient
  - Processors execute multiple memory operations simultaneously
    - Memory level parallelism
  - Operations can be reordered by the memory hierarchy, or even be issued out-of-order
  - This is correct for single-core processors, but not in multicores
- Solution: Store Buffer and Speculation

Outline

1. Memory consistency and program order
2. Relaxing program order with a store buffer
3. Keeping program order via speculation
4. A non-speculative solution: WritersBlock
5. Evaluation results
6. Conclusions

The Store Buffer

- A store operation requires write permission to perform
- Write permission request
  - Cache coherence protocol
  - Unique copy: may require invalidating other copies
  - A long-latency operation

The Store Buffer

- A store operation requires write permission to perform
- Write permission request
  - Cache coherence protocol
  - Unique copy: may require invalidating other copies
  - A long-latency operation
- Solution implemented in x86 processors (Intel, AMD)

  ⇒ The store buffer

The store buffer breaks store→load

- How the store buffer (SB) works?

THE STORE BUFFER BREAKS STORE→LOAD

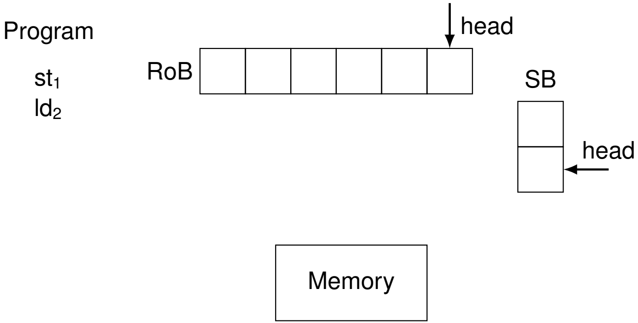
- How the store buffer (SB) works?

Program

st_1
ld_2

THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



Program

st_1
ld_2

RoB

head

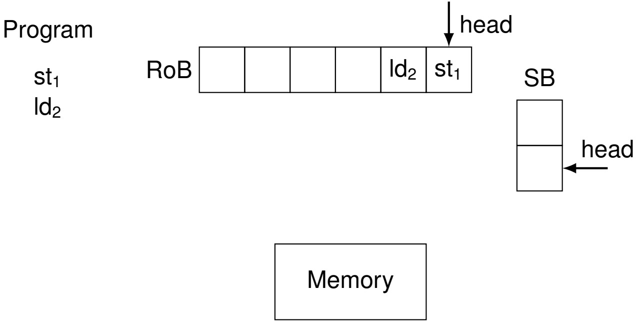
SB

head

Memory

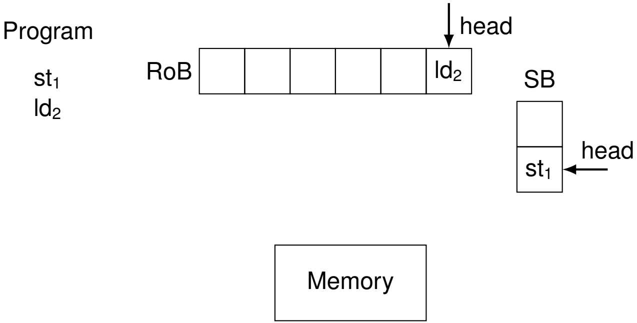
THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



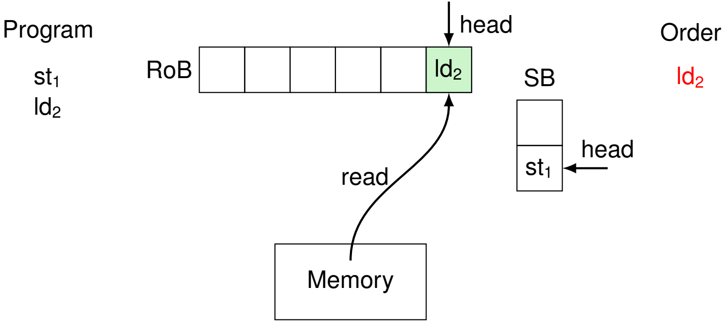
THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



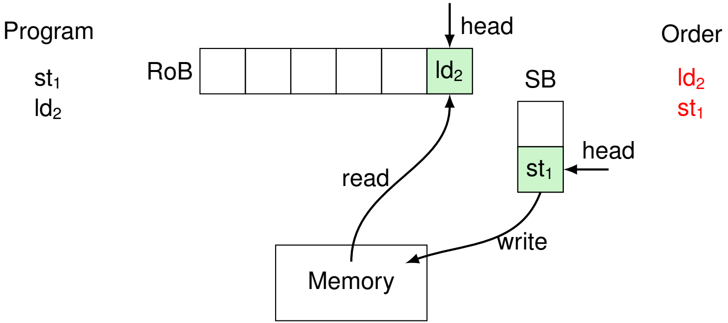
THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



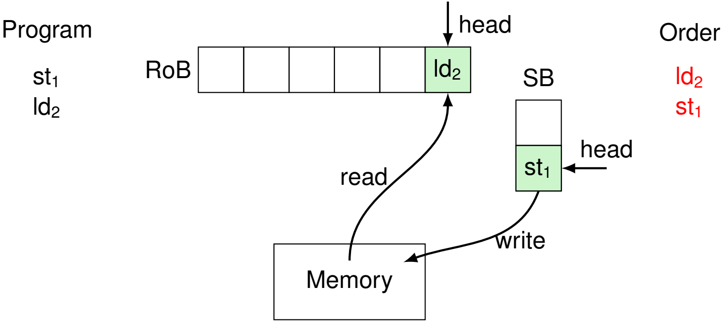
THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



THE STORE BUFFER BREAKS STORE→LOAD

- How the store buffer (SB) works?



- The store buffer breaks the store→load rule

Total Store Order (TSO)

- x86 processors (Intel, AMD) provide a Total Store Order (TSO) memory consistency model

Total Store Order (TSO)

- x86 processors (Intel, AMD) provide a Total Store Order (TSO) memory consistency model

TSO rules

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

Total Store Order (TSO)

- x86 processors (Intel, AMD) provide a Total Store Order (TSO) memory consistency model

TSO rules

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

- TSO does not enforce store→load
- Performance over programmer intuition

THE STORE BUFFER: CONSEQUENCES

- store→load

  ⇒ Relaxed

The Store Buffer: Consequences

- store→load

  ⇒ Relaxed

- load→store

  ⇒ No need to execute stores before the loads since stores are out of the critical path

The Store Buffer: Consequences

- `store→load`
  - ⇒ Relaxed
- `load→store`
  - ⇒ No need to execute stores before the loads since stores are out of the critical path
- `store→store`[1]
  - ⇒ Less critical than without a store buffer, unless the store buffer fills

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

The Store Buffer: Consequences

- store→load
  - ⇒ Relaxed
- load→store
  - ⇒ No need to execute stores before the loads since stores are out of the critical path
- store→store[1]
  - ⇒ Less critical than without a store buffer, unless the store buffer fills
- load→load
  - ⇒ It is now the bottleneck

[1] A. Ros and S. Kaxiras, “Racer: TSO Consistency via Race Detection”. MICRO, 2016.

OUTLINE

1 MEMORY CONSISTENCY AND PROGRAM ORDER

2 RELAXING PROGRAM ORDER WITH A STORE BUFFER

3 KEEPING PROGRAM ORDER VIA SPECULATION

4 A NON-SPECULATIVE SOLUTION: WRITERSBLOCK

5 EVALUATION RESULTS

6 CONCLUSIONS

LOAD→LOAD REORDERING

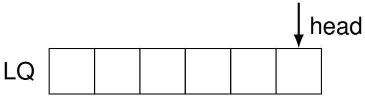
- Executing load operations out of order can break the load→load order

LOAD→LOAD REORDERING

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

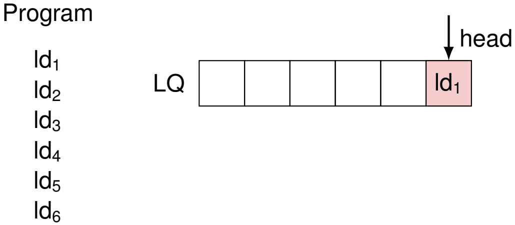
Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6



LOAD→LOAD REORDERING

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



Program

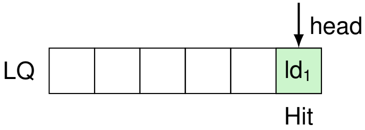
ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

LOAD→LOAD REORDERING

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

Program

ld$_1$
ld$_2$
ld$_3$
ld$_4$
ld$_5$
ld$_6$

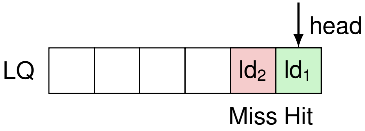


LOAD→LOAD REORDERING

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

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

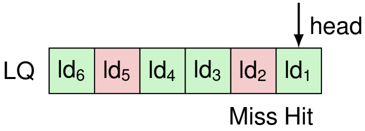


Load→load reordering

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

Program

ld$_1$
ld$_2$
ld$_3$
ld$_4$
ld$_5$
ld$_6$



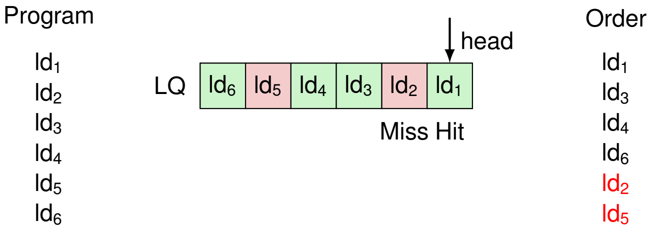
head

LQ	ld$_6$	ld$_5$	ld$_4$	ld$_3$	ld$_2$	ld$_1$
	🟩	🟥	🟩	🟩	🟥	🟩

Miss Hit

Load→load reordering

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



Program

ld$_1$
ld$_2$
ld$_3$
ld$_4$
ld$_5$
ld$_6$

head

LQ ld$_6$ ld$_5$ ld$_4$ ld$_3$ ld$_2$ ld$_1$

Miss Hit

Order

ld$_1$
ld$_3$
ld$_4$
ld$_6$
ld$_2$
ld$_5$

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
  - But always?

LOAD→LOAD REORDERING

- In multicore processors reordering loads can affect the expected result
    - But always?

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
  - But always?

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
  - But always?

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
  - But always?

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

LOAD→LOAD SPECULATION

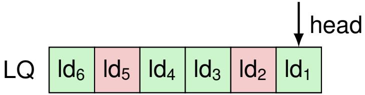
- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)

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

Load→Load speculation

- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)

Program	Order
ld_1	ld_1
ld_2	ld_3
ld_3	ld_4
ld_4	ld_6
ld_5	ld_2
ld_6	ld_5



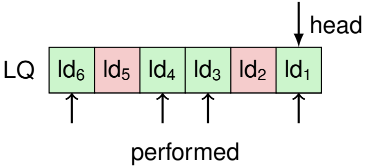
[2] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, “Non-Speculative Load-Load Reordering in TSO”. ISCA, 2017.

Load→load speculation

- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)

Program

ld$_1$
ld$_2$
ld$_3$
ld$_4$
ld$_5$
ld$_6$



head

LQ

ld$_6$	ld$_5$	ld$_4$	ld$_3$	ld$_2$	ld$_1$

performed

Order

ld$_1$
ld$_3$
ld$_4$
ld$_6$
ld$_2$
ld$_5$

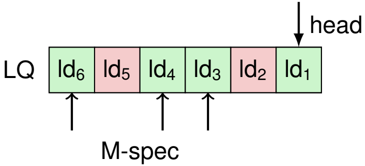
[2] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, “Non-Speculative Load-Load Reordering in TSO”. ISCA, 2017.

Load→load speculation

- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)

Program

ld$_1$
ld$_2$
ld$_3$
ld$_4$
ld$_5$
ld$_6$



Order

ld$_1$
ld$_3$
ld$_4$
ld$_6$
ld$_2$
ld$_5$

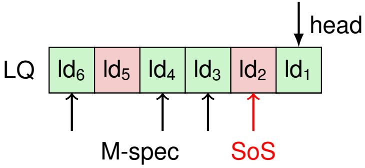
[2] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, “Non-Speculative Load-Load Reordering in TSO”. ISCA, 2017.

LOAD→LOAD SPECULATION

- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)

Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6



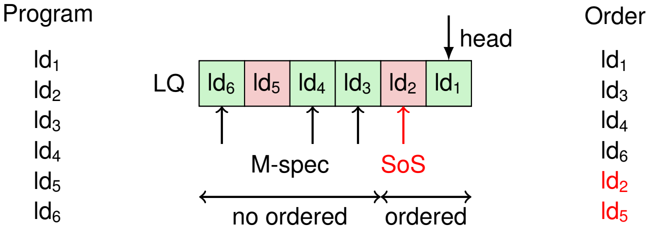
Order

ld_1
ld_3
ld_4
ld_6
ld_2
ld_5

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

LOAD→LOAD SPECULATION

- Solution: To allow speculative load→load reordering
- Some definitions[2]: performed, ordered, source of speculation (SoS)



Program

ld_1
ld_2
ld_3
ld_4
ld_5
ld_6

head

LQ

ld_6	ld_5	ld_4	ld_3	ld_2	ld_1

M-spec

SoS

no ordered

ordered

Order

ld_1
ld_3
ld_4
ld_6
ld_2
ld_5

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

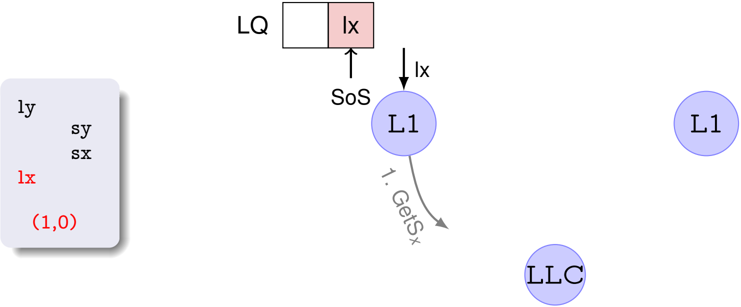
Squash and re-execute upon invalidation

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

```
ly
        sy
        sx
lx

  (1,0)
```

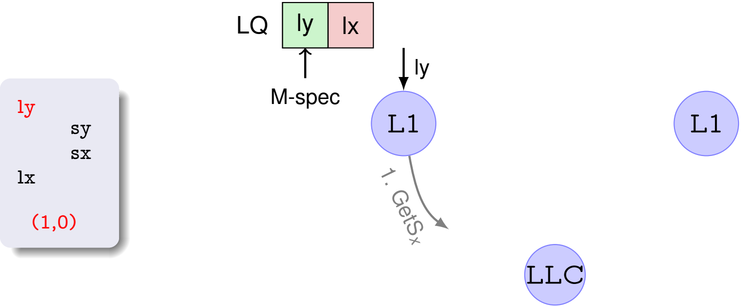
Squash and re-execute upon invalidation

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



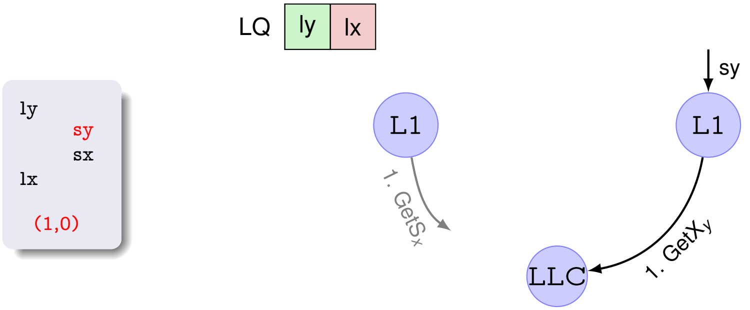
Squash and re-execute upon invalidation

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



Squash and re-execute upon invalidation

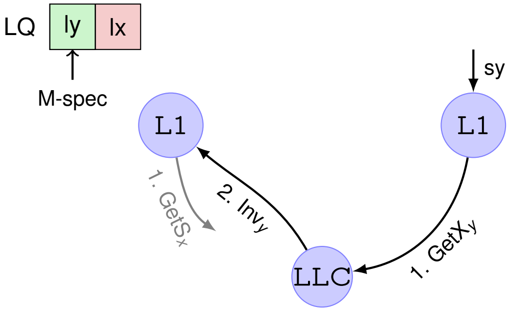
- Current multicore avoid incorrect results
  - With the help of the cache coherence protocol



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)
```

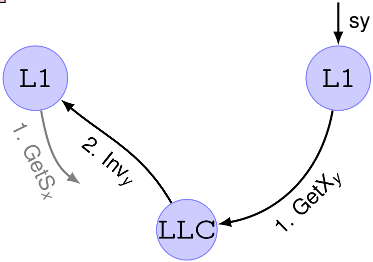


Squash and re-execute upon invalidation

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

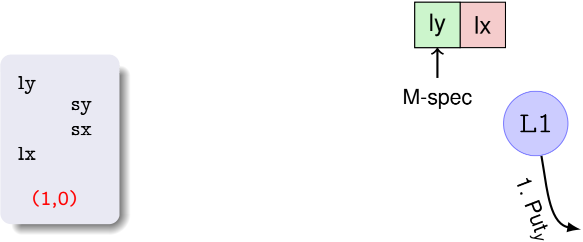
LQ	ly	lx

```
ly
        sy
        sx
lx

  (1,0)
```



Squash and re-execute upon evictions

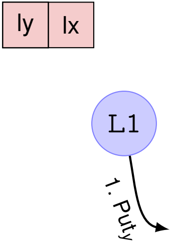
- What happens when a cache block loaded by an M-spec load is evicted?
  - If the directory stops tracking the block, the M-spec load will not receive an invalidation



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

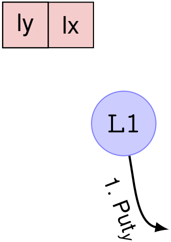
```
ly
        sy
        sx
lx

  (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 have MLP and load→load

PROBLEMS OF SPECULATION

- Memory-related speculation is the current solution to have MLP and load→load
- Why is good?
  - Squashing is not frequent!

PROBLEMS OF SPECULATION

- Memory-related speculation is the current solution to have MLP and load→load
- Why is good?
  - Squashing is not frequent!
- Why is bad?
  - Speculative loads hold critical resources (LQ, RoB)
  - The processor needs to keep continuously the rollback path

PROBLEMS OF SPECULATION

- Memory-related speculation is the current solution to have MLP and load→load
- Why is good?
  - Squashing is not frequent!
- Why is bad?
  - Speculative loads hold critical resources (LQ, RoB)
  - The processor needs to keep continuously the rollback path

QUESTION

Can we execute loads out of order, non-speculatively and guaranteeing load→load?

Outline

1. Memory consistency and program order
2. Relaxing program order with a store buffer
3. Keeping program order via speculation
4. A non-speculative solution: WritersBlock
5. Evaluation results
6. Conclusions

WritersBlock in a nutshell[2]

- What?
  - Multiple loads executing simultaneously
  - Load→load
  - Without memory-related speculation
- How?
  - Blocking write requests
  - With the help of the cache coherence protocol

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

How?

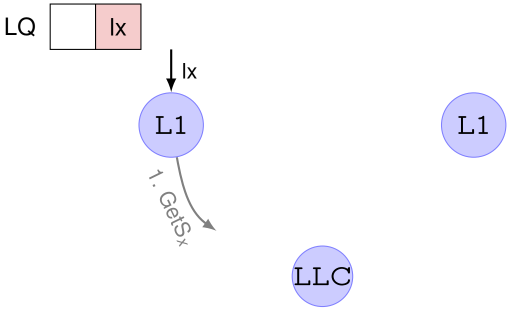
- With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

How?

- With the help of the cache coherence protocol

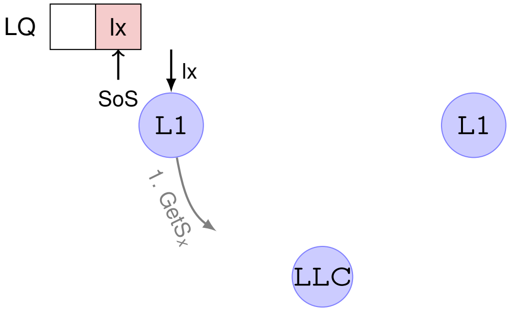
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

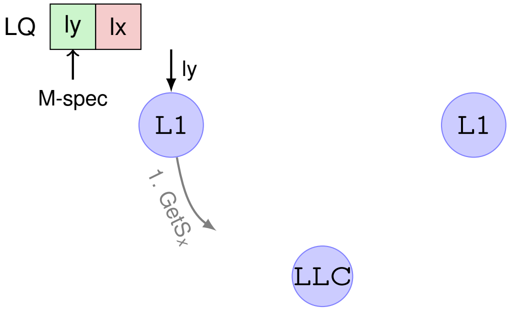
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

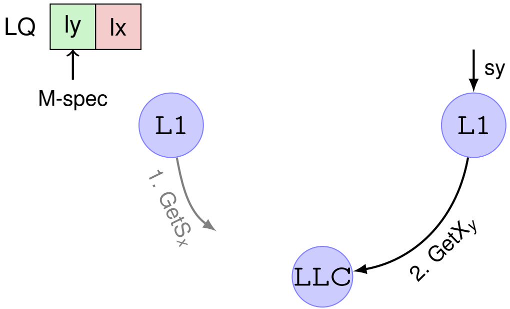
```
ly
        sy
        sx
lx

  (1,0)
```



How?

- With the help of the cache coherence protocol

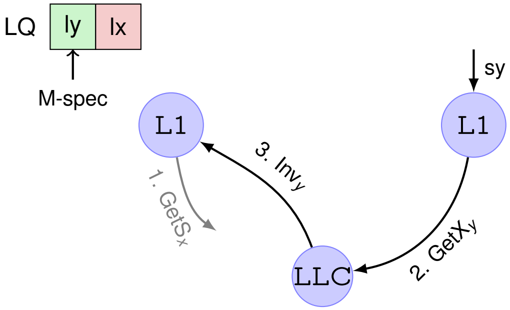
```
ly
        sy
        sx
lx

  (1,0)
```



How?

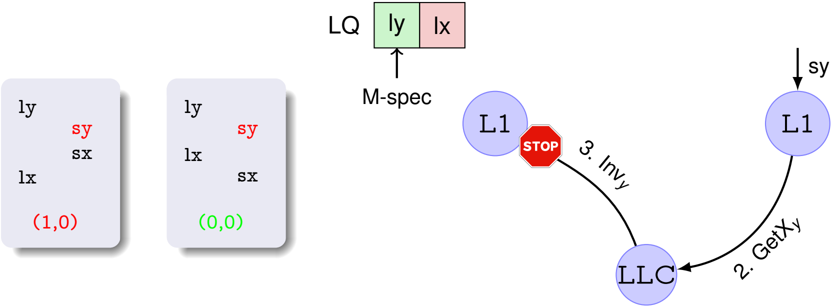
- With the help of the cache coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```



How?

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



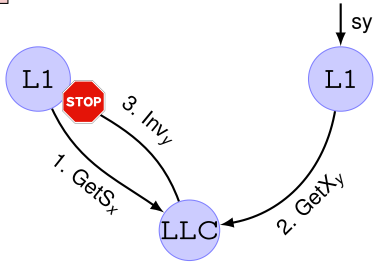
How?

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

LQ	ly	lx

```
ly
        sy
        sx
lx

  (1,0)
```

```
ly
        sy
lx
        sx

  (0,0)
```



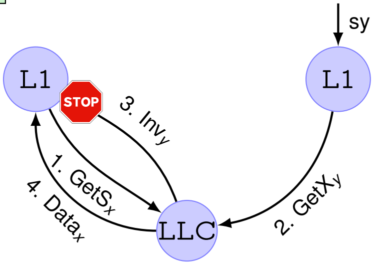
How?

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

LQ	ly	lx

ly
sy
sx
lx
(1,0)

ly
sy
lx
sx
(0,0)



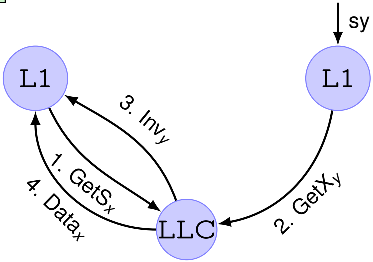
How?

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

LQ	ly	lx

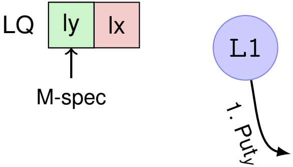
ly	
	sy
	sx
lx	
(1,0)	

ly	
	sy
lx	
	sx
(0,0)	



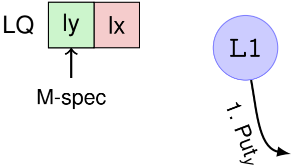
Evictions

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



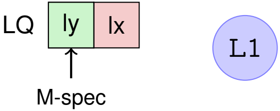
EVICTIONS

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



Evictions

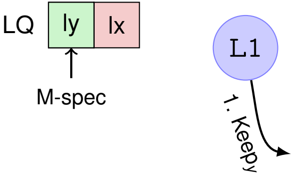
- What happens upon an eviction? Do we squash loads?
  - No, just need to guarantee that the invalidation will arrive upon a remote write
- Solution:
  - Clean blocks implement silent evictions[3]



[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

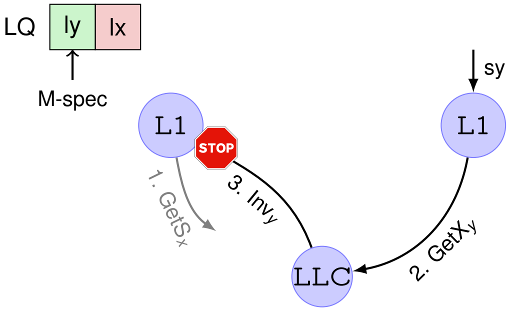


[3] R. Fernandez-Pascual, A. Ros, and M. E. Acacio, "To Be Silent or Not: On the Impact of Evictions of Clean Data in Cache-Coherent Multicores", Journal of Supercomputing, 2017.

Deadlock

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

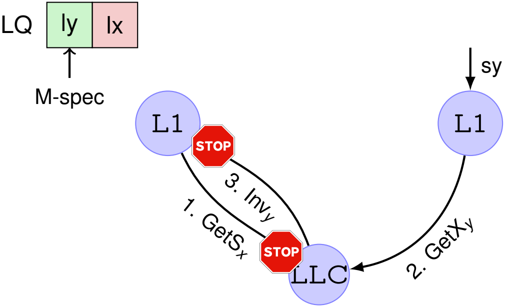
```
ly
        sy
        sx
lx

  (1,0)
```



Deadlock

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

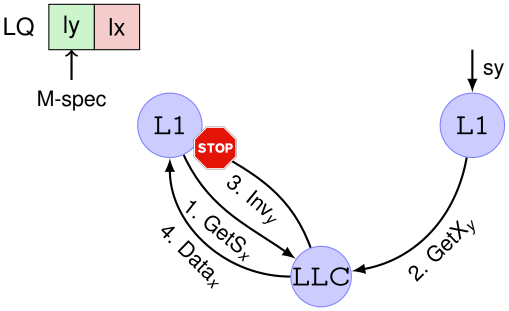
```
ly
        sy
        sx
lx

  (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

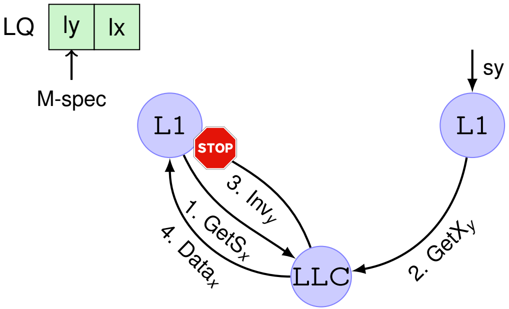
```
ly
        sy
        sx
lx

  (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

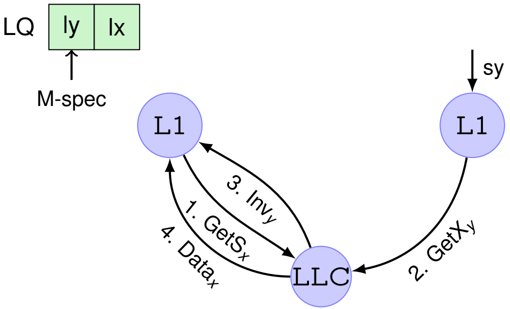
```
ly
        sy
        sx
lx

  (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 has been invalidated will cause a second invalidation
  - Blocked$_i$, Read$_j$, Unblock$_i$, Invalidate$_j$, Blocked$_j$, ...

LIVELOCK

- Resolving reads while blocking writes can cause livelock
  - Resolving a read once the data has been invalidated will cause a second invalidation
  - Blocked$_i$, Read$_j$, Unblock$_i$, Invalidate$_j$, Blocked$_j$, ...
- Solution
  - Reads resolved through WritersBlock are non-cacheable

    ⇒ No invalidations needed
  - 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

Deadlock avoidance

- WritersBlock cause writes to be blocked
  - Until a load stop being M-speculative
- Deadlock-free condition:

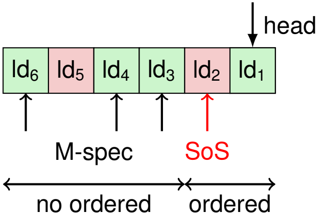
  ⇒ Loads are not stopped by pending write misses

Deadlock avoidance

- WritersBlock cause writes to be blocked
  - Until a load stop being M-speculative
- Deadlock-free condition:
  - ⇒ Loads are not stopped by pending write misses
- Other blocking causes and solutions:
  - MSHR address occupied by write miss
    - ⇒ Duplicate read-write MSHR allocation
  - Full directory/LLC
    - ⇒ Non-cacheable loads
  - Atomic Read-Modify-Write
    - ⇒ Non-speculative (ordered)

Case of use: Out-of-order commit

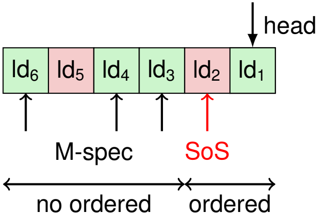
- Out-of-order commit[4] allows the processor to retire instructions from the reorder buffer (RoB) even if they are not at the head
- It cannot retire instructions that can be squashed



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

Case of use: Out-of-order commit

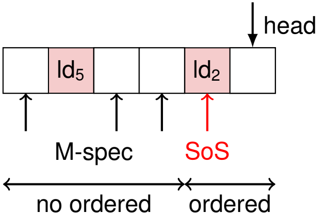
- Out-of-order commit[4] allows the processor to retire instructions from the reorder buffer (RoB) even if they are not at the head
- It cannot retire instructions that can be squashed
- WritersBlock allows the retirement of out-of-order loads
- Better RoB/LQ usage



[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 can be squashed
- 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

1. Memory consistency and program order
2. Relaxing program order with a store buffer
3. Keeping program order via speculation
4. A non-speculative solution: WritersBlock
5. Evaluation results
6. Conclusions

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.

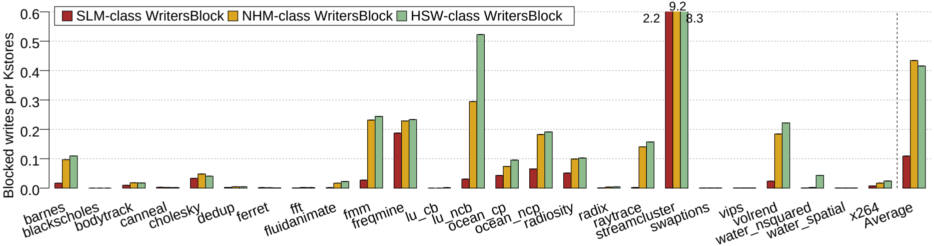
Simulation environment

- 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.

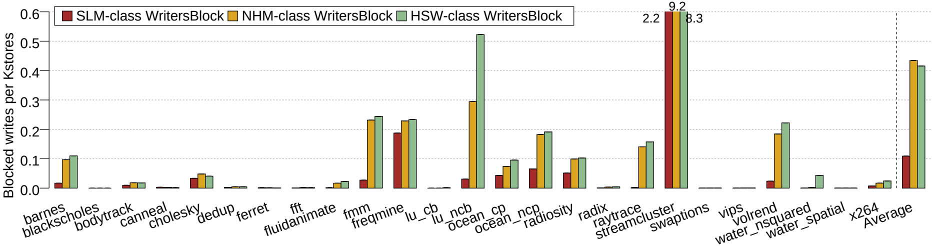
WritersBlock: Blocked writes

- Results for InOrderCommit
- Normalized to Directory



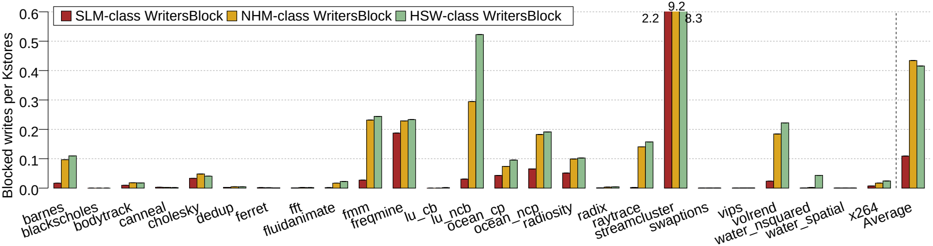
WritersBlock: Blocked writes

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more loads executed out-of-order, and the more blocked writes



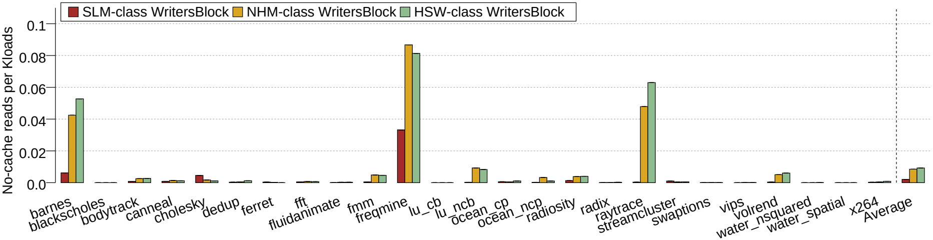
WritersBlock: Blocked writes

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



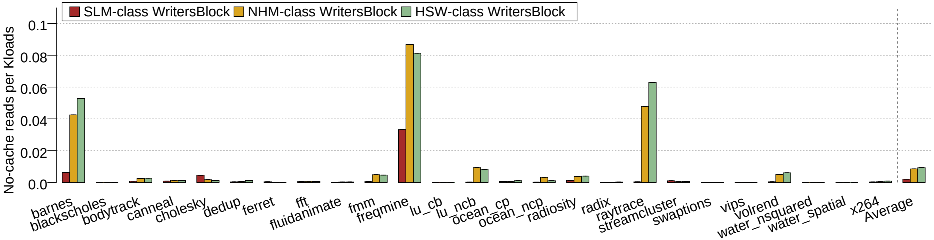
WritersBlock: Non-cacheable data

- Results for InOrderCommit
- Normalized to Directory



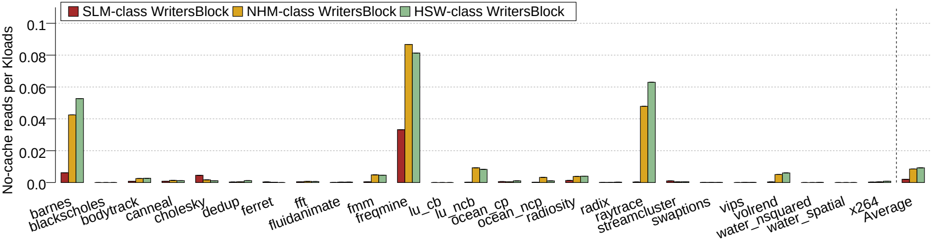
WritersBlock: Non-cacheable data

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



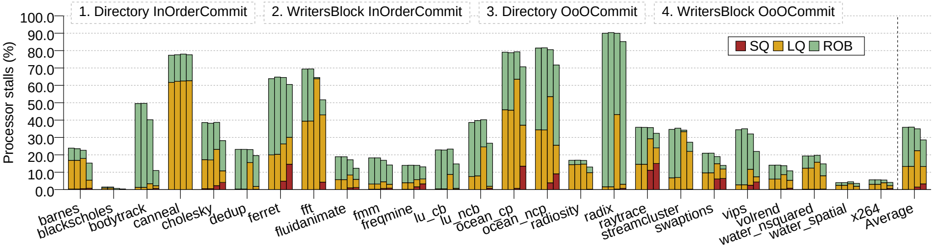
WritersBlock: Non-cacheable data

- Results for InOrderCommit
- Normalized to Directory
- The larger the RoB, the more writes blocked, and the more non-cacheable data
- ≈ 1 non-cacheable data per 100,000 loads, on average



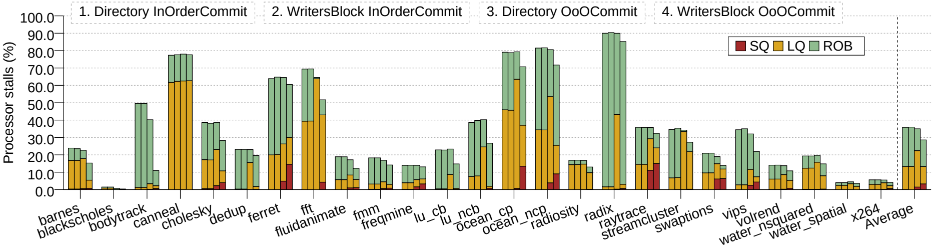
Out-of-order Commit: Processor stalls

- Normalized to Directory + InOrderCommit



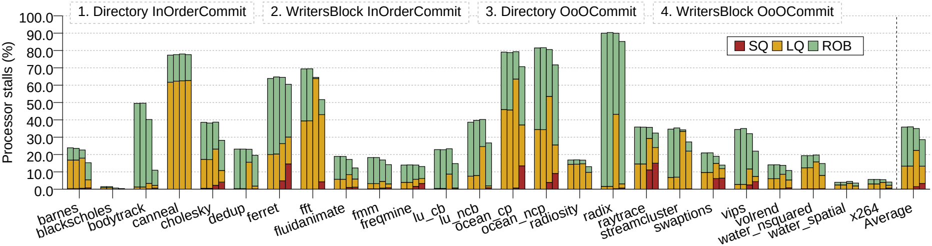
Out-of-order Commit: Processor stalls

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



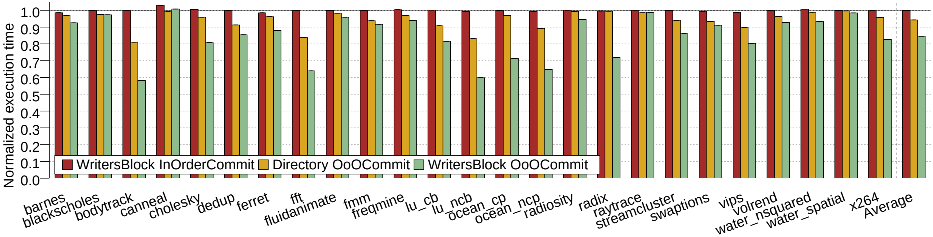
Out-of-order Commit: Processor stalls

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



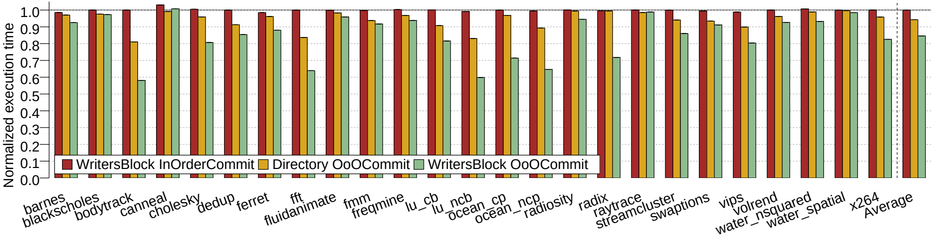
Out-of-order Commit: Execution time

- Normalized to Directory + InOrderCommit



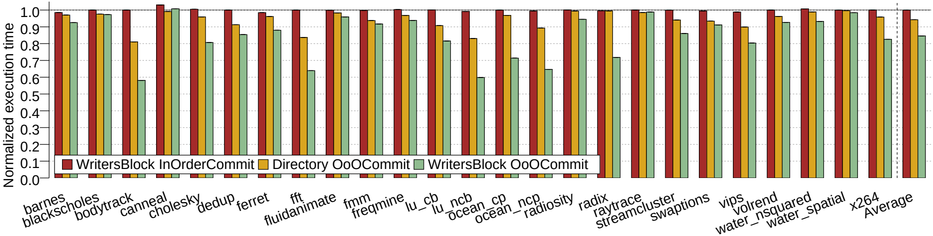
Out-of-order Commit: Execution time

- 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

1. Memory consistency and program order
2. Relaxing program order with a store buffer
3. Keeping program order via speculation
4. A non-speculative solution: WritersBlock
5. Evaluation results
6. Conclusions

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 REORDERING OF MEMORY OPERATIONS WITH STRONG CONSISTENCY

Alberto Ros

Universidad de Murcia

November 29th, 2017