Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations

Eduardo José Gómez-Hernández[1] Juan M. Cebrian[1]
Rubén Titos-Gil[1] Stefanos Kaxiras[2] Alberto Ros[1]

[1] Computer Engineering Department
University of Murcia, Spain

[2] Department of Information Technology
Uppsala University, Sweden

OVERVIEW

Overview

- Programmers have always request the support of read-modify-write atomics of several address

OVERVIEW

- Programmers have always request the support of read-modify-write atomics of several address

- Ideally multi-address atomics should be:
  - fine-grained locking to enable concurrency
  - non-speculative to prevent retries (re-executions/aborts)

OVERVIEW

- Programmers have always request the support of read-modify-write atomics of several address
- Ideally multi-address atomics should be:
  - fine-grained locking to enable concurrency
  - non-speculative to prevent retries (re-executions/aborts)
- Our goal is:
  - achieve both goals: fine-grained locking and non-speculative
  - avoid deadlocks due to limited resources:
    - Rely only on the coherence protocol and a predetermined locking order
  - Outperform software locks (3.4×) and Intel transactional memory (2.7×)
    - with just 68 bytes of extra storage per core

Outline

Motivation

Background

MAD Atomics

Deadlocks

Evaluation

Conclusions

Motivation

- Atomic read-modify-write (RMW) instructions
  - are the most efficient way to atomically update a variable

Motivation

- Atomic read-modify-write (RMW) instructions
    - are the most efficient way to atomically update a variable
- Non-blocking algorithms
    - rely on atomic RMW primitives
    - commonly, the compare-and-swap(CAS) instruction

Motivation

- Atomic read-modify-write (RMW) instructions
  - are the most efficient way to atomically update a variable
- Non-blocking algorithms
  - rely on atomic RMW primitives
  - commonly, the compare-and-swap(CAS) instruction
- In general, increase the scalability of commonly used data structures and applications

Previous Work

A hardware implementation of the MCAS synchronization primitive[1]

- ☺ MCAS table to setup the locks
- ☹ A set of instructions fill the structure, and later another one start locking the stored addresses
- ☹ Deadlocks due to resource limitations or lack of non-speculative solution.

[1] Patel et al, In 2017 Design, Automation, and Test in Europe (DATE)

[2] Ros and Kaxiras, ISCA 45, 2018

Previous Work

A hardware implementation of the MCAS synchronization primitive[1]

- ☺ MCAS table to setup the locks
- ☹ A set of instructions fill the structure, and later another one start locking the stored addresses
- ☹ Deadlocks due to resource limitations or lack of non-speculative solution.

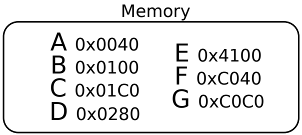
Non-Speculative Store Coalescing in Total Store Order[2]

- ☺ Limited resources are taken into account
- ☹ Atomic groups established arbitrarily, on conflict atomic groups are split
- ☹ Atomic groups for atomic operations are established by the programmer and cannot be split

[1] Patel et al, In 2017 Design, Automation, and Test in Europe (DATE)

[2] Ros and Kaxiras, ISCA 45, 2018

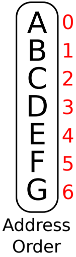
Background: Address versus Lexicographical Order



Memory

A 0x0040
B 0x0100
C 0x01C0
D 0x0280
E 0x4100
F 0xC040
G 0xC0C0

- Typical solution Address Order[1]



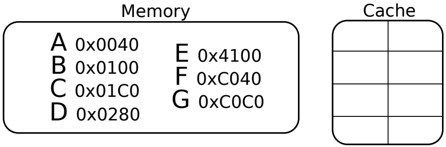
A 0
B 1
C 2
D 3
E 4
F 5
G 6

Address Order

[1] Dijkstra, EDW-310, E.W. Dijkstra Archive, Center for American History, 1971

[2] Ros and Kaxiras, ISCA 45, 2018

Background: Address versus Lexicographical Order

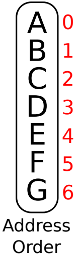


Memory

A	0x0040
B	0x0100
C	0x01C0
D	0x0280
E	0x4100
F	0xC040
G	0xC0C0

Cache

- Typical solution Address Order[1]
- Address order does not take into account some hardware structures like the cache



A	0
B	1
C	2
D	3
E	4
F	5
G	6

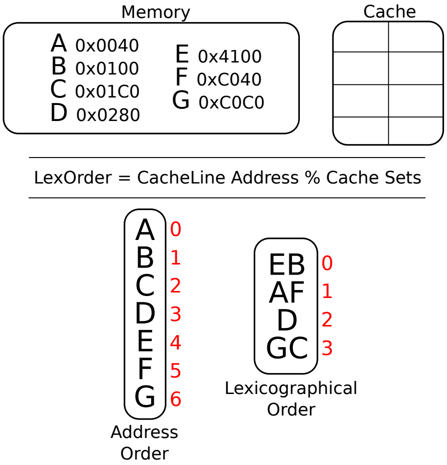
Address Order

[1] Dijkstra, EDW-310, E.W. Dijkstra Archive, Center for American History, 1971

[2] Ros and Kaxiras, ISCA 45, 2018

BACKGROUND: ADDRESS VERSUS LEXICOGRAPHICAL ORDER

- Typical solution Address Order[1]
- Address order does not take into account some hardware structures like the cache
- Lexicographical Order[2]



Memory

A	0x0040
B	0x0100
C	0x01C0
D	0x0280
E	0x4100
F	0xC040
G	0xC0C0

Cache

LexOrder = CacheLine Address % Cache Sets

Address Order

A	0
B	1
C	2
D	3
E	4
F	5
G	6

Lexicographical Order

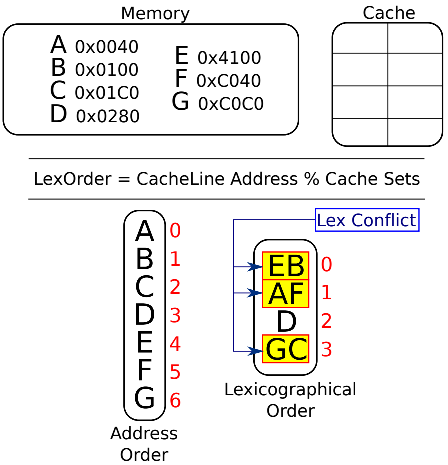
EB	0
AF	1
D	2
GC	3

[1] Dijkstra, EDW-310, E.W. Dijkstra Archive, Center for American History, 1971

[2] Ros and Kaxiras, ISCA 45, 2018

BACKGROUND: ADDRESS VERSUS LEXICOGRAPHICAL ORDER

- Typical solution Address Order[1]
- Address order does not take into account some hardware structures like the cache
- Lexicographical Order[2]



Memory

A 0x0040
B 0x0100
C 0x01C0
D 0x0280
E 0x4100
F 0xC040
G 0xC0C0

Cache

LexOrder = CacheLine Address % Cache Sets

A 0
B 1
C 2
D 3
E 4
F 5
G 6

Address Order

Lex Conflict

EB 0
AF 1
D 2
GC 3

Lexicographical Order

[1] Dijkstra, EDW-310, E.W. Dijkstra Archive, Center for American History, 1971

[2] Ros and Kaxiras, ISCA 45, 2018

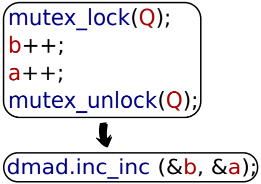
MAD ATOMICS

- Lock-protected critical sections

```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

MAD Atomics

- Lock-protected critical sections
- Single instructions multi-address atomics

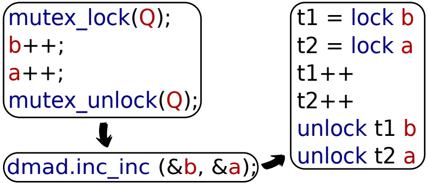


```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

```
dmad.inc_inc (&b, &a);
```

MAD Atomics

- Lock-protected critical sections
- Single instructions multi-address atomics
    - Decoded micro-ops



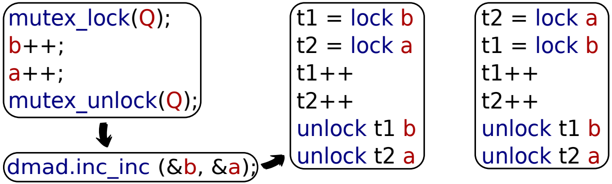
```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

```
dmad.inc_inc (&b, &a);
```

```
t1 = lock b
t2 = lock a
t1++
t2++
unlock t1 b
unlock t2 a
```

MAD Atomics

- Lock-protected critical sections
- Single instructions multi-address atomics
  - Decoded micro-ops
  - Out of Order execution



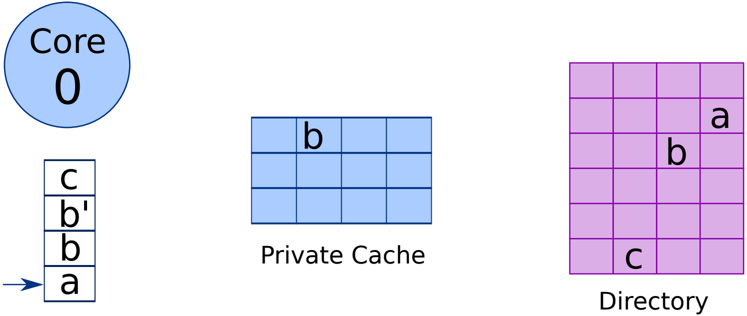
```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

```
dmad.inc_inc (&b, &a);
```

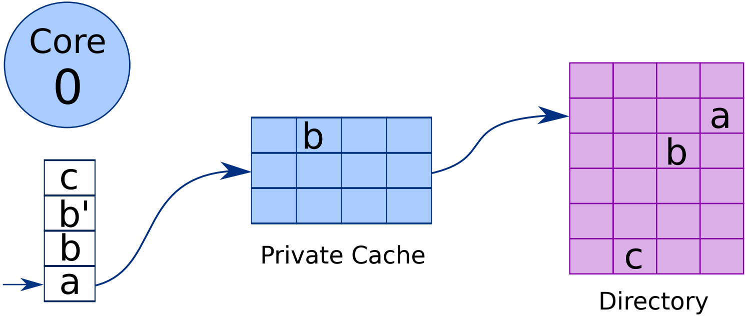
```
t1 = lock b
t2 = lock a
t1++
t2++
unlock t1 b
unlock t2 a
```

```
t2 = lock a
t1 = lock b
t1++
t2++
unlock t1 b
unlock t2 a
```

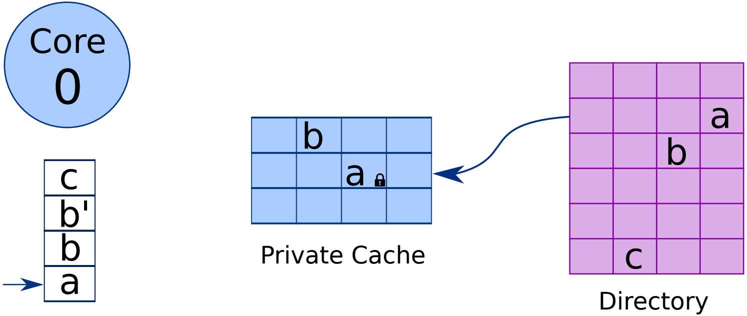
Running example



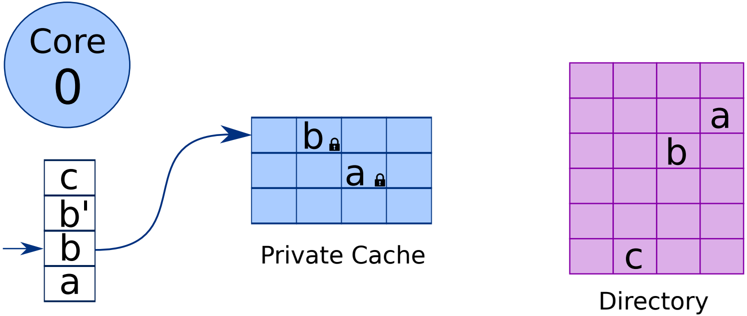
Running example



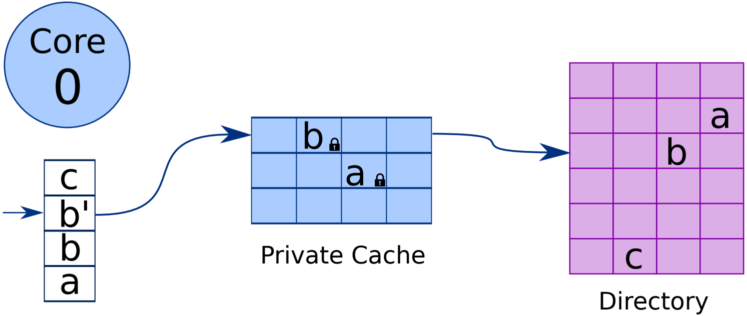
Running example



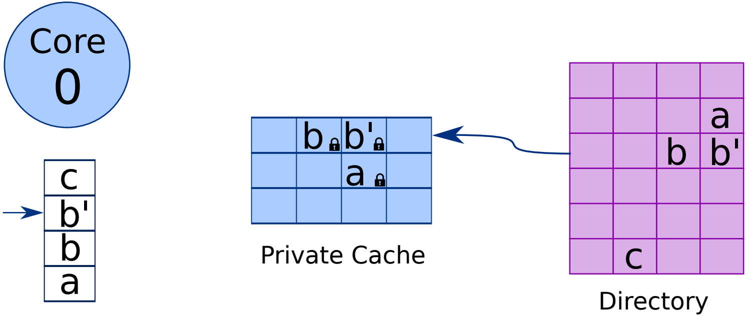
Running example



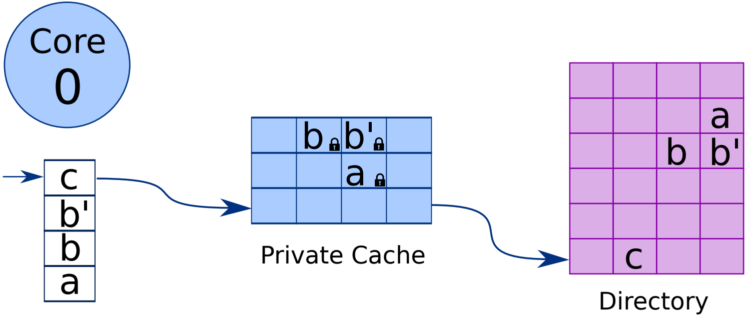
Running example



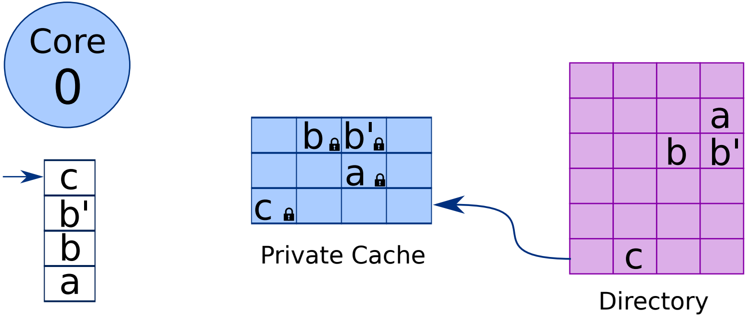
Running example



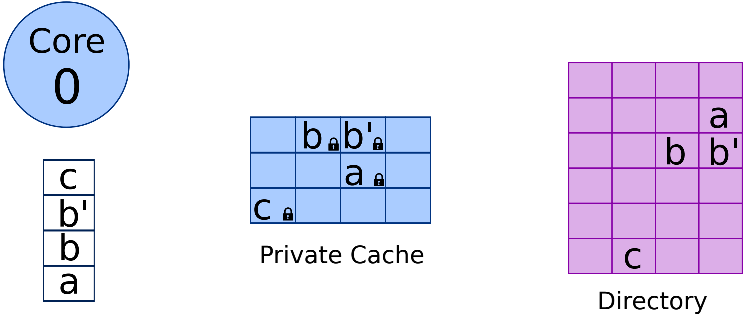
Running example



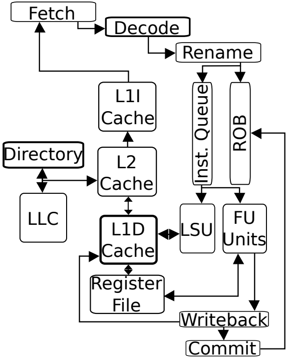
Running example



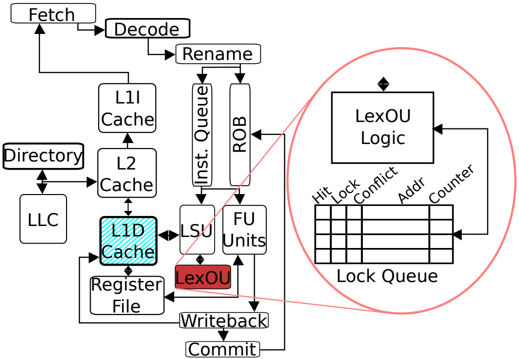
Running example



MAD atomics: Lex reOrder Unit (LexOU)

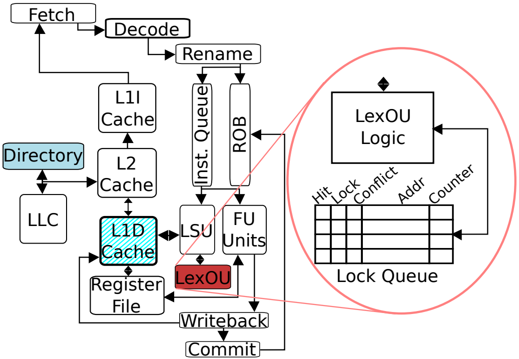


MAD atomics: Lex reOrder Unit (LexOU)



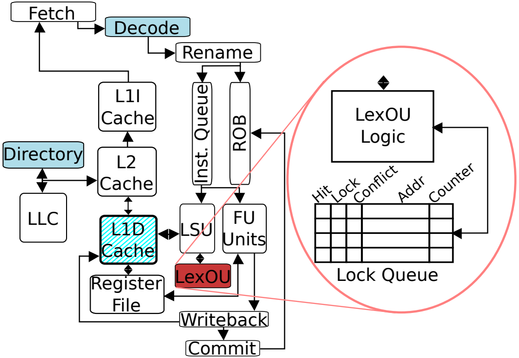
- Lexicographical reOrder Unit

MAD ATOMICS: LEX REORDER UNIT (LEXOU)



- Lexicographical reOrder Unit
- Extra bit at each set of the directory

MAD ATOMICS: LEX REORDER UNIT (LEXOU)



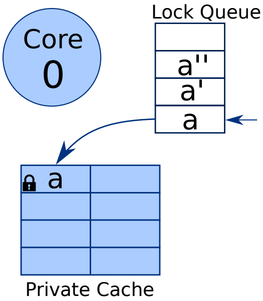
- Lexicographical reOrder Unit
- Extra bit at each set of the directory
- Load_locked & Store_unlock

Deadlocks

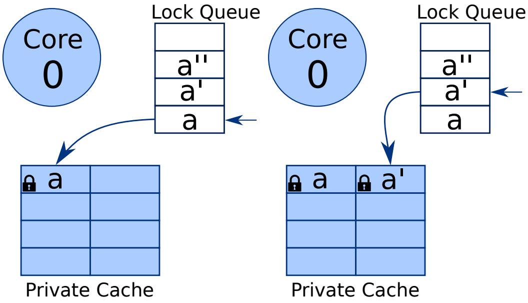
We have identified several deadlocks scenarios due to resource limitations:

- Private Cache
- Shared Cache
- Eviction Buffers

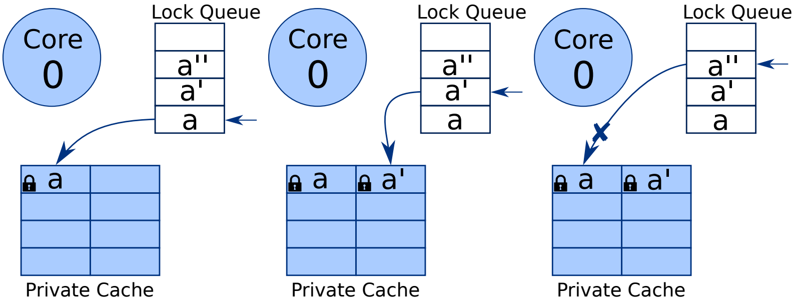
Deadlocks: Private Cache



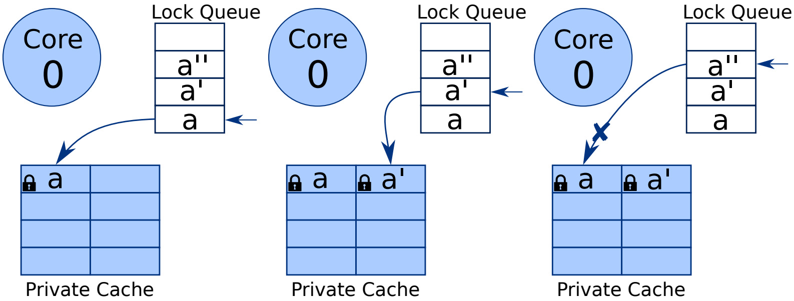
DEADLOCKS: PRIVATE CACHE



Deadlocks: Private Cache

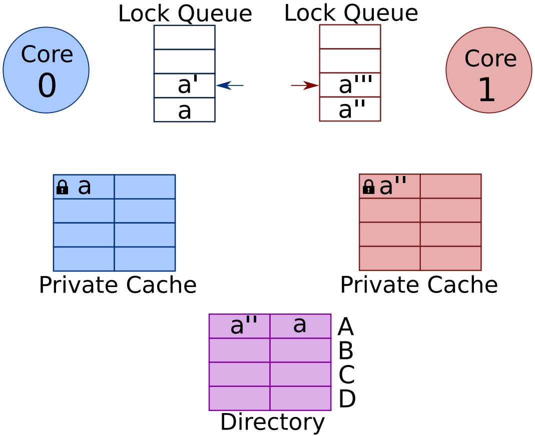


Deadlocks: Private Cache

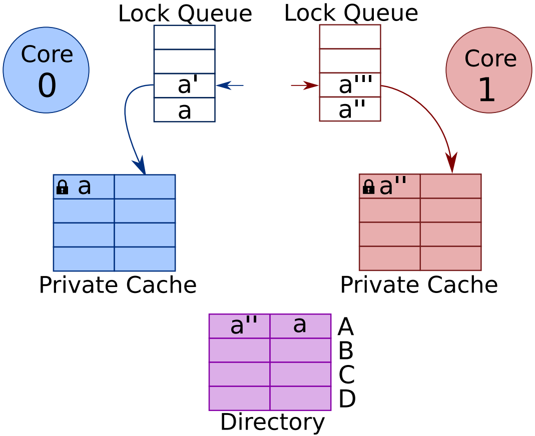


MAD atomics are limited to a maximum of 4 addresses

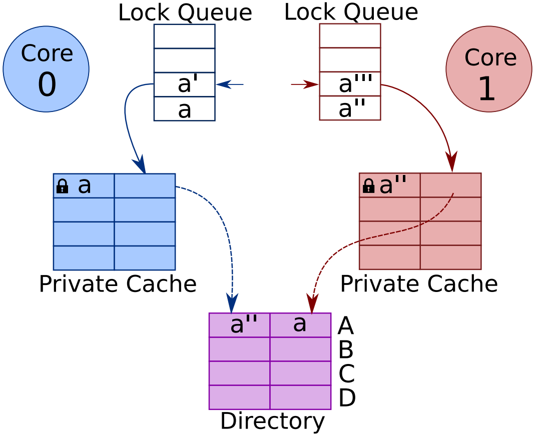
Deadlocks: Shared Cache



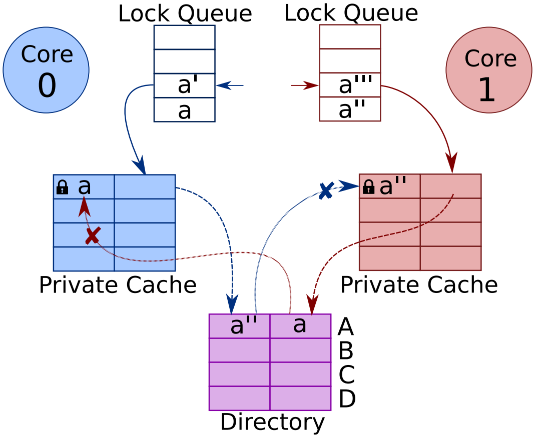
Deadlocks: Shared Cache



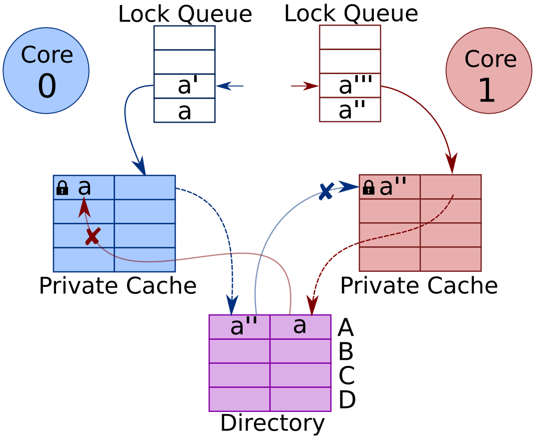
DEADLOCKS: SHARED CACHE



Deadlocks: Shared Cache

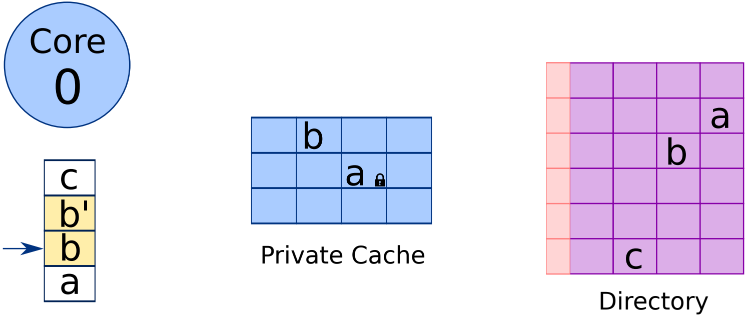


Deadlocks: Shared Cache

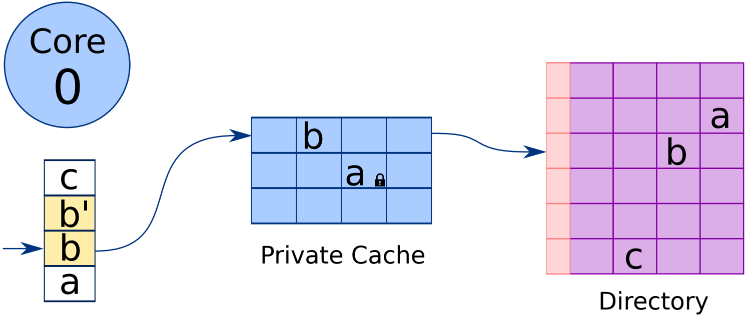


The set lock prevents multiple conflicts to clash in the same set

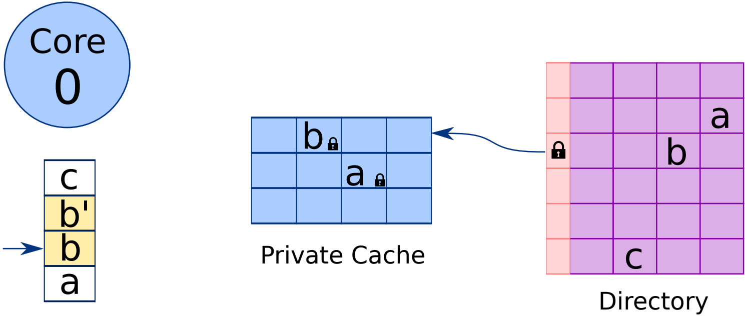
Deadlocks: Shared Cache Running Solution



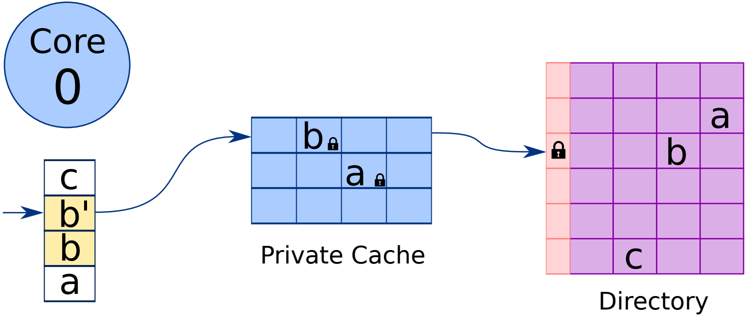
Deadlocks: Shared Cache Running Solution



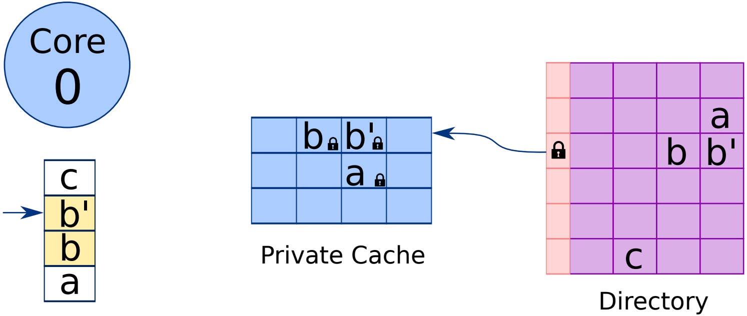
Deadlocks: Shared Cache Running Solution



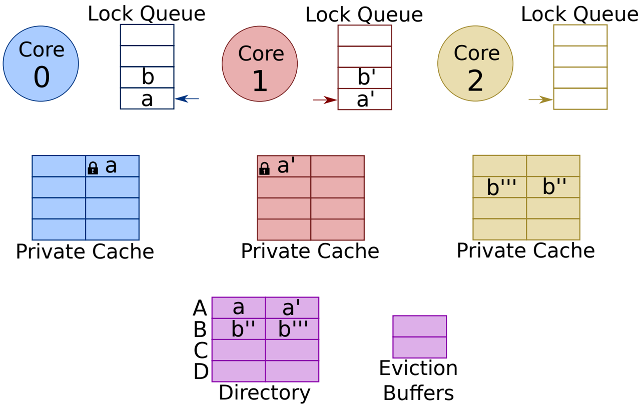
Deadlocks: Shared Cache Running Solution



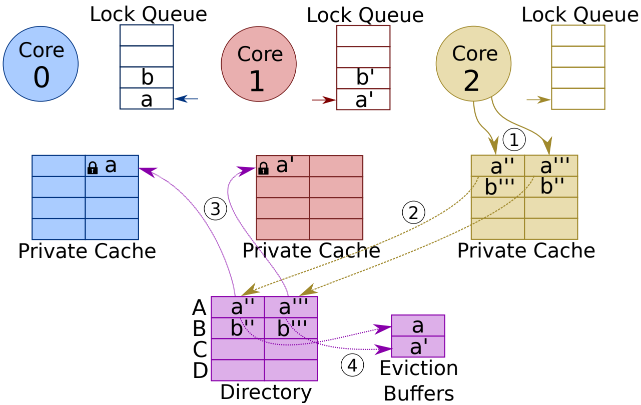
Deadlocks: Shared Cache Running Solution



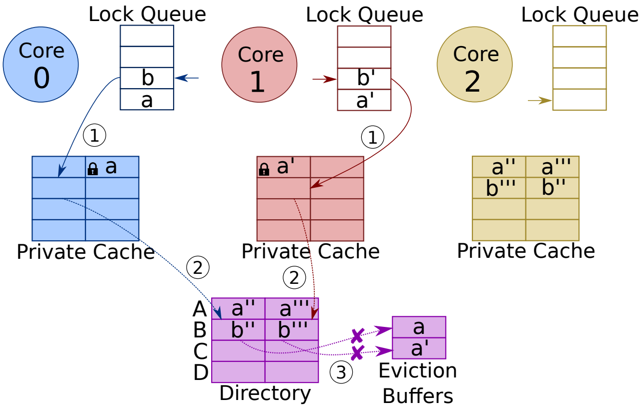
DEADLOCKS: EVICTION BUFFERS



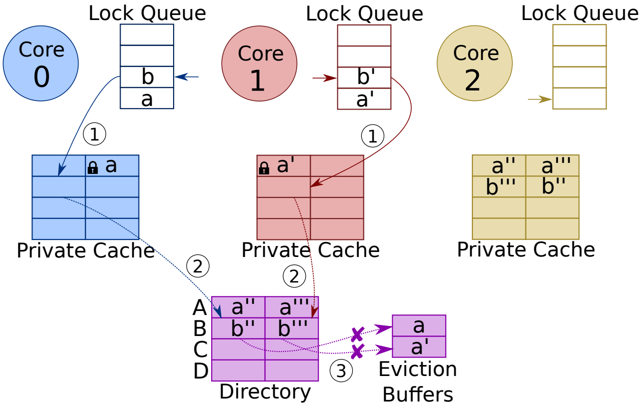
Deadlocks: Eviction Buffers



DEADLOCKS: EVICTION BUFFERS



DEADLOCKS: EVICTION BUFFERS



We propose to enable *in-situ* replacements in this scenario

EVALUATION: SIMULATOR

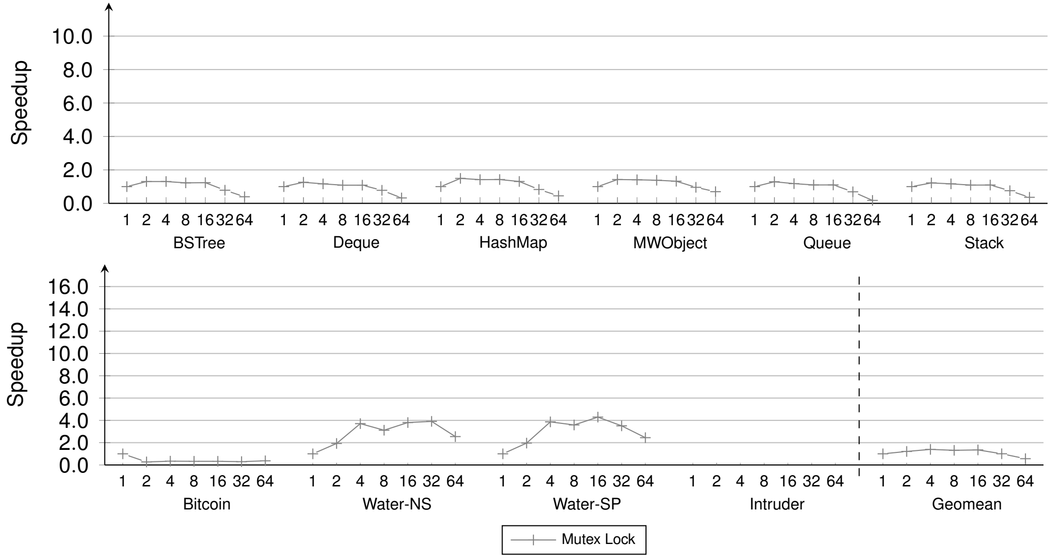
- Gem5-20 full system simulator
- Mimicking an Intel Skylake processor from 1 up to 64 cores
- Memory hierarchy and coherence protocol modeled with Ruby
- Execution and issue latencies modeled as measured on real hardware by Fog[1]

[1] Fog, `http://www.agner.org/optimize/instruction_tables.pdf`, 2018

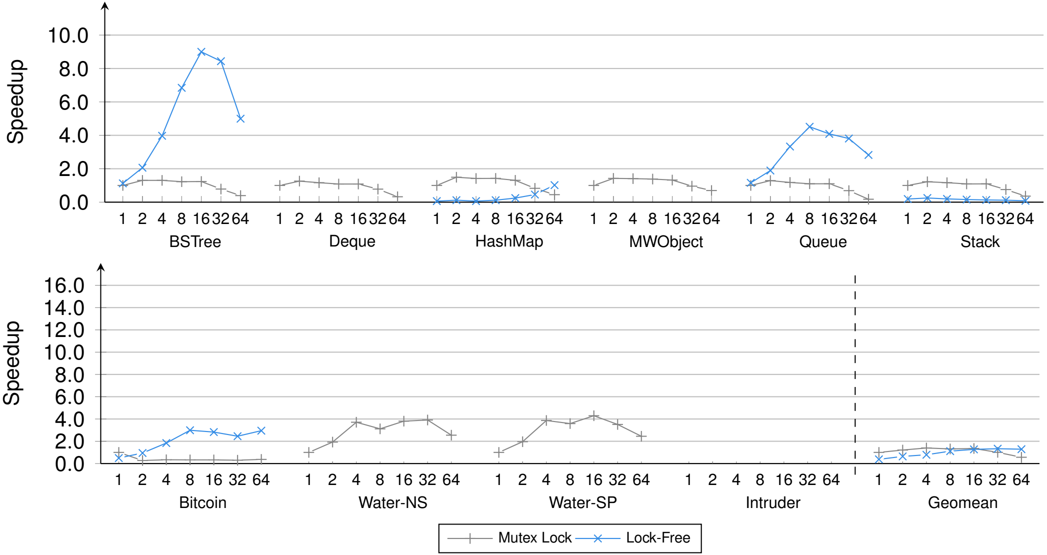
Evaluation: Benchmarks

- Commonly used concurrent data structures and some parallel applications
- Critical sections can be translated to two categories:
  - multi-address atomic operations
  - multi-address compare-and-swap (MCAS) operations

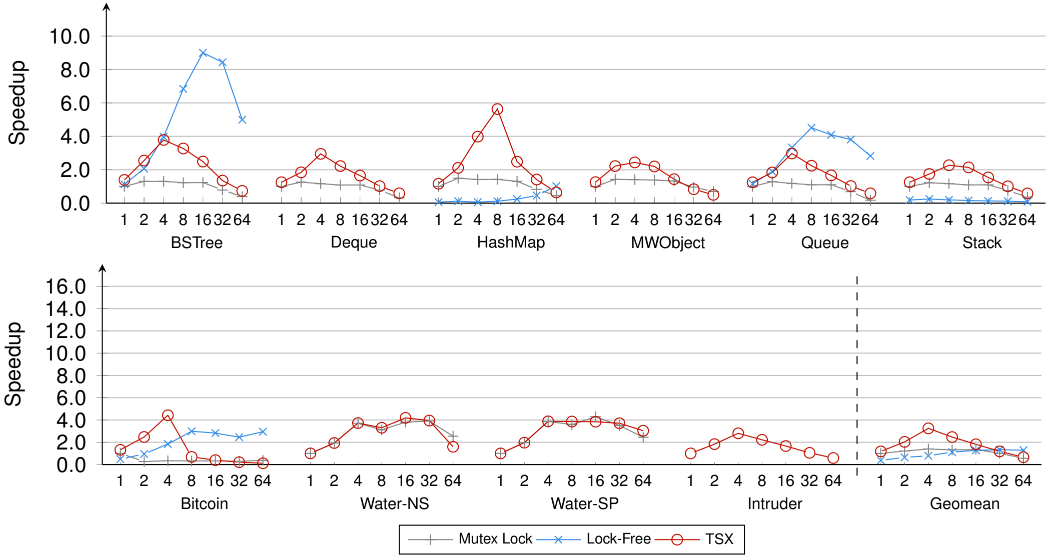
EVALUATION: RESULTS



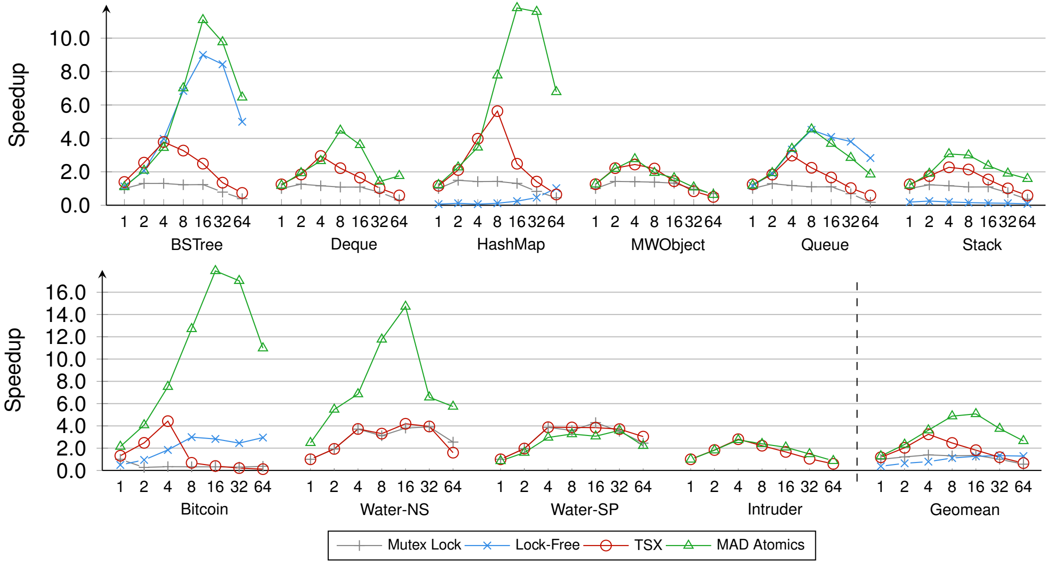
Evaluation: Results



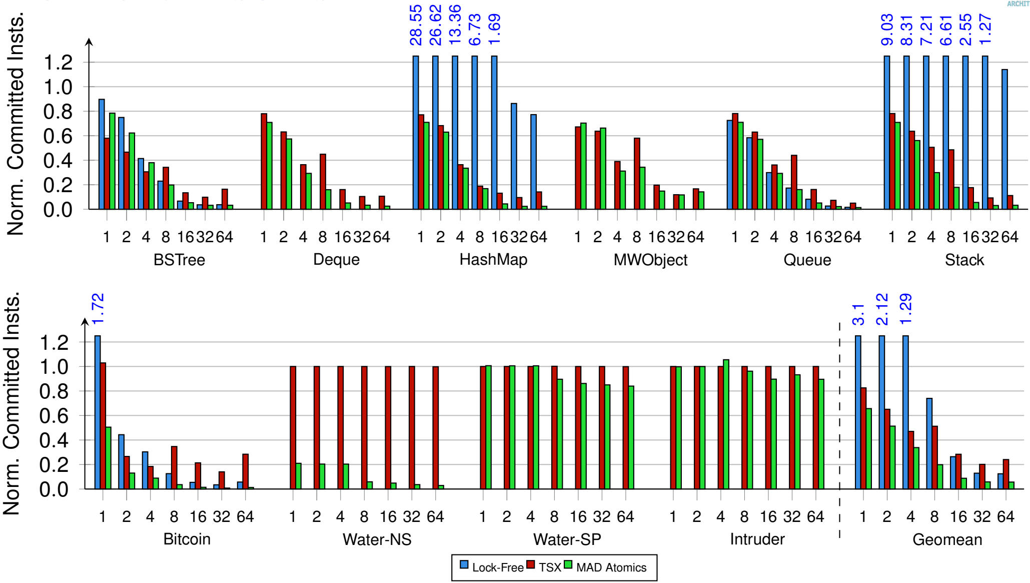
EVALUATION: RESULTS



EVALUATION: RESULTS



EVALUATION: RESULTS



Conclusion

- New efficient, more flexible, non-speculative, deadlock-free multi-address (MAD) atomic operations.

Conclusion

- New efficient, more flexible, non-speculative, deadlock-free multi-address (MAD) atomic operations.
- Avoid deadlocks due to limited resources relying only on the coherence protocol and a predetermined locking order

Conclusion

- New efficient, more flexible, non-speculative, deadlock-free multi-address (MAD) atomic operations.
- Avoid deadlocks due to limited resources relying only on the coherence protocol and a predetermined locking order
- Performance is increased:
  - 3.4× on average against software locks
  - 2.7× on average compared to TSX
  - In general improving scalability from one core (software locks) up to 16 cores.

CONCLUSION

- New efficient, more flexible, non-speculative, deadlock-free multi-address (MAD) atomic operations.
- Avoid deadlocks due to limited resources relying only on the coherence protocol and a predetermined locking order
- Performance is increased:
  - 3.4× on average against software locks
  - 2.7× on average compared to TSX
  - In general improving scalability from one core (software locks) up to 16 cores.

with just 68 bytes of extra storage per core

Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations

Eduardo José Gómez-Hernández[1] Juan M. Cebrian[1]
Rubén Titos-Gil[1] Stefanos Kaxiras[2] Alberto Ros[1]

eduardojose.gomez@um.es

Thank you for your attention!

ECHO, ERC Consolidator Grant (No 819134)
Vetenskapsradet project 2018-05254 and EPEEC (No 801051)