Advancements Towards non-Speculative Concurrent Execution of Critical Sections

Eduardo José Gómez Hernández

Advisors:
Alberto Ros Bardisa
Stefanos Kaxiras

Outline

1. Introduction
2. Objectives
3. Background
4. Methodology
5. Contribution 1: 1-Address Critical Sections (Splash-4)
6. Contribution 2: 4-Address Critical Sections (Hardware Multi-Address Atomics)
7. Contribution 3: n-Address Critical Sections (CLEAR Bounding TM to a single retry)
8. Conclusion a Future Lines

Introduction

- The increase of single core performance is no longer an easy task
- This is an issue to continue showing attractive improvements on each generation
- Chip Multi Processor (CMPs) is the paradigm switch that is currently being used
- Now each generation of chips includes more and more cores inside the same package

- These cores are identical, or at least very similar, stablishing the Symmetic Multi-Processing (SMP) as the today's standard of high-performance computing

Introduction

Antoher problem appear in the raise of CMPs and SMP, data coherence

- Chips already moved away from direct memory access as its speed was not able to catch up with the speed of the computing core

- Cores access a much faster but slower in chip memory called cache
- Each core has its own private cache, introducing the problem of the data coherency

Introduction

- Ideally, software should be able to use all the available cores and increase the performance in the same proportion.

- However, the real scenario is totally different
  - Applications require synchronization between different cores (either data or computing)

Introduction

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.

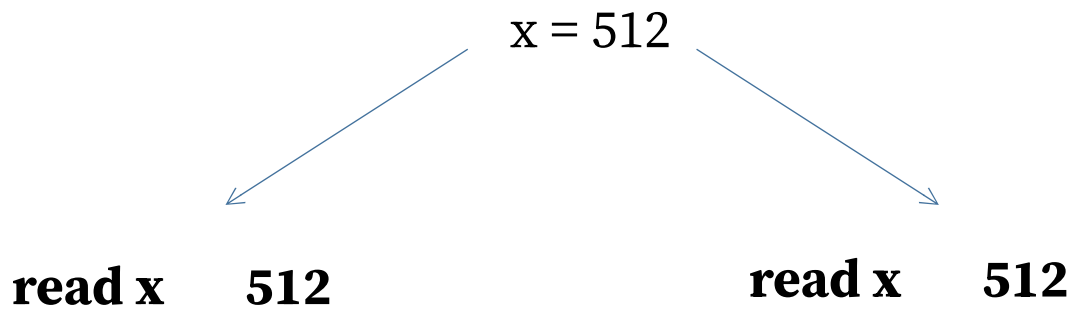
Introduction

x = 512

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.

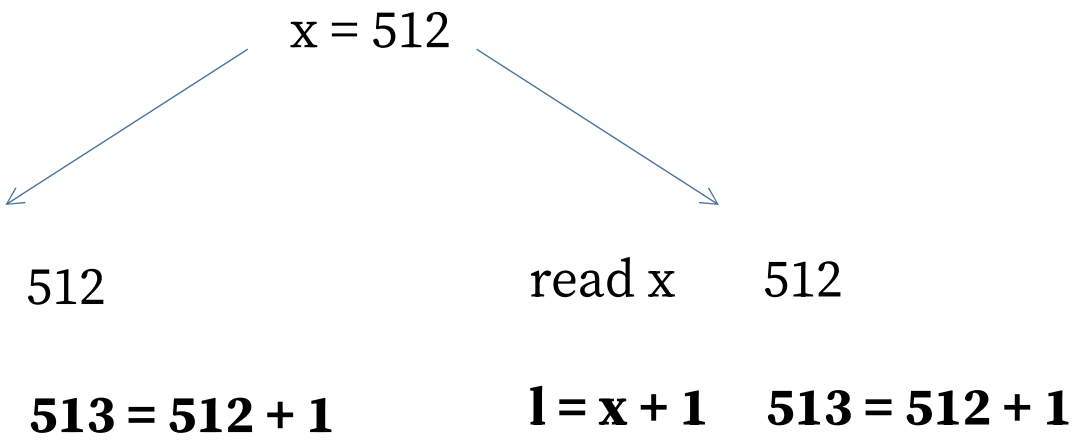
Introduction

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.



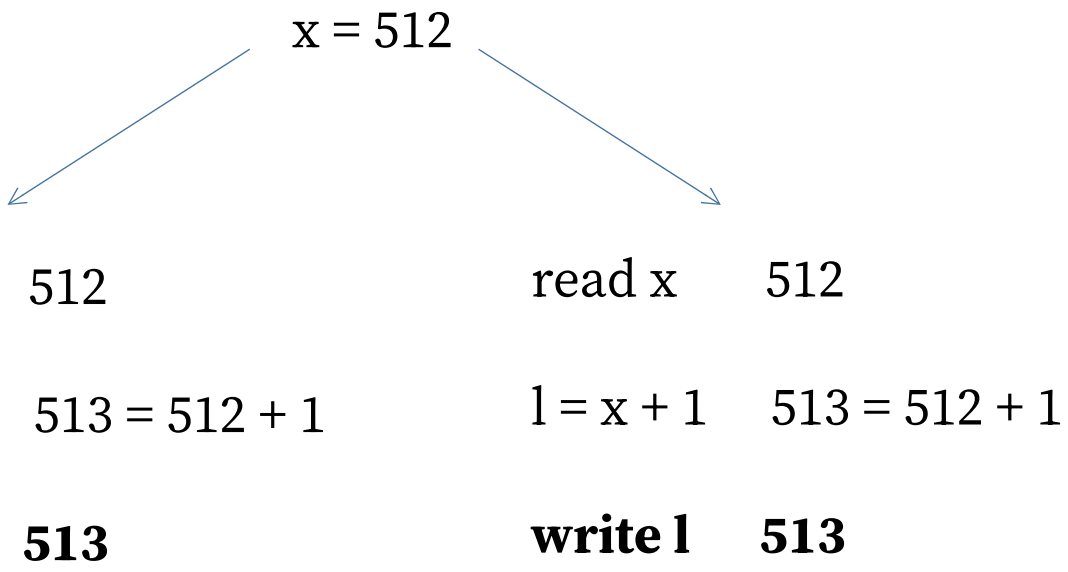
Introduction

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.



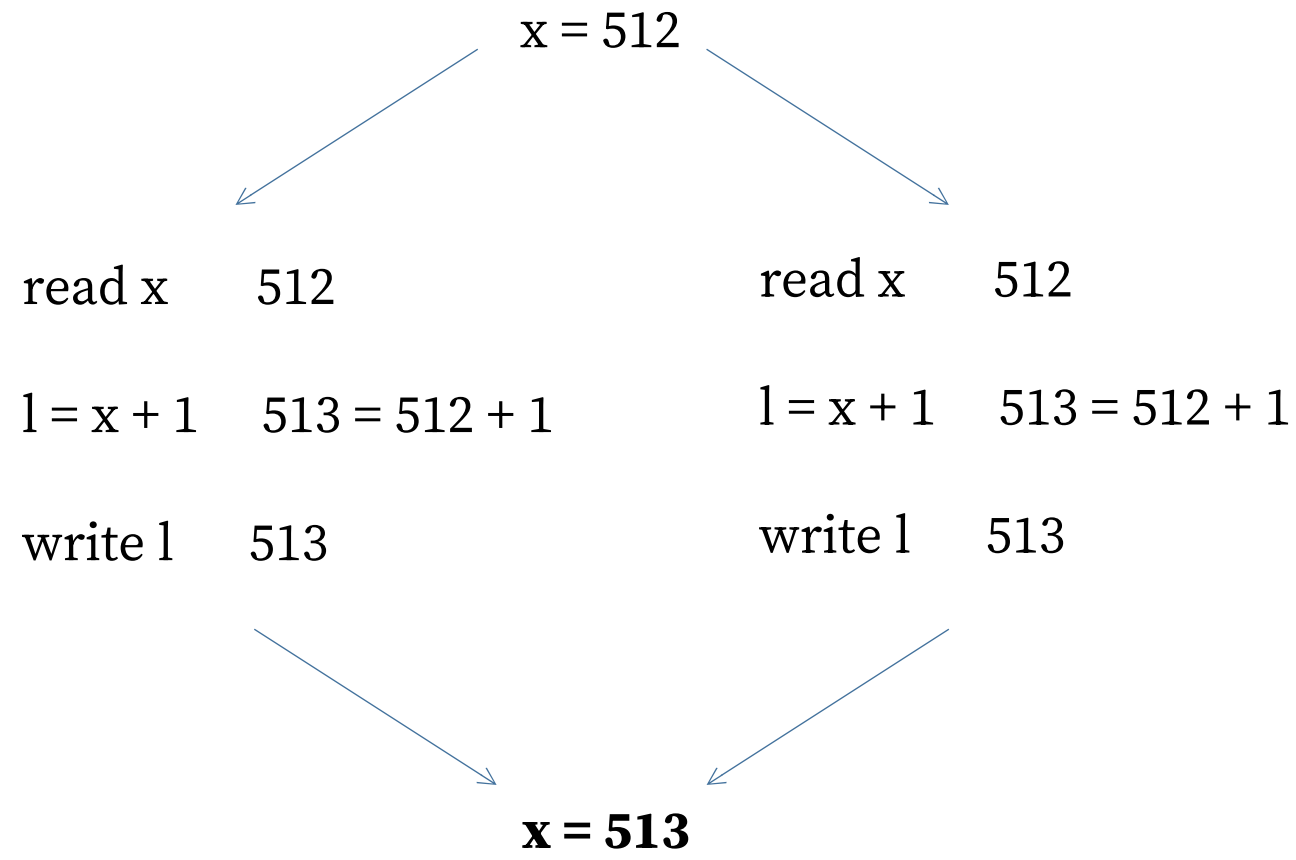
Introduction

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.



Introduction

But when working with multiple threads in the SMP model, with a Shared Memory Model, coherence is not enough.



Objective (Problem Statement)

The final objective of this thesis is to develop a non-speculative execution method that allows the

high-performance concurrent execution of critical sections

Objective (Problem Statement)

The final objective of this thesis is to develop a non-speculative execution method that allows the high-performance concurrent execution of critical sections

1. **What is the current state to evaluate the performance of critical sections?**

Objective (Problem Statement)

The final objective of this thesis is to develop a non-speculative execution method that allows the high-performance concurrent execution of critical sections

1. What is the current state to evaluate the performance of critical sections?
2. **Can we extend the best current solution to solve the issue?**

Objective (Problem Statement)

The final objective of this thesis is to develop a non-speculative execution method that allows the high-performance concurrent execution of critical sections

1. What is the current state to evaluate the performance of critical sections?
2. Can we extend the best current solution to solve the issue?
3. **Can we integrate the best solution techniques into a more generic approach?**

Background

To prevent this from happening, we have two main approaches

Background

To prevent this from happening, we have two main approaches

1. Protect the code region against concurrent execution

Background

To prevent this from happening, we have two main approaches

1. Protect the code region against concurrent execution

2. Protect the data against concurrent accesses/modifications

Background

To prevent this from happening, we have two main approaches

1. Protect the code region against concurrent execution

2. **Protect the data against concurrent accesses/modifications**

Previous approaches

Atomic Instructions	Lock Free	Mutual Exclusion	Speculative Approaches

Previous approaches

Atomic Instructions | Lock Free | Mutual Exclusion | Speculative Approaches

- Widely supported by modern languages and architectures
- The most efficient way of performing an operation atomically
  but limited to a single memory location
- Commonly implemented in hardware: cache locking

Previous approaches

Atomic Instructions | Lock Free | Mutual Exclusion | Speculative Approaches

- Based on constructs developed over atomic instructions

- Challenging and error prone

Previous approaches

Atomic Instructions	Lock Free	Mutual Exclusion	Speculative Approaches

- Protect critical sections using locks (or mutexes)
- Simple and easy to implement, but at the cost of concurrency

Previous approaches

Atomic Instructions	Lock Free	Mutual Exclusion	Speculative Approaches

- Speculative Lock Elision (SLE), Hardware Transactional Memory (HTM)
- Extract parallelism when the ARs* are not conflicting
- Retry if conflicts: Limit number of retries + alternative not-concurrent path

* ARs (or Atomic Regions) a unified view of Critical Section and Transactions

Methodology

Two environments have been used in this thesis:

- Gem5 Simulator
- Real Machine
  - AMD EPYC 7702P (64 cores @ 2GHz)
  - 32KiB L1 D and I caches
  - 512KiB L2
  - 16MB L3

All applications have been run multiple times measuring the Region of Interest

Gem5 Version	Paper I: Gem5-20, Paper II: Gem5-19, Paper II: Gem5-21
Core	32-core out-of-order Icelake-like (Skylake-like in Paper II). Fetch/Decode/Rename width: 5 instructions per cycle; Issue/Commit width: 10 instructions per cycle; ROB: 352 uops (224 in Paper II); LQ: 128 entries (72 in Paper II); SQ: 72 entries (56 in Paper II); RAS: 64 entries (16 in Paper I and II); Branch predictor: LTAGE (TAGE_SC_L_64K in Paper I and II)
L1 Cache	Instructions: 32KiB, 8-way, 1-cycle access latency; Data: 48KiB (32KiB in Paper II), 12-way (8-way in Paper II), 1-cycle access latency.
L2 Cache	512KiB (256KiB in Paper II), 8-way, 10-cycle access latency.
L3 Cache	4MiB (2MiB in Paper II), 16-way, 45-cycle access latency.
Memory	80-cycle access latency.
Coherence	Three-level MESI protocol interconnected with a crossbar. Directory has 800% coverage.
HTM	Intel TSX-like requester wins, and Power-TM. Best of 1 to 10 retries before taking the fallback lock.

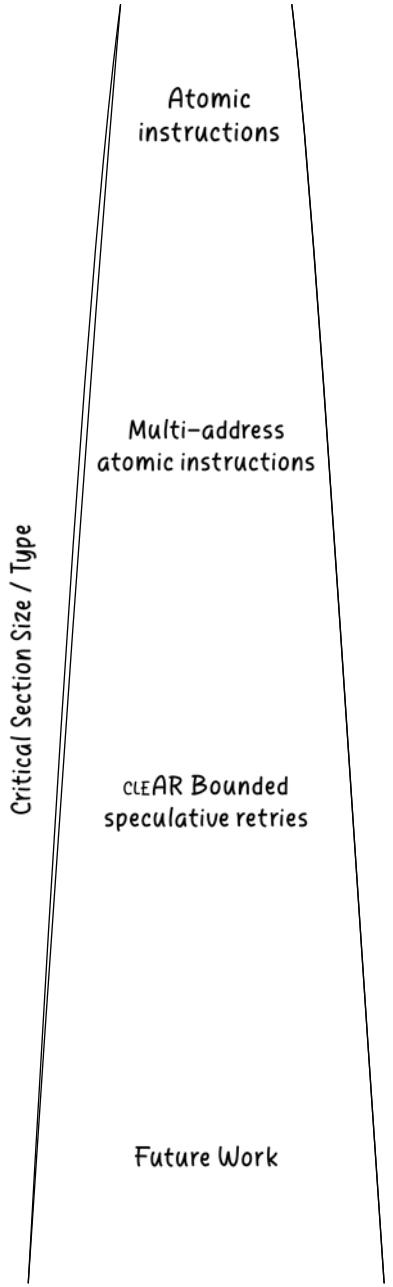
Summary

1-address immutable critical sections → (I) SPLASH-4: An update on the SPLASH Benchmark Suite focus on synchronization

4-addresses immutable critical sections → (II) MAD Atomics: A general method to perform non-speculative updates up to 4 addresses

n-addresses immutable/partially mutable critical sections → (III) CLEAR: Limit the maximum retries of SLE and HTM by cache locking and non-speculative execution

n-addresses immutable/mutable critical sections → Future Work: Allow the non-speculative execution of any critical section

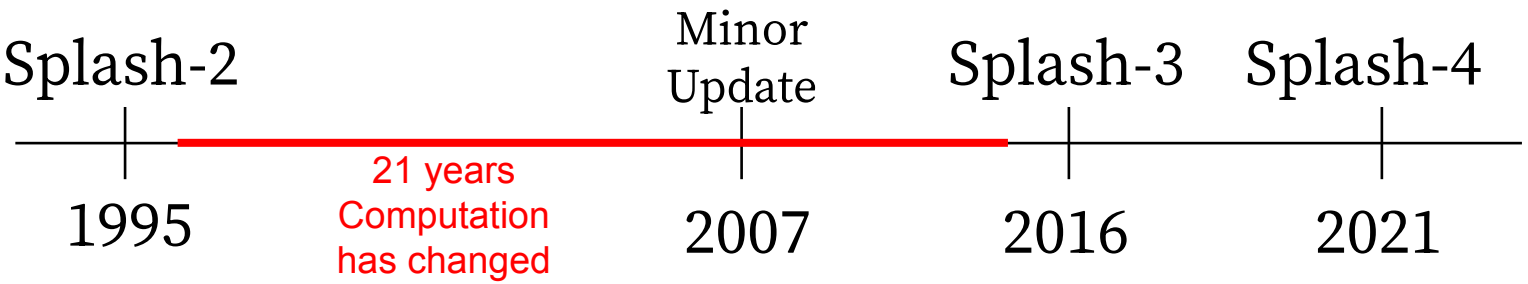


Published at:
- ISPASS 2021
- IISWC 2022

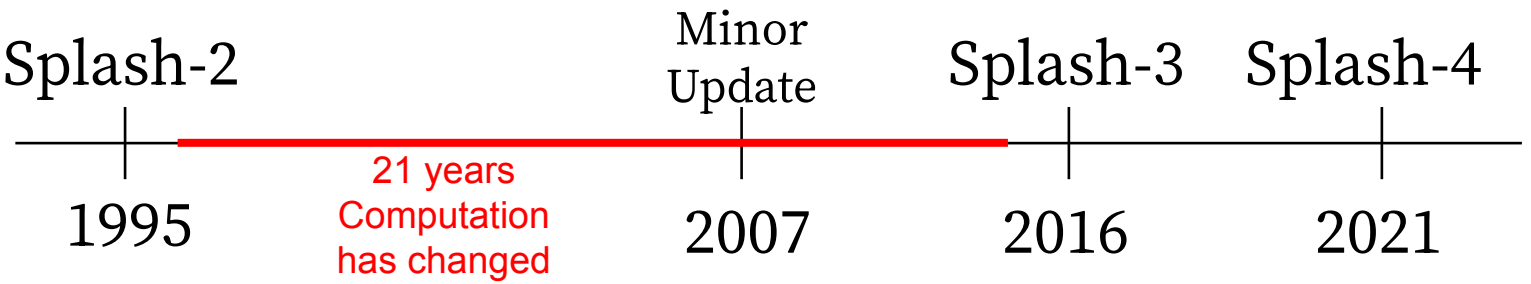
Splash-4

Let's start with single-address atomics!

Splash-4 - Splash History



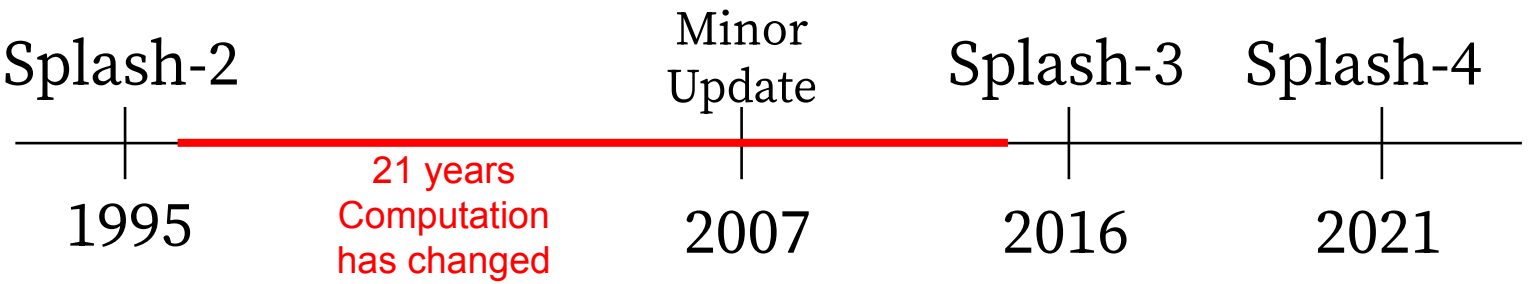
Splash-4 - Splash History



The first major parallel benchmark suite. Still in use (+5k cites)

1. Woo, Steven Cameron, et al, "The SPLASH-2 programs: Characterization and methodological considerations." ACM SIGARCH computer architecture news 23, 1995

Splash-4 - Splash History

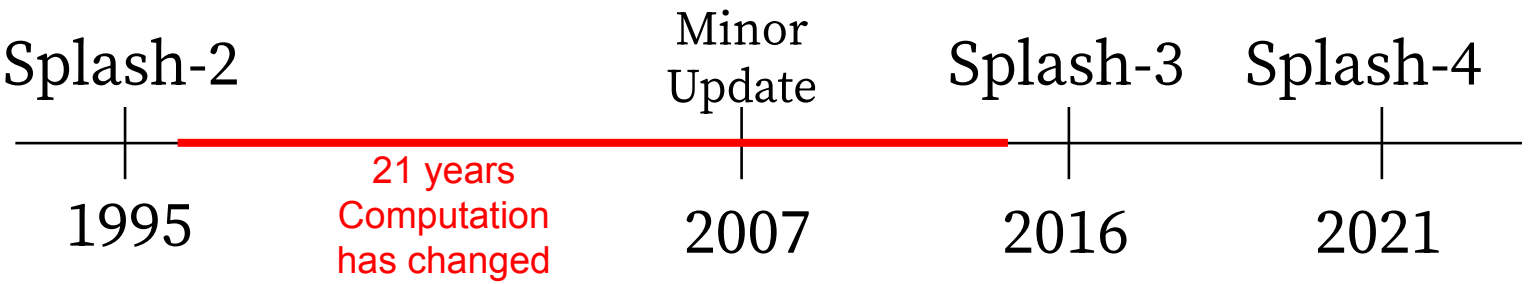


The first major parallel benchmark suite. Still in use (+5k cites)

A small update that fixes bugs and updates the programming style

1. Woo, Steven Cameron, et al, "The SPLASH-2 programs: Characterization and methodological considerations." ACM SIGARCH computer architecture news 23, 1995
2. Venetis, Ioannis E., et al, "The Modified SPLASH-2", https://www.capsl.udel.edu//splash/ 2007

Splash-4 - Splash History



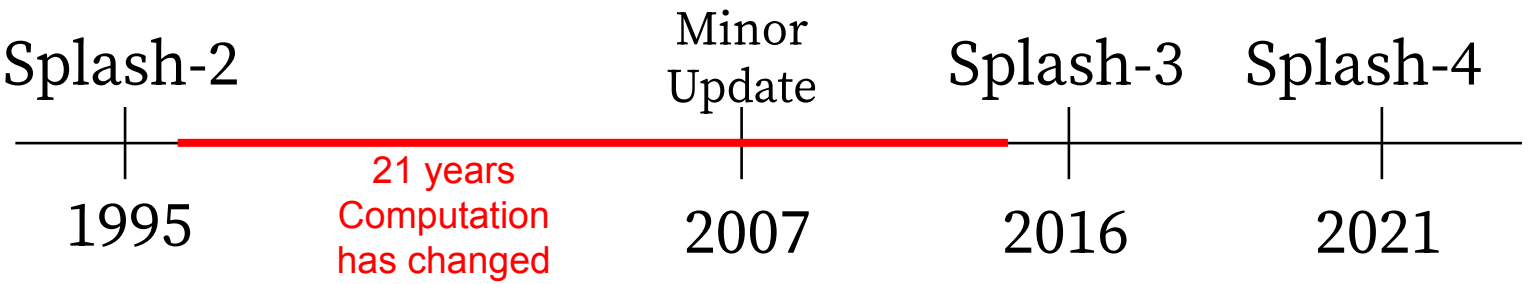
The first major parallel benchmark suite. Still in use (+5k cites)

A small update that fixes bugs and updates the programming style

First major update that fixes data races and performance bugs

1. Woo, Steven Cameron, et al, "The SPLASH-2 programs: Characterization and methodological considerations." ACM SIGARCH computer architecture news 23, 1995
2. Venetis, Ioannis E., et al, "The Modified SPLASH-2", https://www.capsl.udel.edu//splash/ 2007
3. Sakalis, Christos, et al, "Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research", ISPASS 2016

Splash-4 - Splash History



The first major parallel benchmark suite. Still in use (+5k cites)

A small update that fixes bugs and updates the programming style

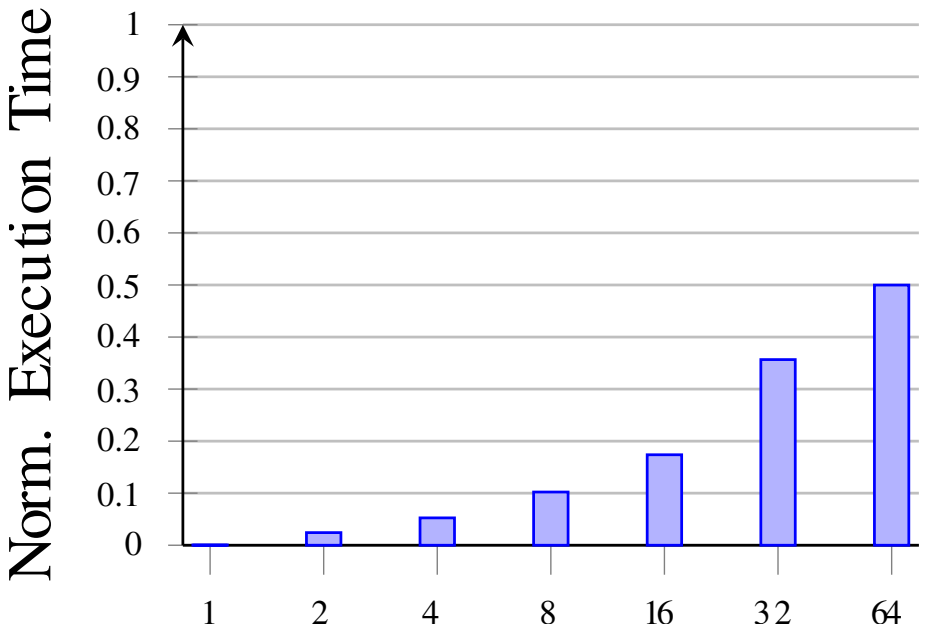
First major update that fixes data races and performance bugs

Current update, exploiting lockfree and atomic operations

1. Woo, Steven Cameron, et al, "The SPLASH-2 programs: Characterization and methodological considerations." ACM SIGARCH computer architecture news 23, 1995
2. Venetis, Ioannis E., et al, "The Modified SPLASH-2", https://www.capsl.udel.edu//splash/ 2007
3. Sakalis, Christos, et al, "Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research", ISPASS 2016
4. Gómez-Hernández, Eduardo José et al, "Splash-4: Improving Scalability with Lock-Free Constructs", ISPASS 2021

Splash-4 - Sync Overhead

- Splash-2 and Splash-3 are crafted using outdated programming techniques
- Previous works noticed that the default input sizes limit the scalability of some applications.
- The computation between synchronization points is not substantially longer than the synchronization
- Using larger datasets increases the execution time, and that is a problem when using simulation infrastructures



Splash-4 - Splash-3 critical section status

- Modern ISAs typically provide a basic set of atomic operations that offer both atomicity and synchronization
- This basic set consists of atomic loads and stores, atomic read-modify-write (RMW) operations, and some atomic comparisons and exchange operations
- Typical hardware RMW atomics are only available for integer types

Application	Barriers		Critical Sections						Conditionals					
			Mutex		C11		CAExch		Wait		Signal		Broad	
	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn
Splash-3														
Barnes	6	19	10	2140090	0	0	0	0	1	360	0	0	2	23539
Cholesky	4	6	8	95182	0	0	0	0	1	4588	1	20508	0	0
Fft	7	9	1	64	0	0	0	0	0	0	0	0	0	0
Fmm	13	36	38	488126	0	0	0	0	8	1467	1	6207	5	23282
Lu	5	69	1	64	0	0	0	0	0	0	0	0	0	0
Lu-NonContiguous	5	69	1	64	0	0	0	0	0	0	0	0	0	0
Ocean	20	902	4	13312	0	0	0	0	0	0	0	0	0	0
Ocean-NonContiguous	19	872	4	13312	0	0	0	0	0	0	0	0	0	0
Radiosity	5	12	48	3861123	0	0	0	0	0	0	0	0	0	0
Radix	7	17	1	64	0	0	0	0	0	0	0	0	0	0
Raytrace	3	3	8	355184	0	0	0	0	0	0	0	0	0	0
Volrend	15	146	12	311164	0	0	0	0	0	0	0	0	0	0
Water-Nsquared	9	22	8	68672	0	0	0	0	0	0	0	0	0	0
Water-Spatial	9	22	6	1217	0	0	0	0	0	0	0	0	0	0

Splash-4 - Replacing critical sections

```
/* CAExch */
var oldValue = LOAD(ptr);
var newValue;
do {
  newValue = new;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue,
    newValue)) != oldValue);
```

Splash-4 - Replacing critical sections

```
/* CAExch */
var oldValue = LOAD(ptr);
var newValue;
do {
  newValue = new;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue,
    newValue)) != oldValue);
```

```
double oldValue = LOAD(ptr);
double newValue;
do {
 newValue = oldValue + addition;
} while (!CAExch(ptr, oldValue, newValue));
```

Splash-4 - Replacing critical sections

```
/* CAExch */
var oldValue = LOAD(ptr);
var newValue;
do {
  newValue = new;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue,
    newValue)) != oldValue);
```

```
/* Lock */
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
  multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)
```

```
/* Lock-free */
double expected = LOAD(multi->err_multi);
do {
  if (local_err <= expected) break;
} while (!CAExch(multi->err_multi,
    expected, local_err));
```

```
double oldValue = LOAD(ptr);
double newValue;
do {
 newValue = oldValue + addition;
} while (!CAExch(ptr, oldValue, newValue));
```

Splash-4 - Replacing critical sections

```
/* CAExch */
var oldValue = LOAD(ptr);
var newValue;
do {
  newValue = new;
} while (!CAExch(ptr, oldValue, newValue))
    ;
```

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue,
    newValue)) != oldValue);
```

```
double oldValue = LOAD(ptr);
double newValue;
do {
 newValue = oldValue + addition;
} while (!CAExch(ptr, oldValue, newValue));
```

```
/* Lock */
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
  multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)
```

```
/* Lock-free */
double expected = LOAD(multi->err_multi);
do {
  if (local_err <= expected) break;
} while (!CAExch(multi->err_multi,
    expected, local_err));
```

```
/* Lock */
LOCK(gl->PotengSumLock);
*POTA = *POTA + LPOTA;
*POTR = *POTR + LPOTR;
*PTRF = *PTRF + LPTRF;
UNLOCK(gl->PotengSumLock);
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(POTA, LPOTA);
FETCH_AND_ADD_DOUBLE(POTR, LPOTR);
FETCH_AND_ADD_DOUBLE(PTRF, LPTRF);
```

Splash-4 - Atomic Barrier

Sense-Reversing Centralized Barrier

- Optimized for short waits
- Spinloops on a variable (only reading)
- A write will trigger all threads to exit the spinloop

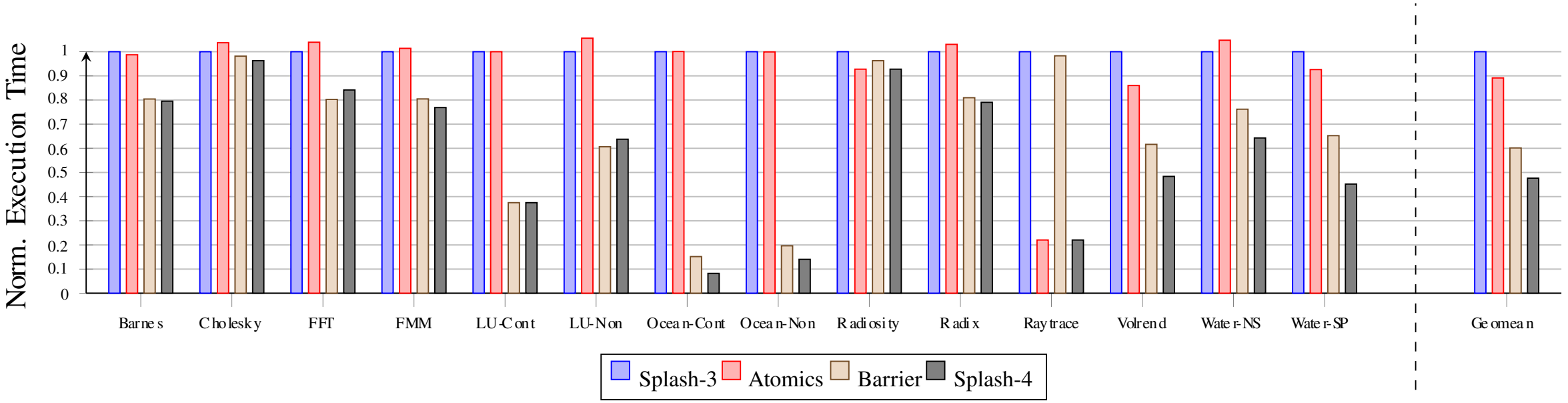
```
1 local_sense = !local_sense;
2 if (atomic_fetch_sub(&(count), 1) == 1) {
3     count = threads;
4   STORE(sense, local_sense);
5 } else {
6   do {} while (LOAD(sense) != local_sense);
7 }
```

Splash-4 - Critical section status

Application	Barriers		Critical Sections						Conditionals					
			Mutex		C11		CAExch		Wait		Signal		Broad	
	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn	St	Dyn
Splash-4														
Barnes	6	19	9	2140056	1	64	0	0	1	352	0	0	2	23539
Cholesky	4	6	6	68979	1	64	1	26238	1	3911	1	20508	0	0
Fft	7	9	0	0	1	64	0	0	0	0	0	0	0	0
Fmm	13	36	26	442838	1	64	1	5	8	1485	1	6207	5	23282
Lu	5	69	0	0	1	64	0	0	0	0	0	0	0	0
Lu-NonContiguous	5	69	0	0	1	64	0	0	0	0	0	0	0	0
Ocean	20	902	0	0	1	64	3	13248	0	0	0	0	0	0
Ocean-NonContiguous	19	872	0	0	1	64	3	13248	0	0	0	0	0	0
Radiosity	5	12	36	3478298	3	50497	3	6394618	0	0	0	0	0	0
Radix	7	17	0	0	1	64	0	0	0	0	0	0	0	0
Raytrace	3	3	2	252498	5	92455	1	8816	0	0	0	0	0	0
Volrend	15	146	1	1536	8	245519	0	0	0	0	0	0	0	0
Water-Nsquared	9	22	0	0	1	64	15	608384	0	0	0	0	0	0
Water-Spatial	9	22	0	0	1	64	6	1280	0	0	0	0	0	0

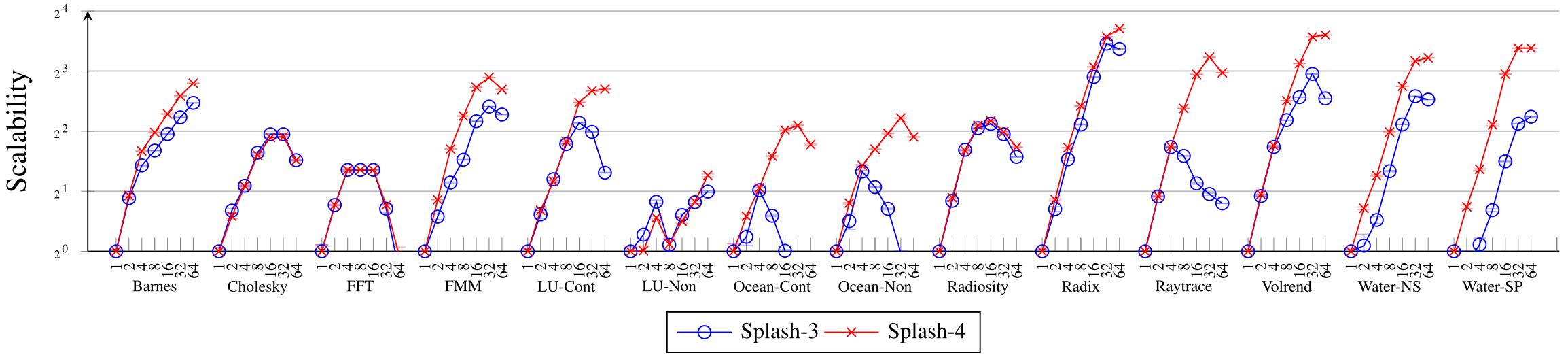
Splash-4 reimplemented a significant portion of critical sections from Mutex into C11 atomics and Atomic Constructs (CAExch)

Splash-4



Splash-4 reduces 50% the average execution time on a real modern 64-core system

Splash-4



Splash-4 boosts the scalability of most applications in the Splash benchmark suite up to 32 cores

Published at:
- MICRO 2020

Hardware Multi-Address Atomics

If 1 was not enough, let's go with 4!
4-Address Atomics (MADs)

Hardware Multi-Address Atomics - Introduction

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

Hardware Multi-Address Atomics - Introduction

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

- Ideally multi-address atomics should be:
    1. **Fine-grained** locking to enable **concurrency**
    2. **non-speculative** to prevent **retries** (re-executions/aborts)

Hardware Multi-Address Atomics - Background

A hardware implementation of the MCAS synchronization primitive[1]

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

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

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

1 Patel et al, DATE 2017
2 Ros and Kaxiras, ISCA 2018

Hardware Multi-Address Atomics - Usage

- What if these three variables are no independent?
- Can we design a non-speculative solution that is able to solve the problem?

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

Hardware Multi-Address Atomics - Usage

- What if these three variables are no independent?
- Can we design a non-speculative solution that is able to solve the problem?
- **YES!**
- How?:
  - Locking the three addresses
  - Without deadlocks

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

Hardware Multi-Address Atomics - Usage

- What if these three variables are no independent?
-
- Can we design a non-speculative solution that is able to solve the problem?

- **YES!**

- How?:
  - Locking the three addresses
  - Without deadlocks

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

```
1 /* Custom Atomic */
2 TFETCH_AND_ADD_DOUBLE(POTA, LPOTA, POTR, LPOTR, PTRF, LPTRF);
```

```
1 /* MCAS */
2 do {
3   pPOTA = LOAD(POTA);
4   pPOTR = LOAD(POTR);
5   pPTRF = LOAD(PTRF);
6   nPOTA = pPOTA + LPOTA;
7   nPOTR = pPOTR + LPOTR;
8   nPTRF = pPTRF + LPTRF;
9 } while(!TCAS(POTA, pPOTA, nPOTA, POTR, pPOTR, LPOTR, PTRF, pPTRF, nPTRF));
```

Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

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

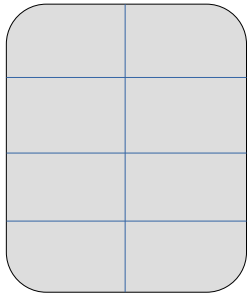
Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

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

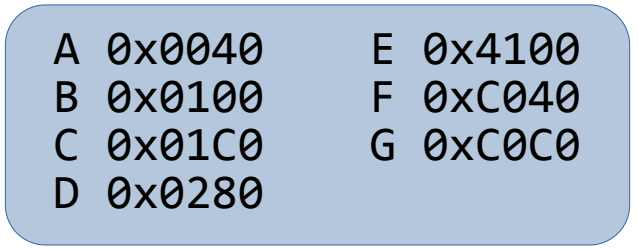
Address Order

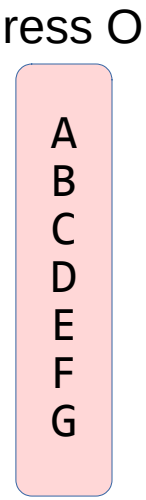
```
A
B
C
D
E
F
G
```

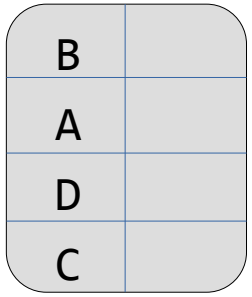


Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

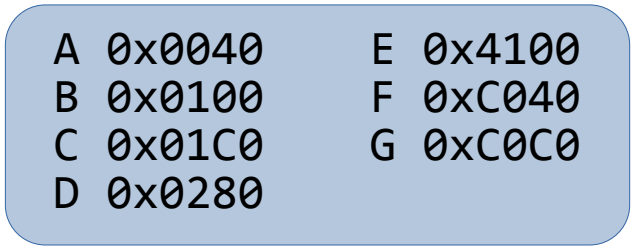




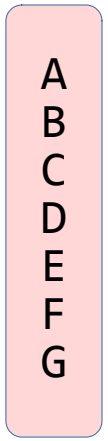


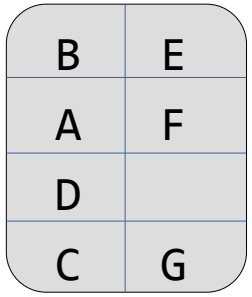
Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.



Address Order



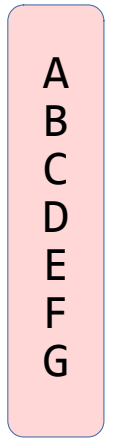


Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

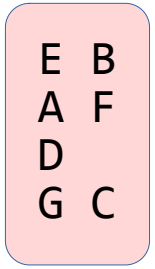
```
A 0x0040    E 0x4100
B 0x0100    F 0xC040
C 0x01C0    G 0xC0C0
D 0x0280
```

Address Order



B	E
A	F
D	
C	G

LexOrder



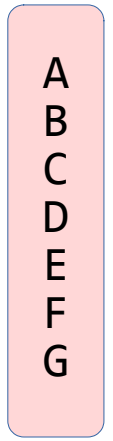

```
LexOrder = CacheLine Address % Cache Sets
```

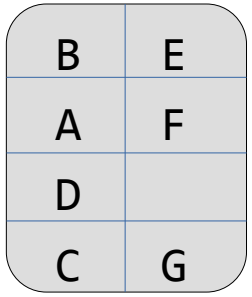
Hardware Multi-Address Atomics - LexOrder

- Lexicographical Order (lexorder) takes into account the limited resources.
- This new order was tailored to only take into account the cache sets.
- In this specific way, cache is forced to be filled from set 0 to set n.

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

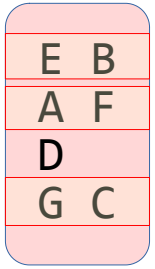
Address Order





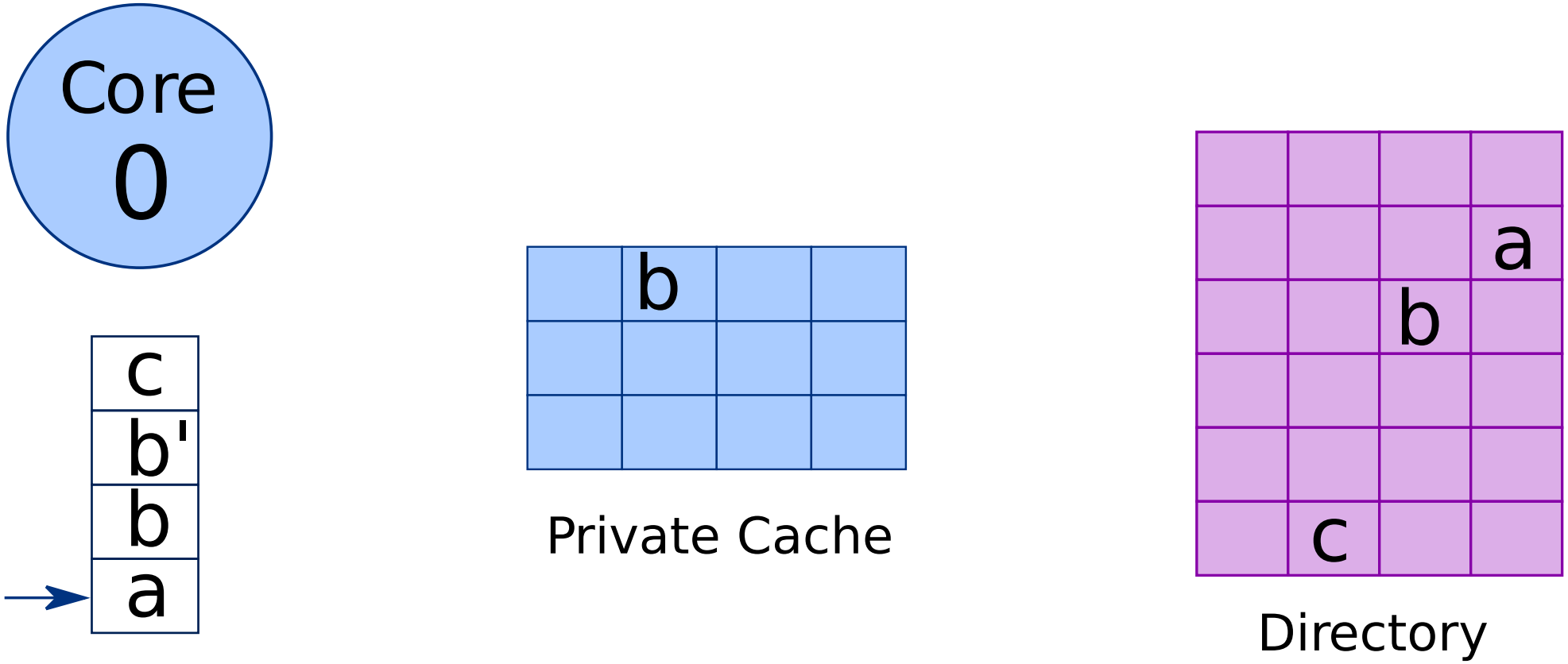
LexConflicts

LexOrder

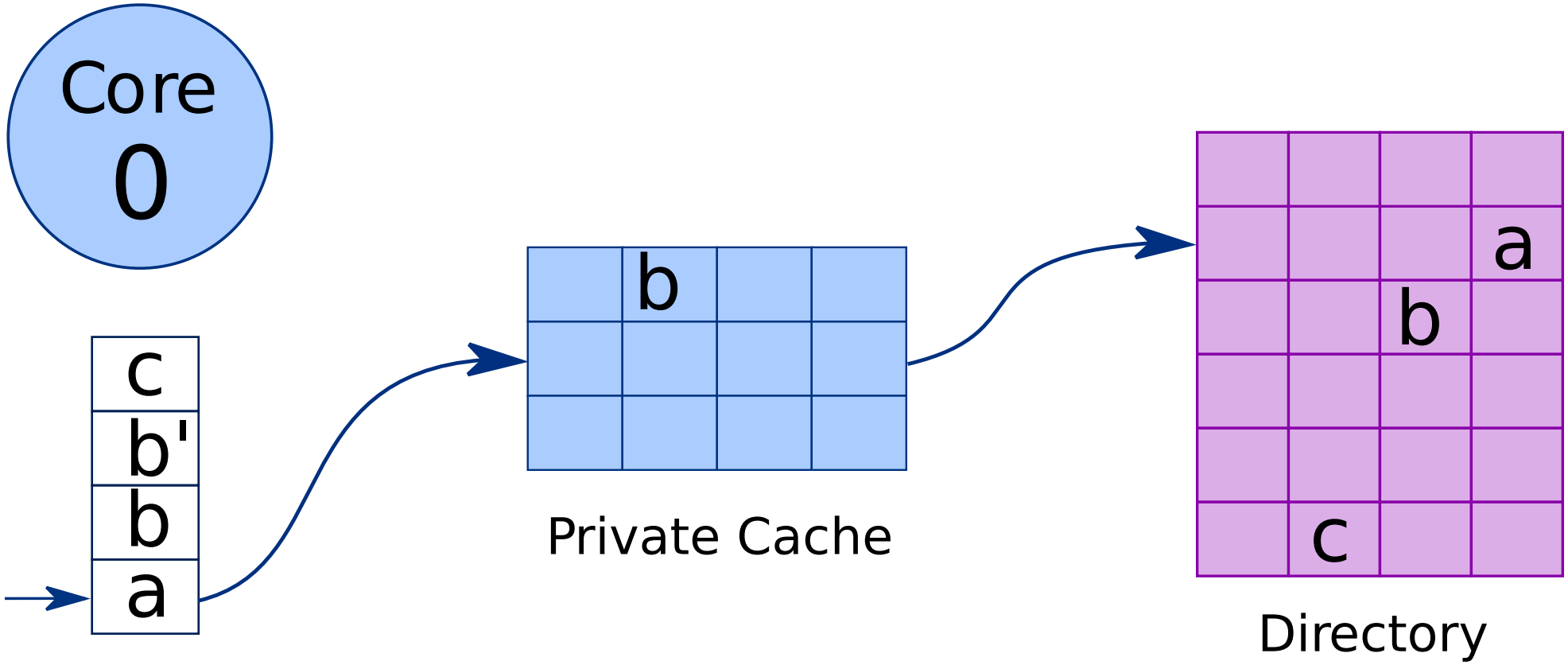



```
LexOrder = CacheLine Address % Cache Sets
```

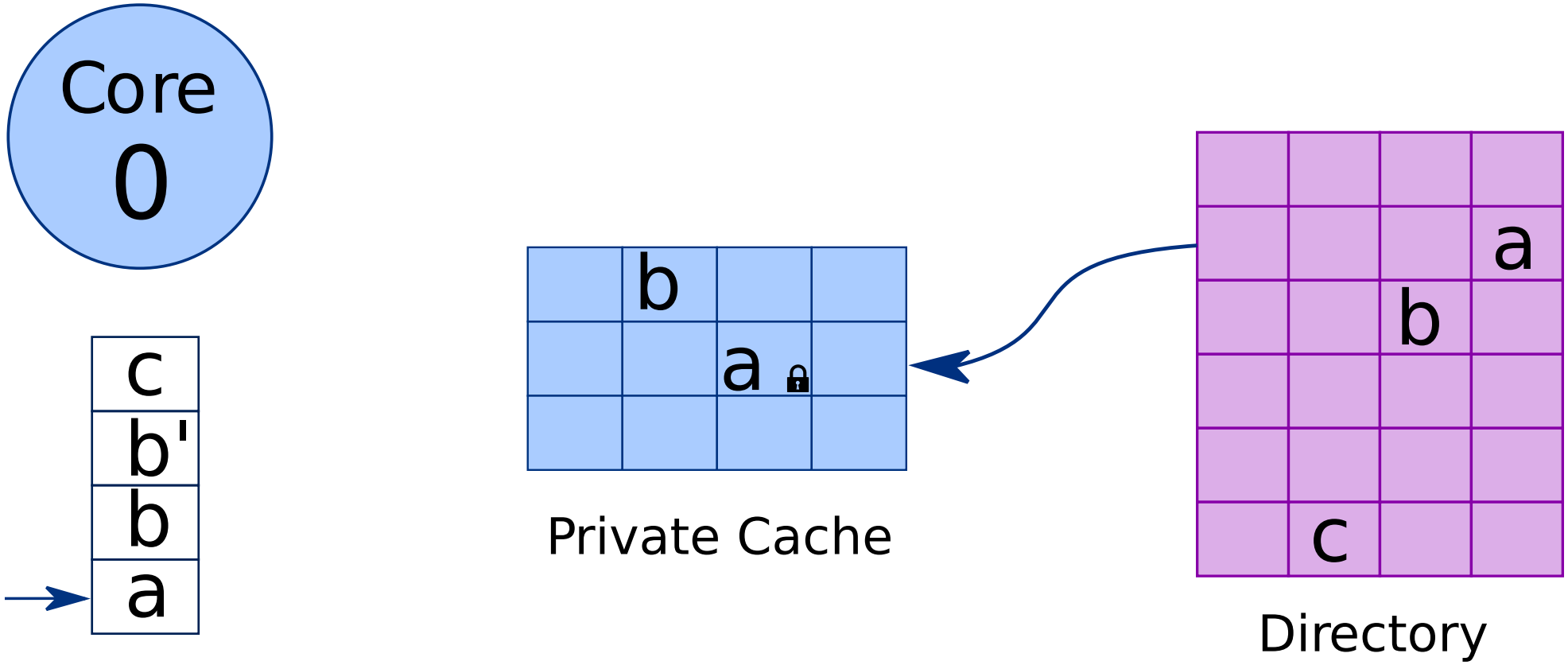
Hardware Multi-Address Atomics - Example



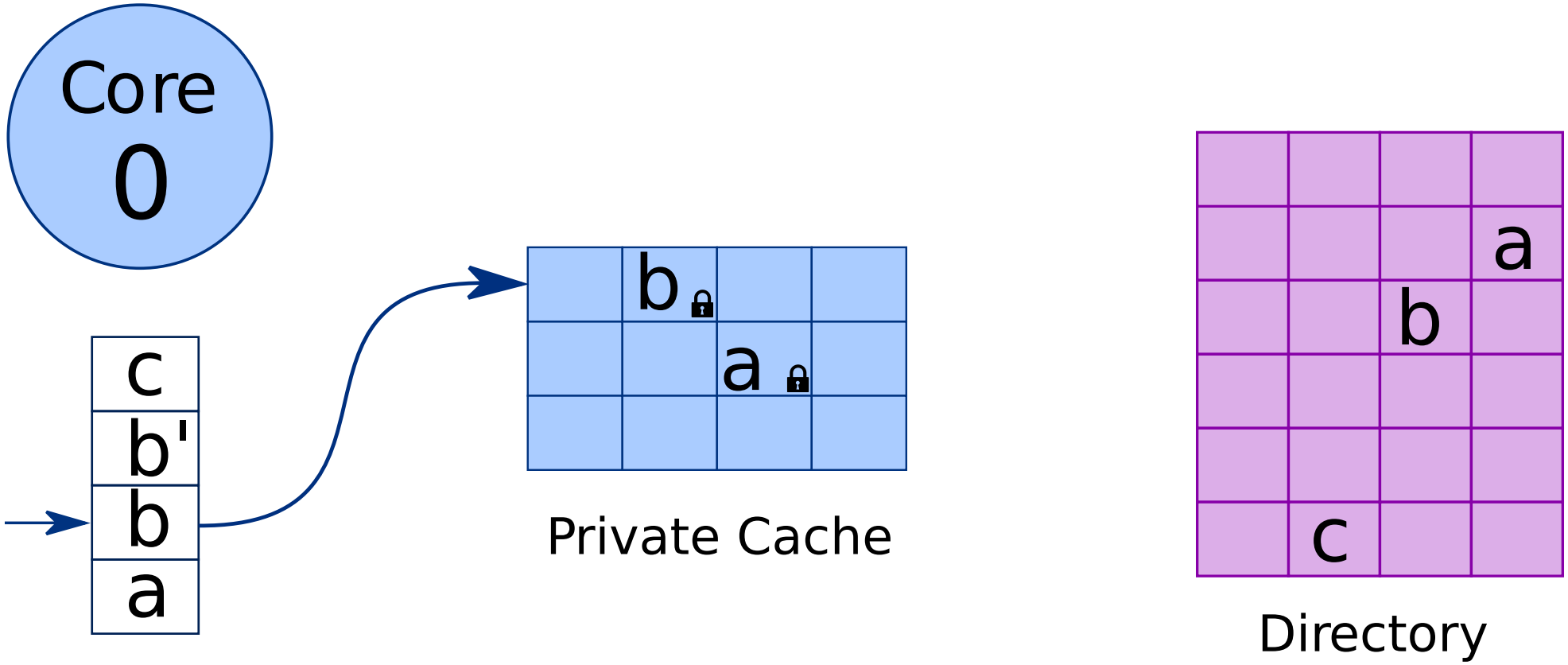
Hardware Multi-Address Atomics - Example



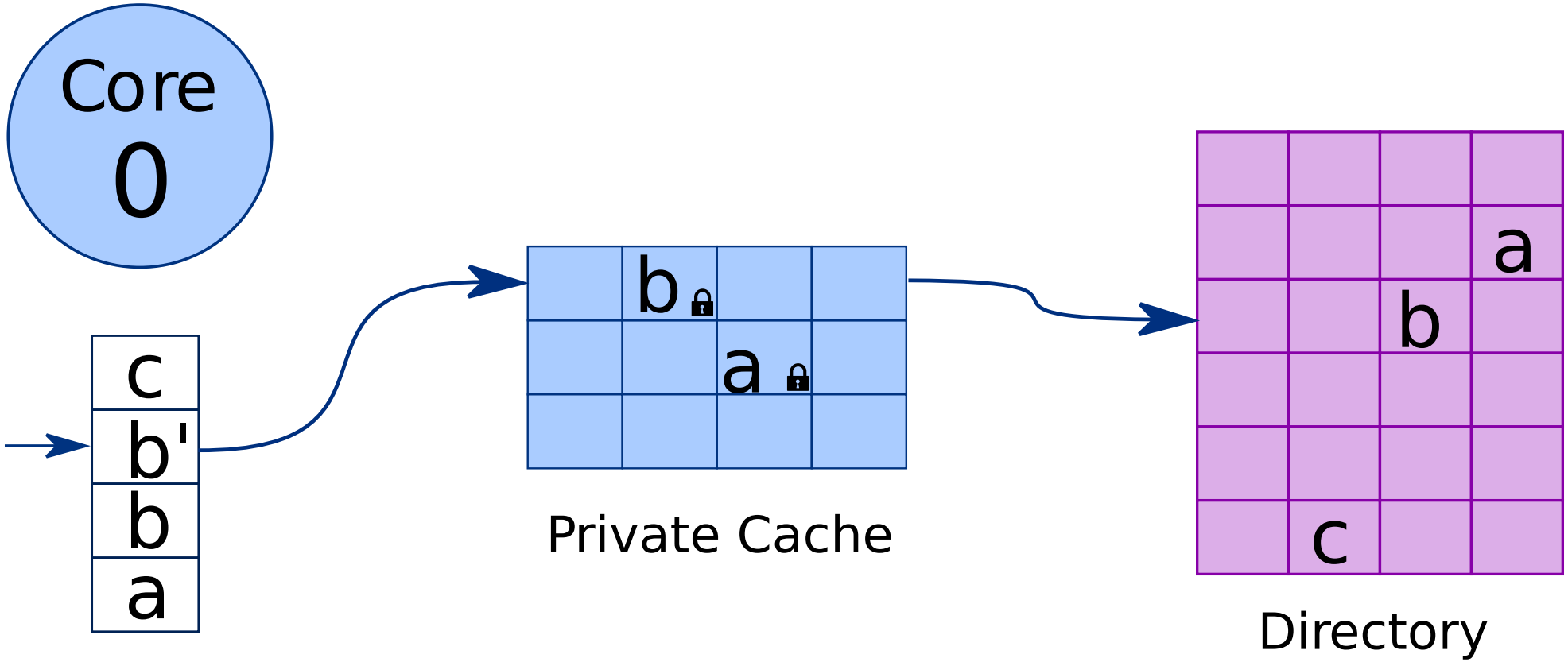
Hardware Multi-Address Atomics - Example



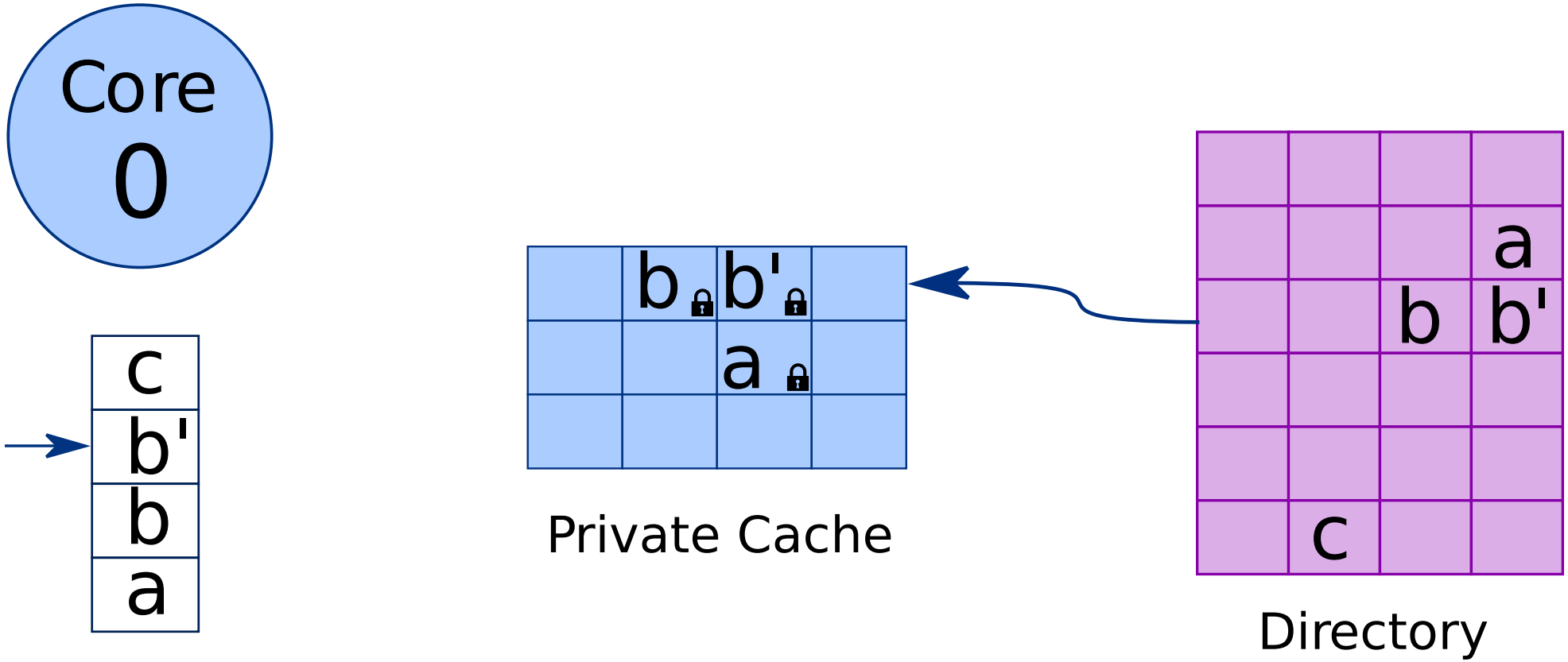
Hardware Multi-Address Atomics - Example



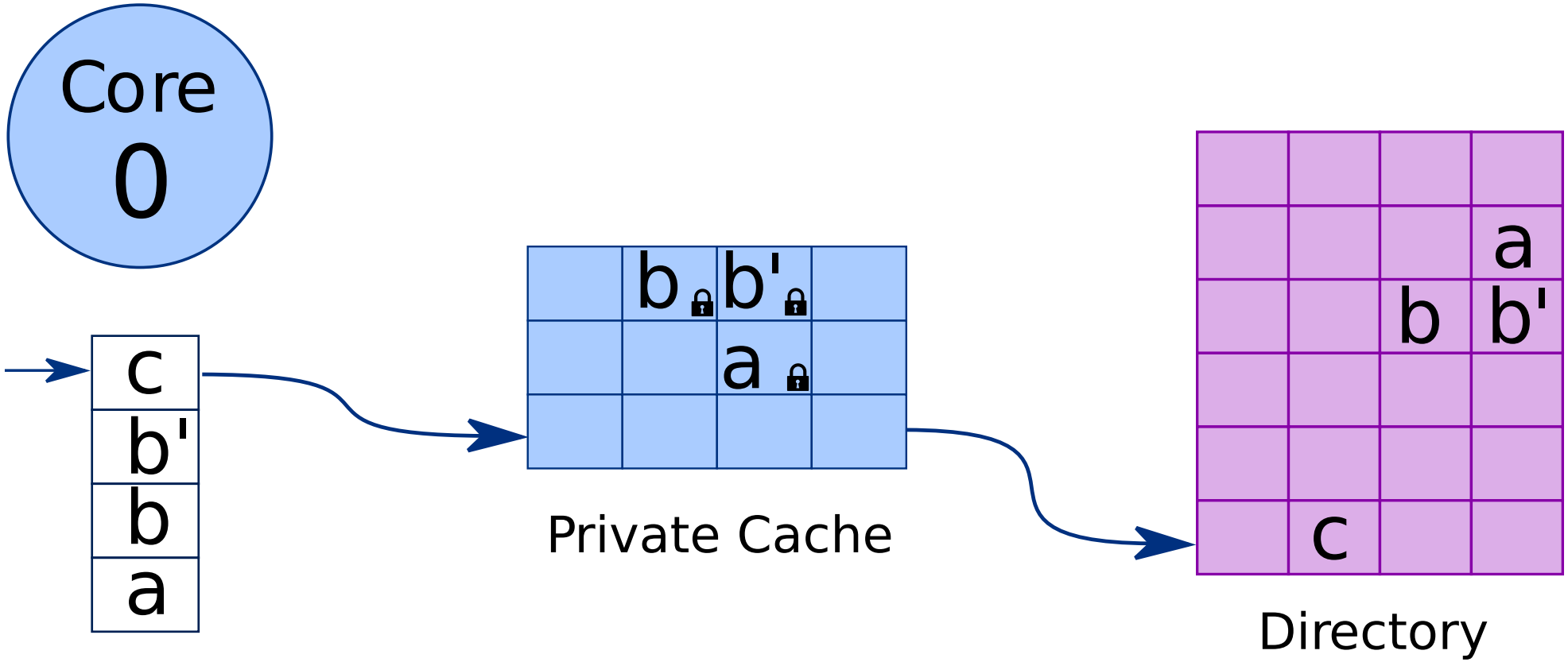
Hardware Multi-Address Atomics - Example



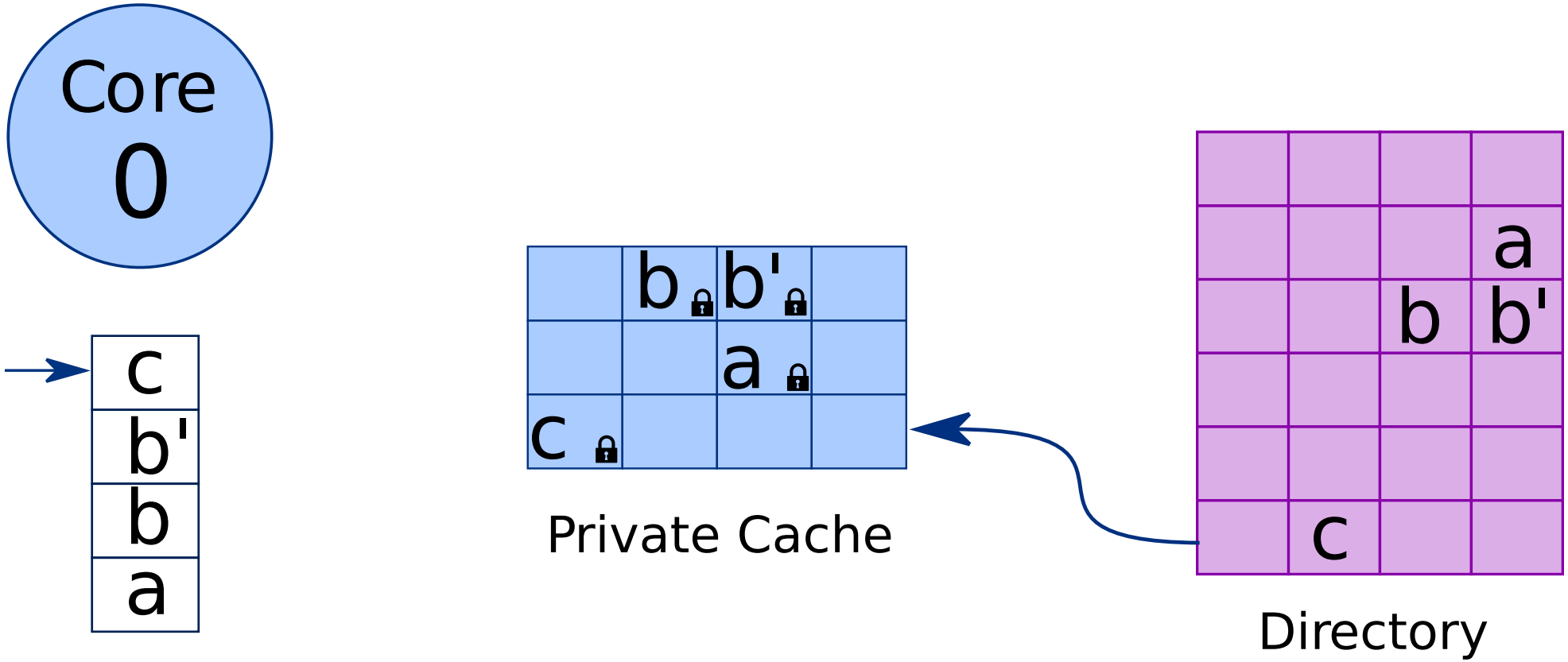
Hardware Multi-Address Atomics - Example



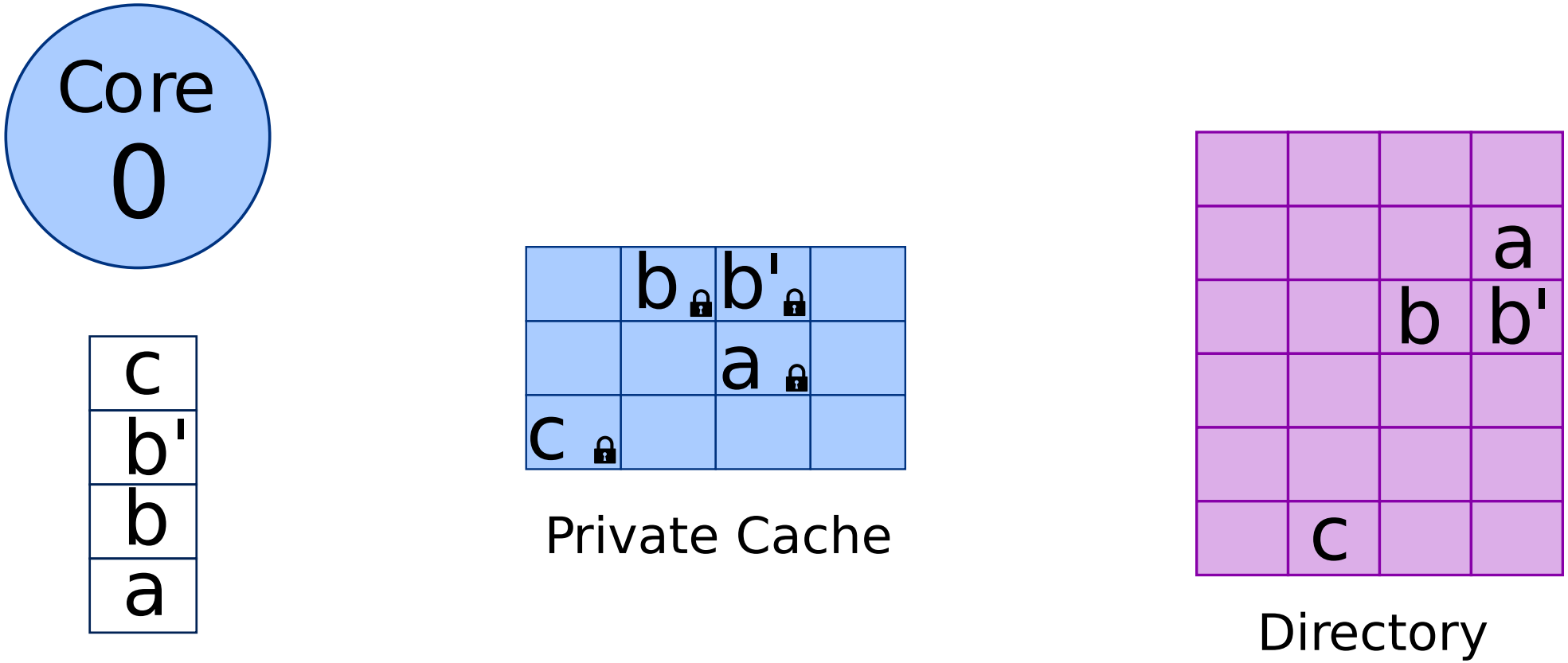
Hardware Multi-Address Atomics - Example



Hardware Multi-Address Atomics - Example



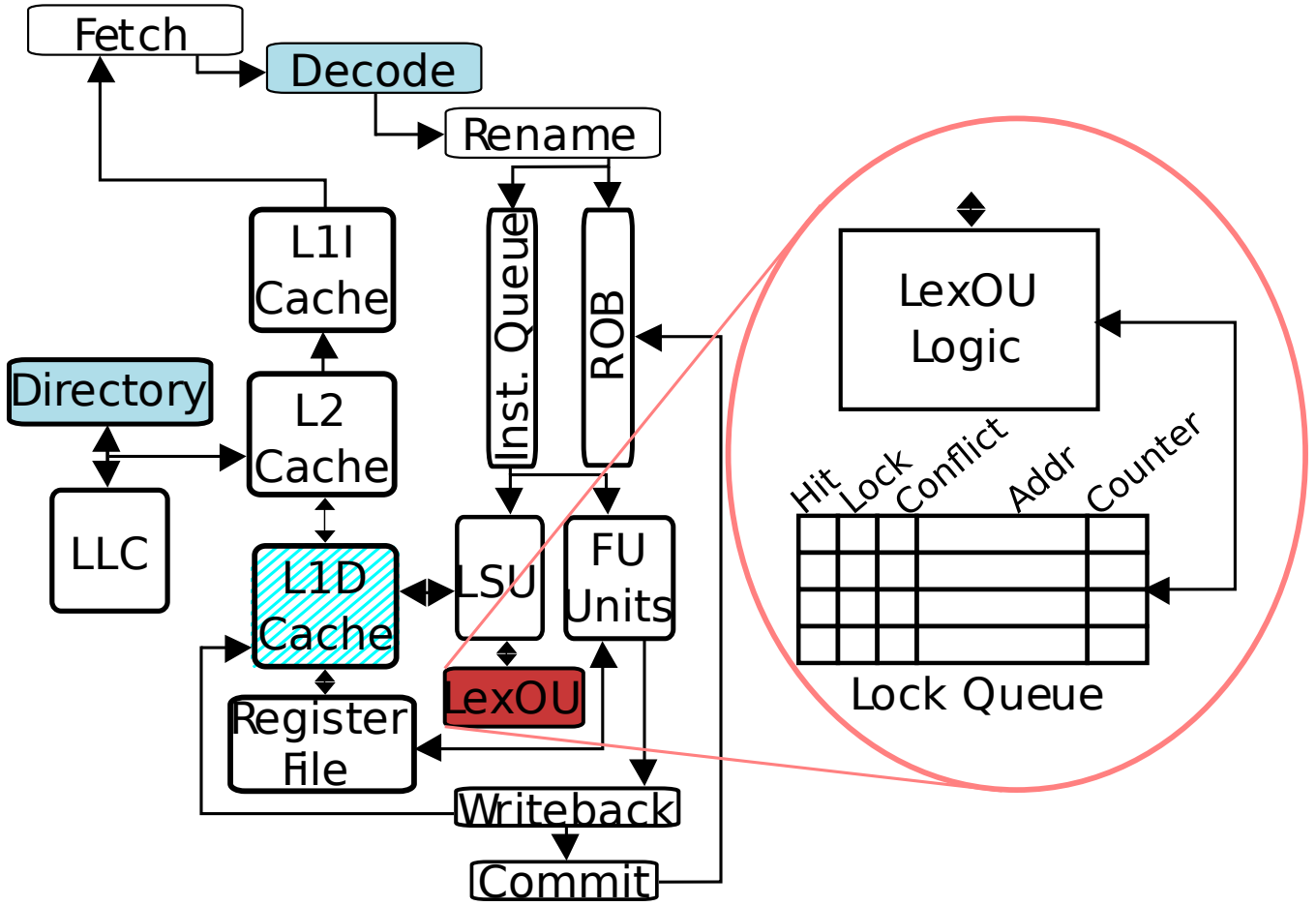
Hardware Multi-Address Atomics - Example



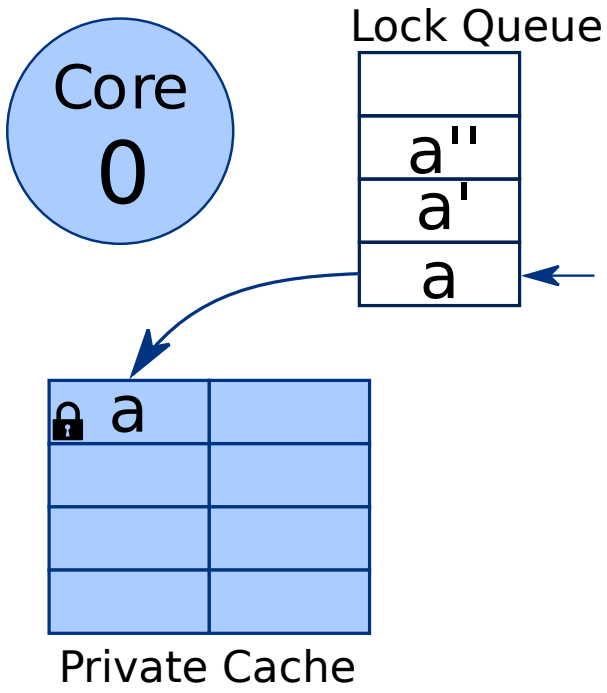
Hardware Multi-Address Atomics

- Minor modifications to:
  - Directory
  - L1 Dcache
  - Decode (to support the new instructions)

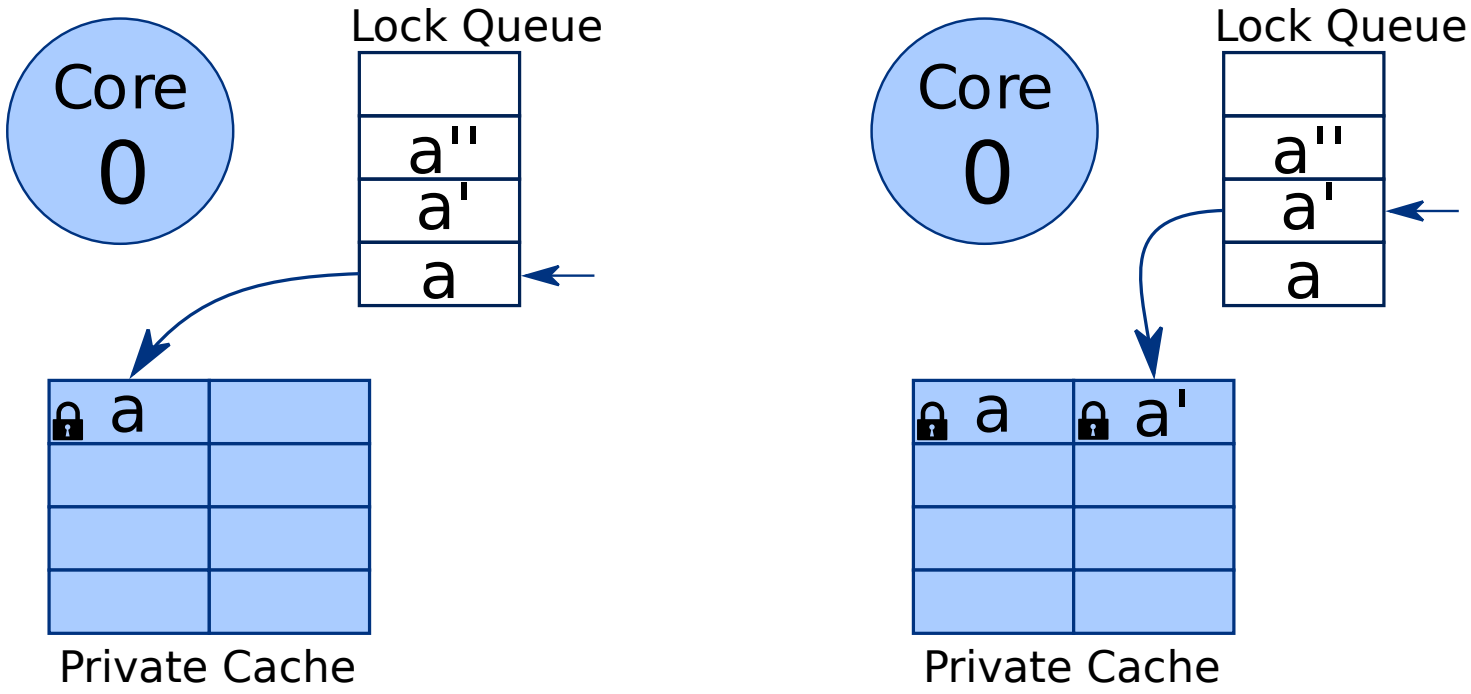
- Added an extra unit attached to the LSU
  - Provides all the logic required for:
    - Tracking
    - Ordering
    - Locking
    - Unlocking



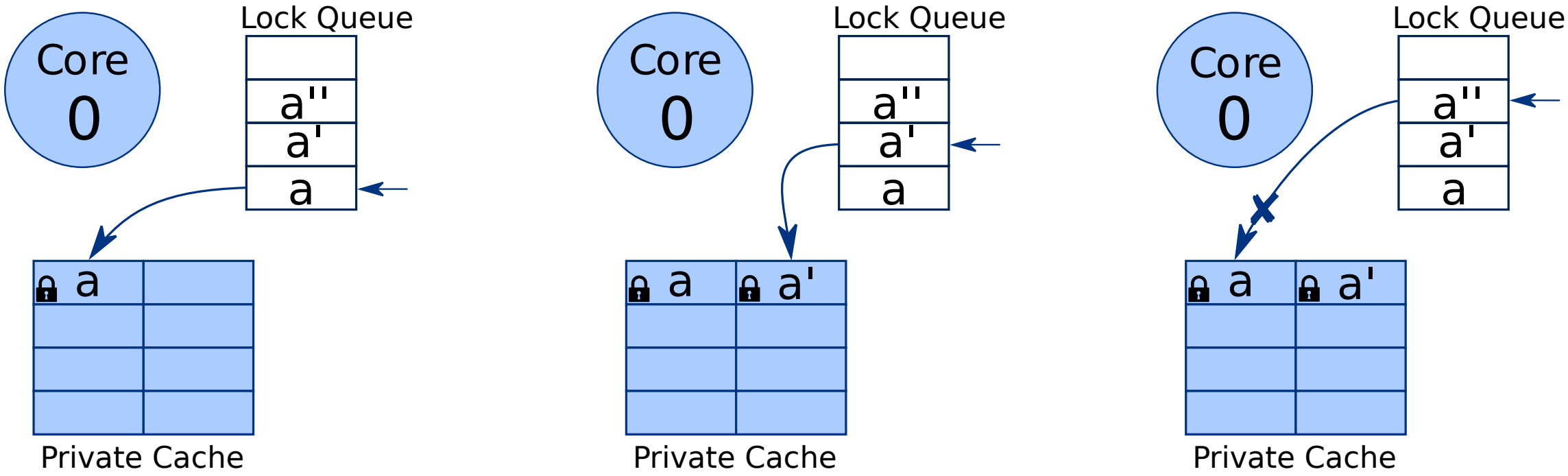
Hardware Multi-Address Atomics – Deadlocks - Private



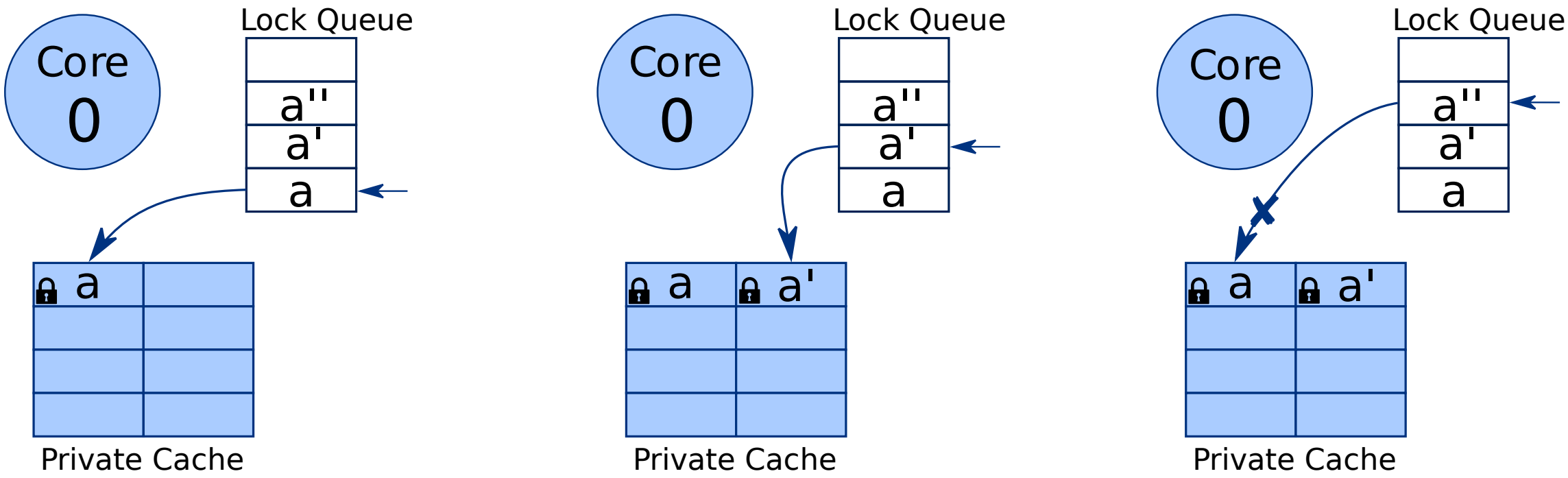
Hardware Multi-Address Atomics – Deadlocks - Private



Hardware Multi-Address Atomics – Deadlocks - Private

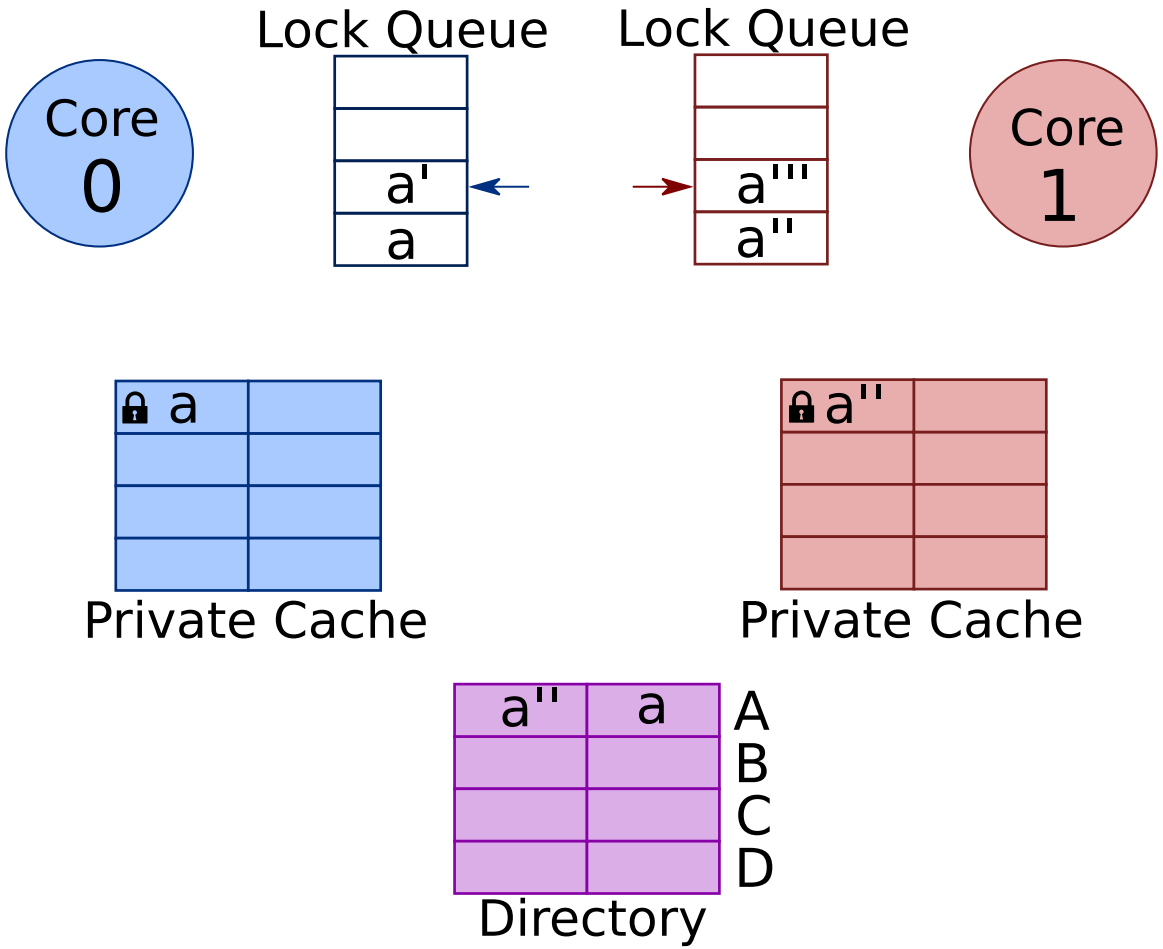


Hardware Multi-Address Atomics – Deadlocks - Private

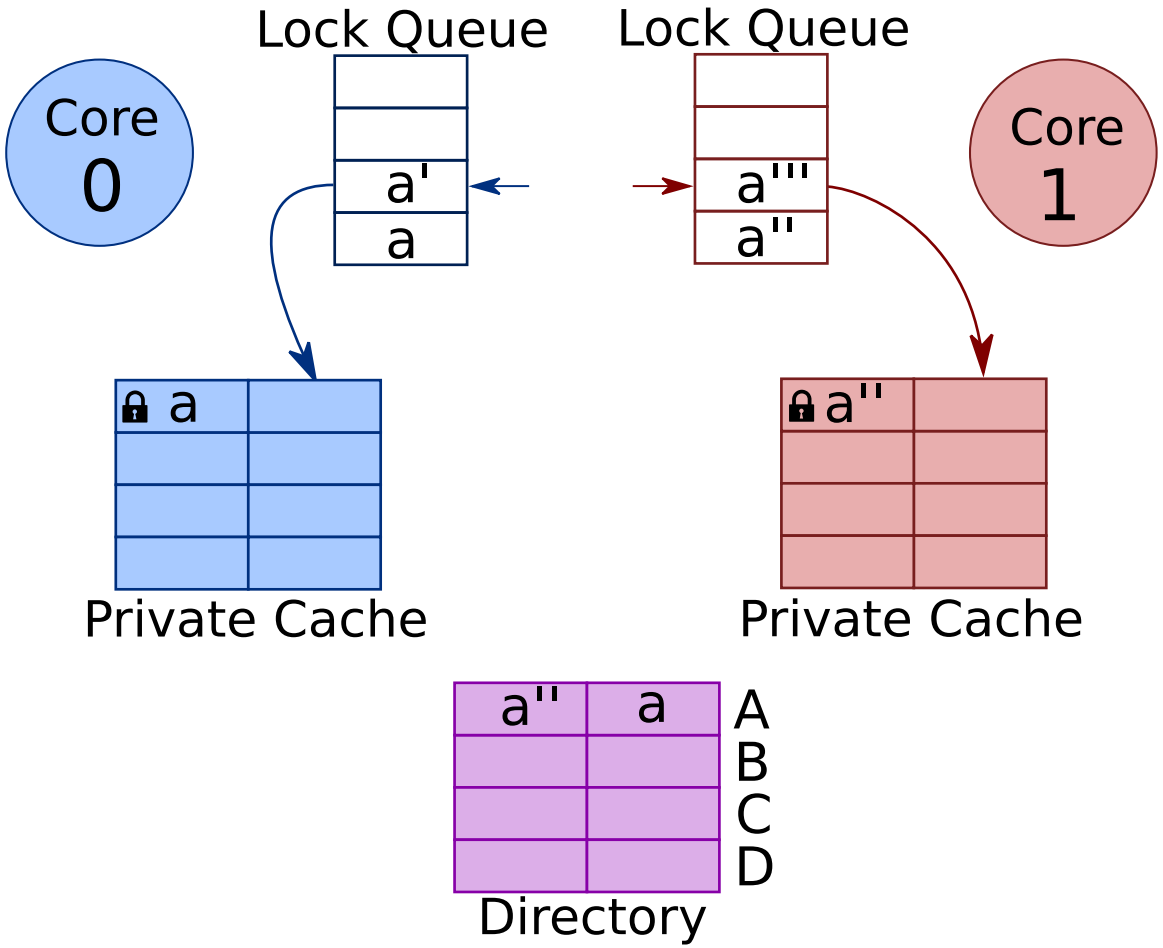


The number of addresses has to be at most the associality of the smallest cache

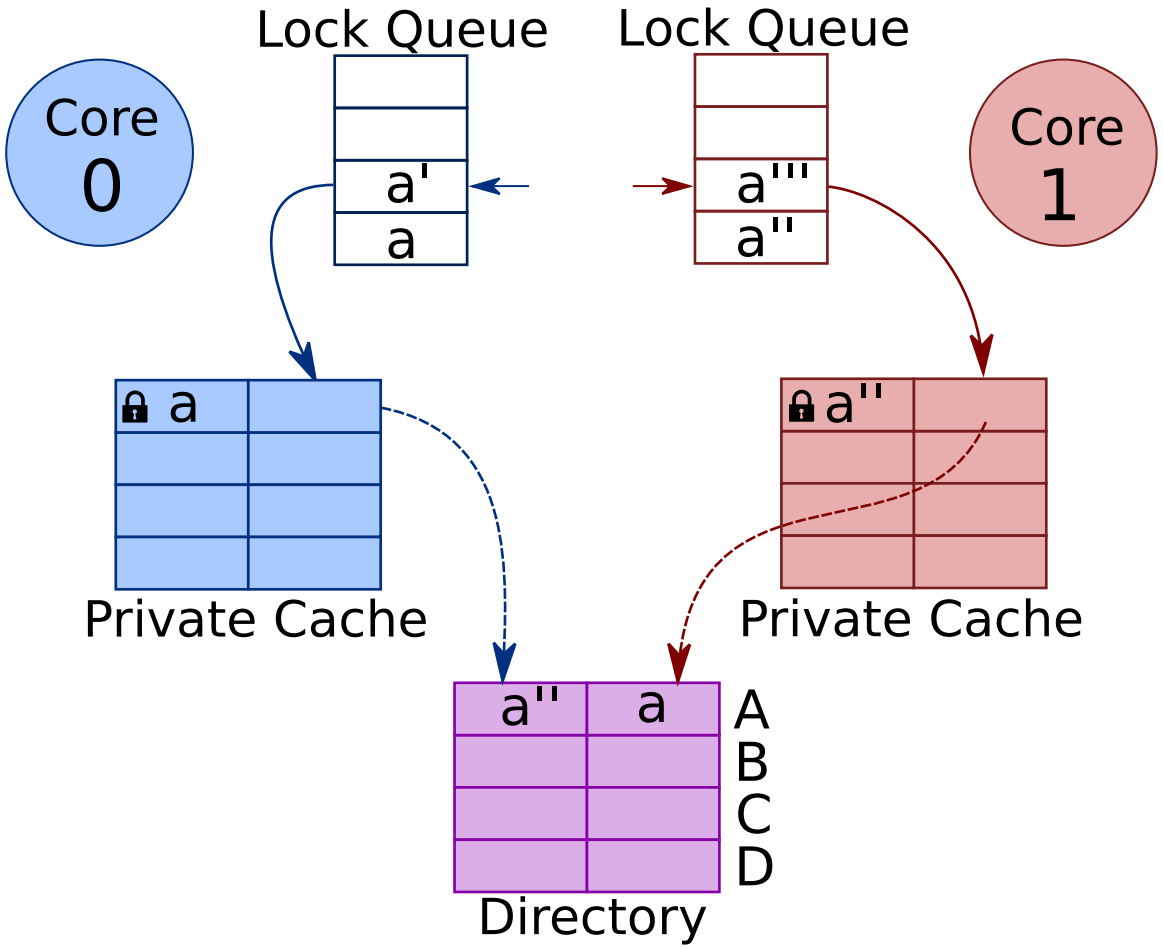
Hardware Multi-Address Atomics – Deadlocks - Shared



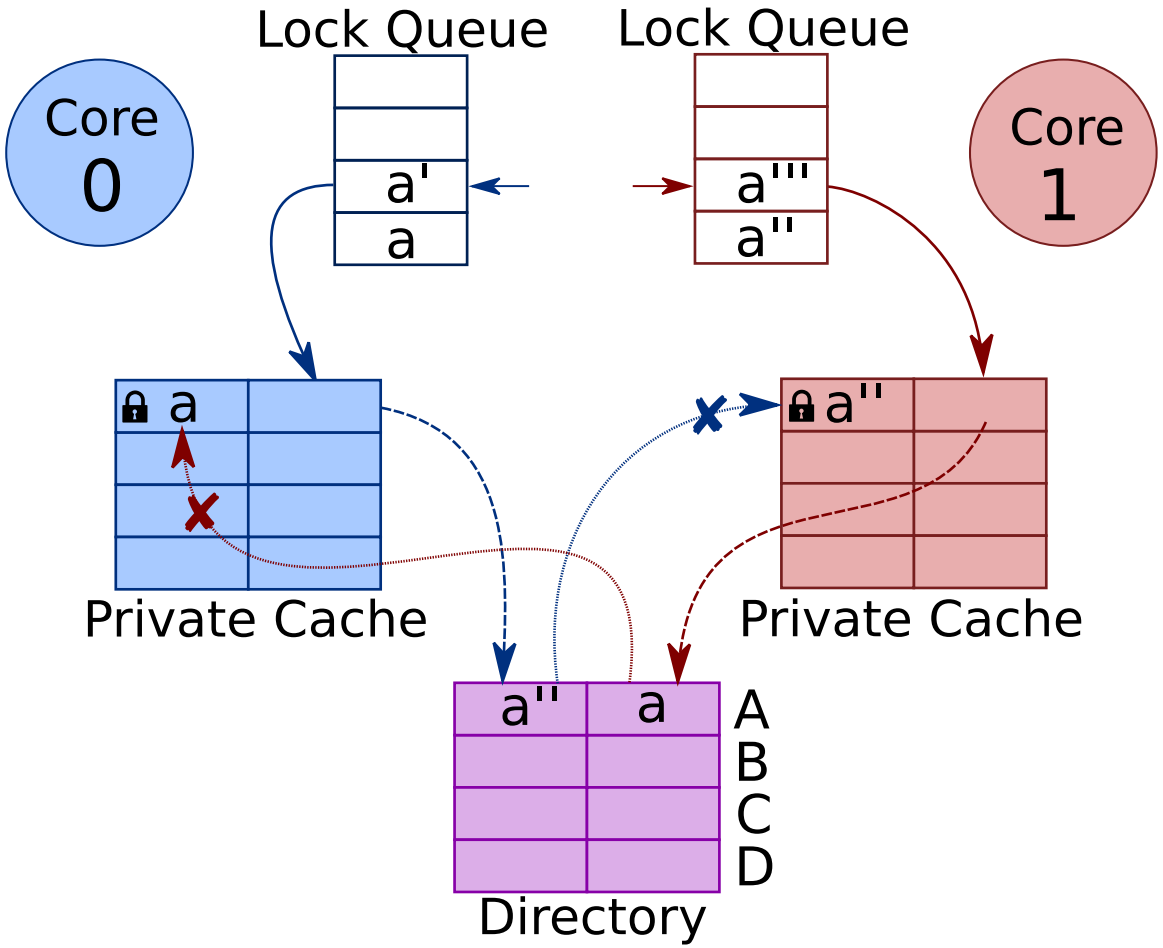
Hardware Multi-Address Atomics – Deadlocks - Shared



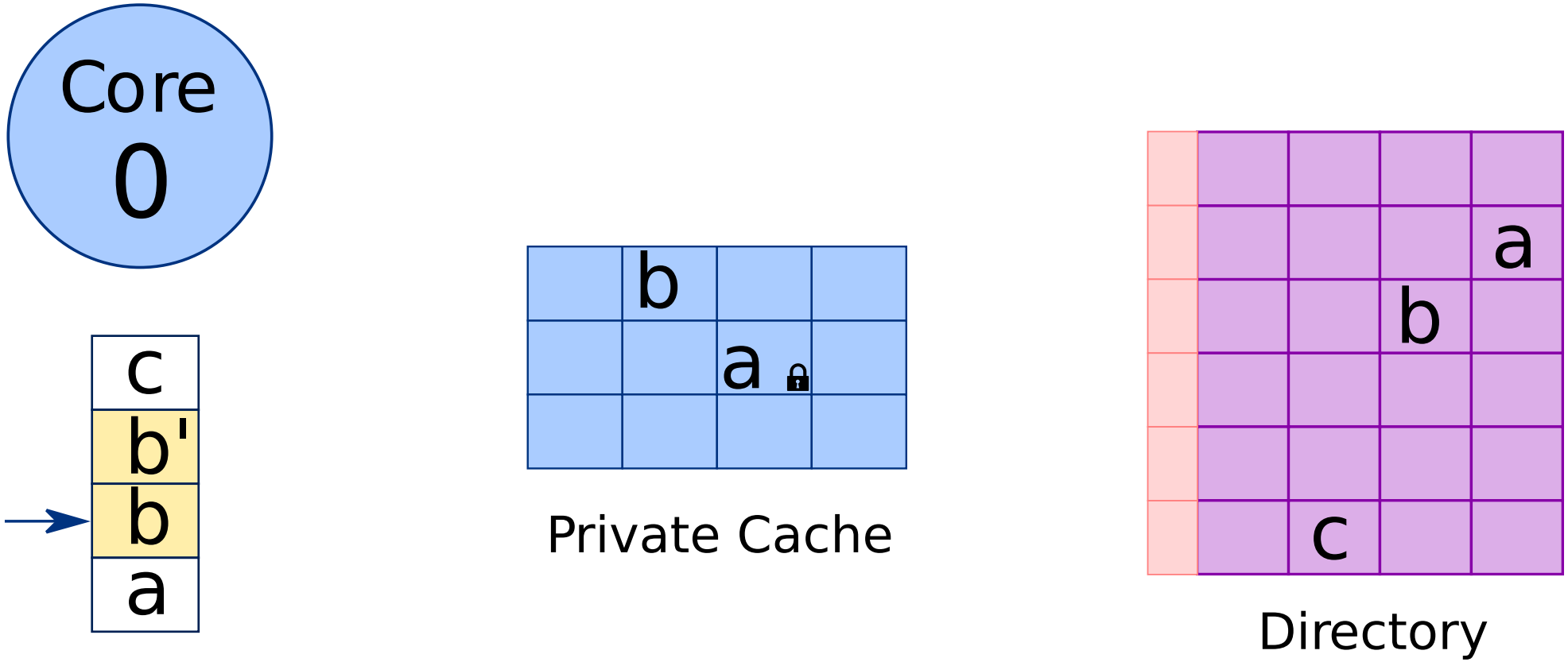
Hardware Multi-Address Atomics – Deadlocks - Shared



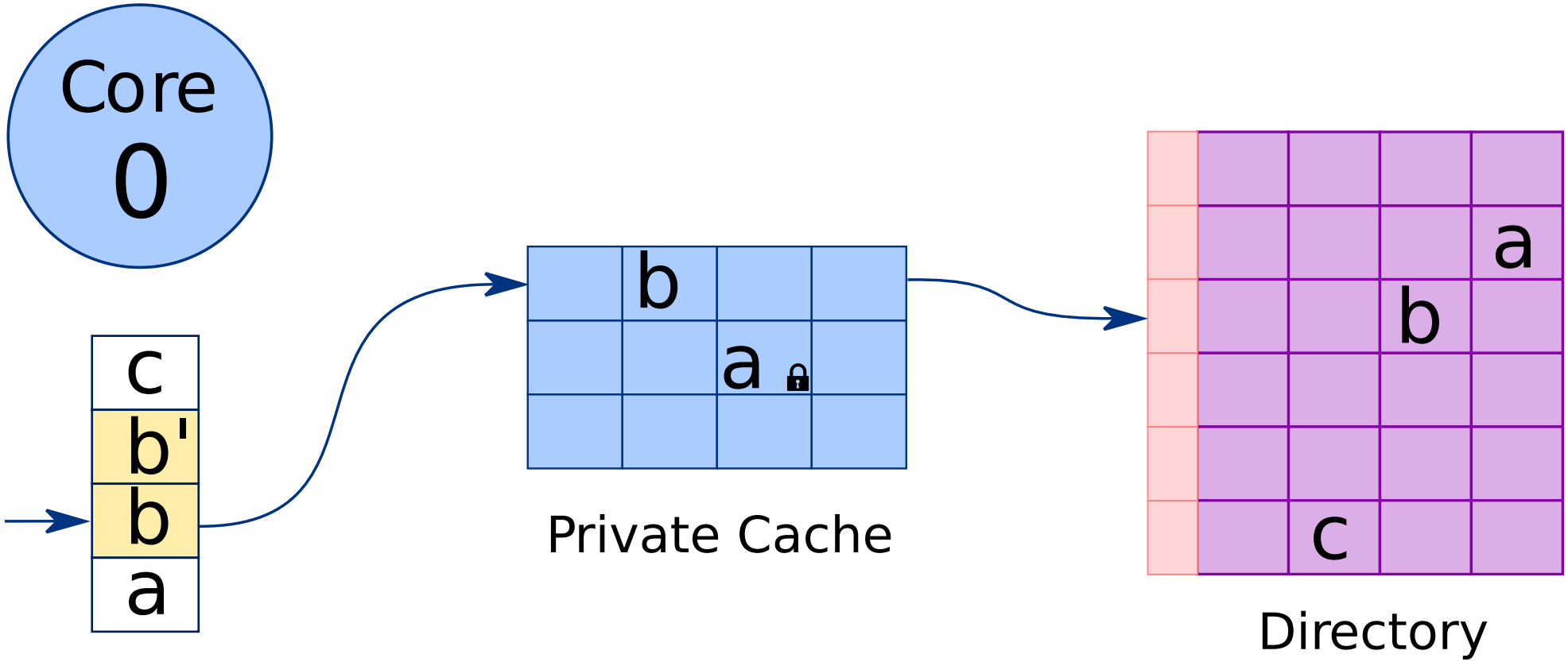
Hardware Multi-Address Atomics – Deadlocks - Shared



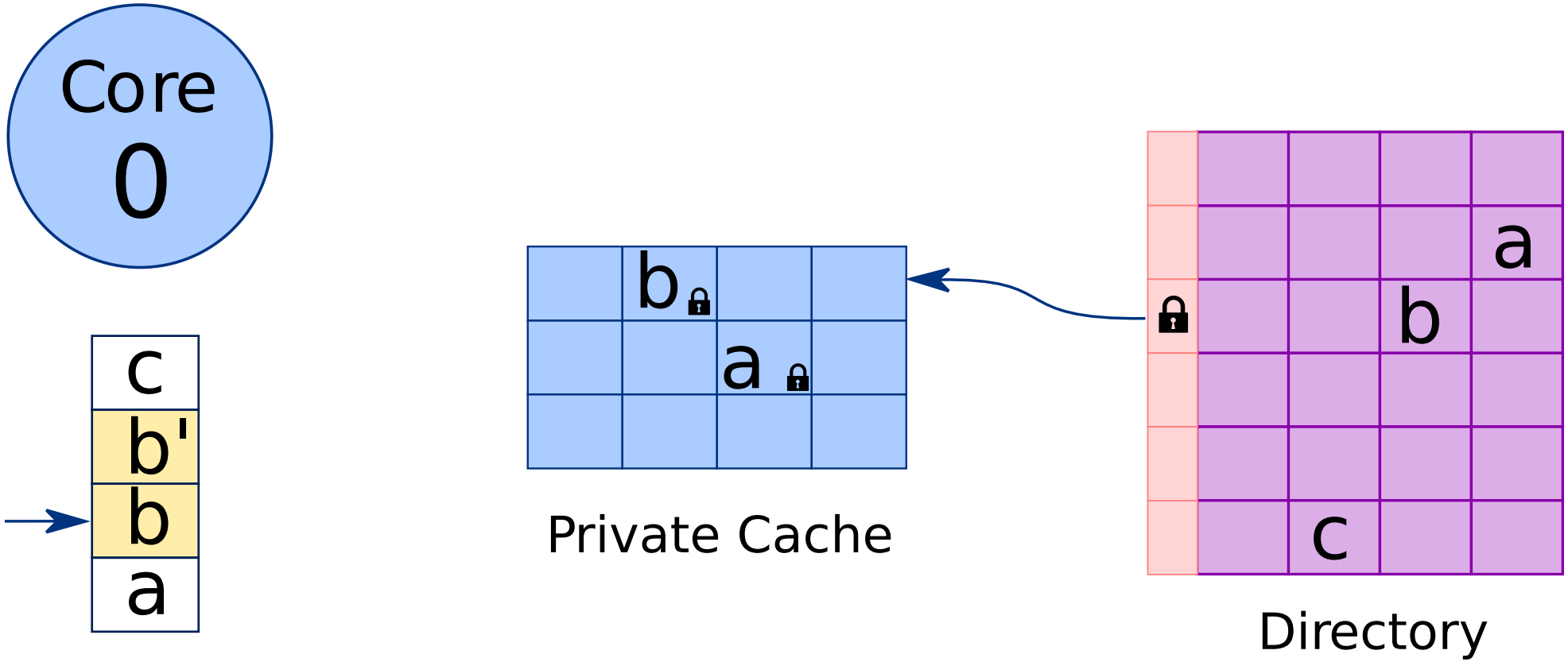
Hardware Multi-Address Atomics – Example (conflict)



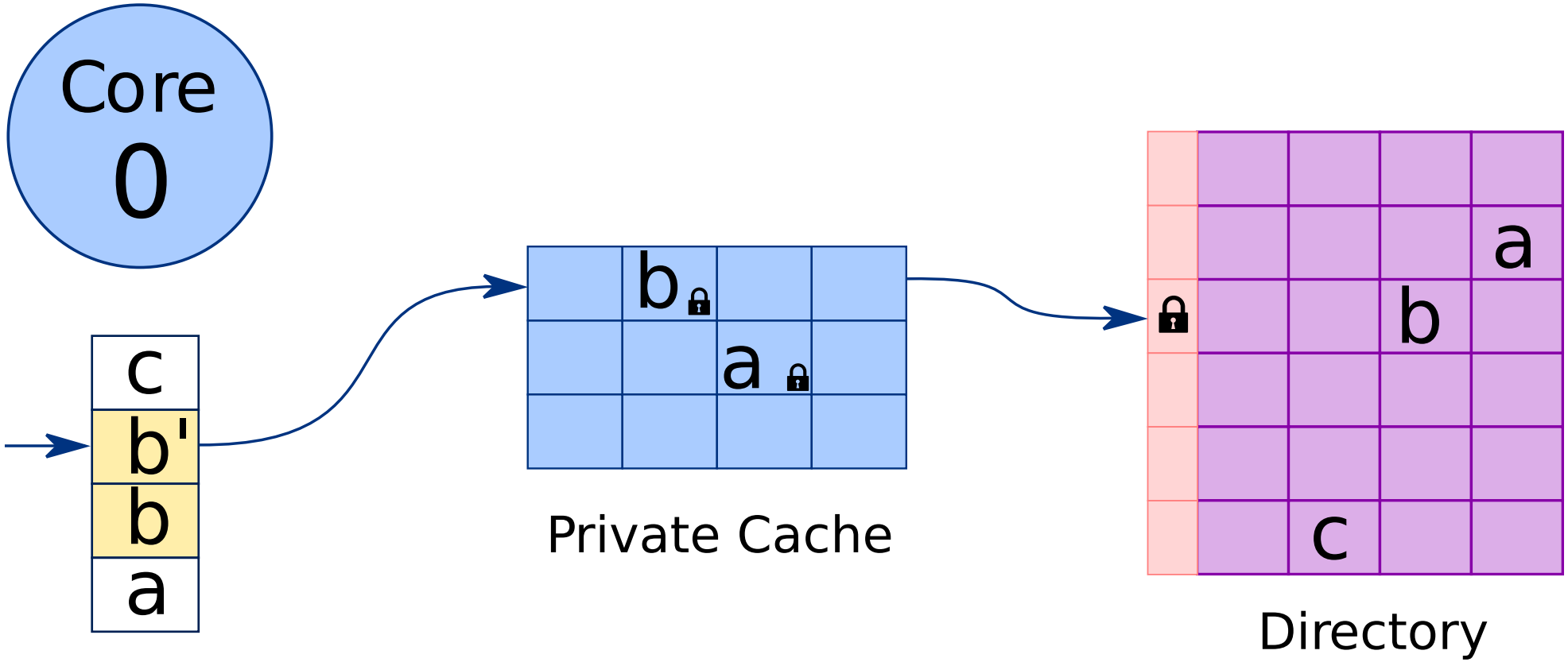
Hardware Multi-Address Atomics – Example (conflict)



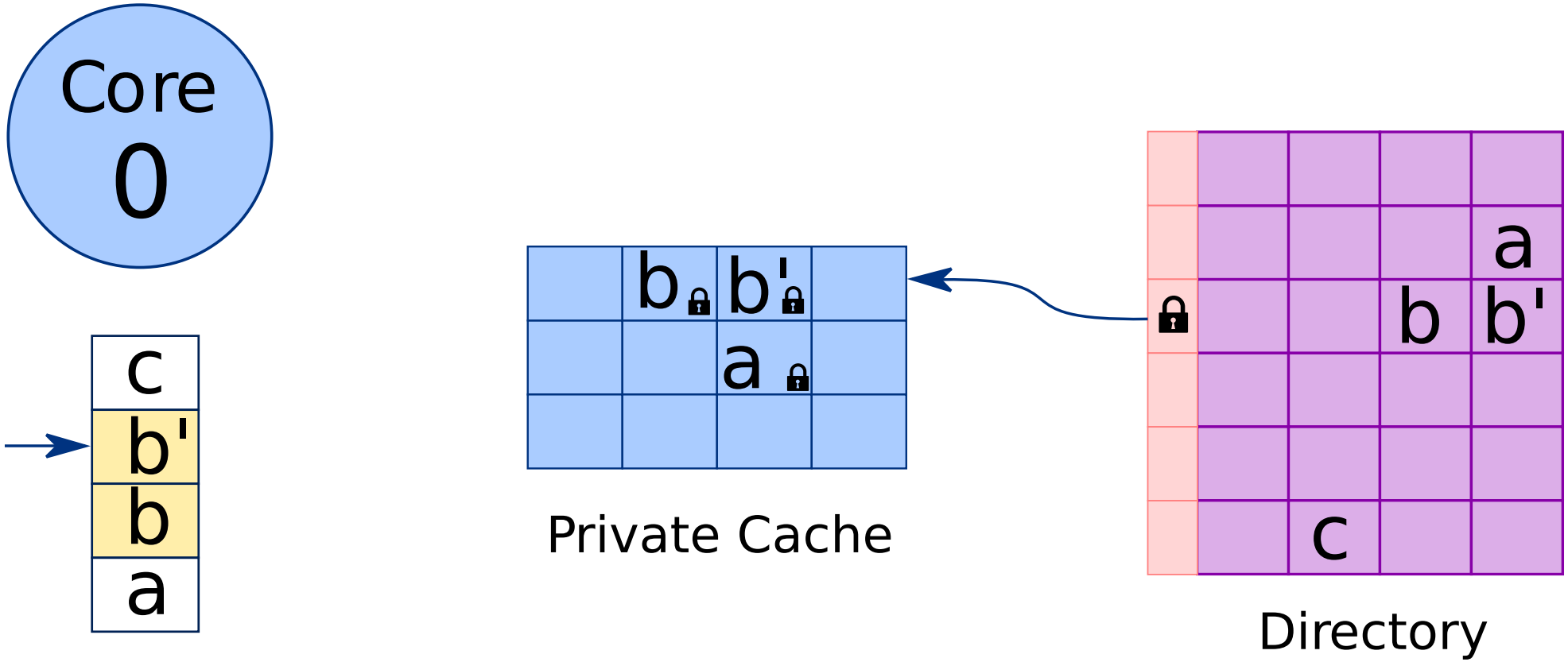
Hardware Multi-Address Atomics – Example (conflict)



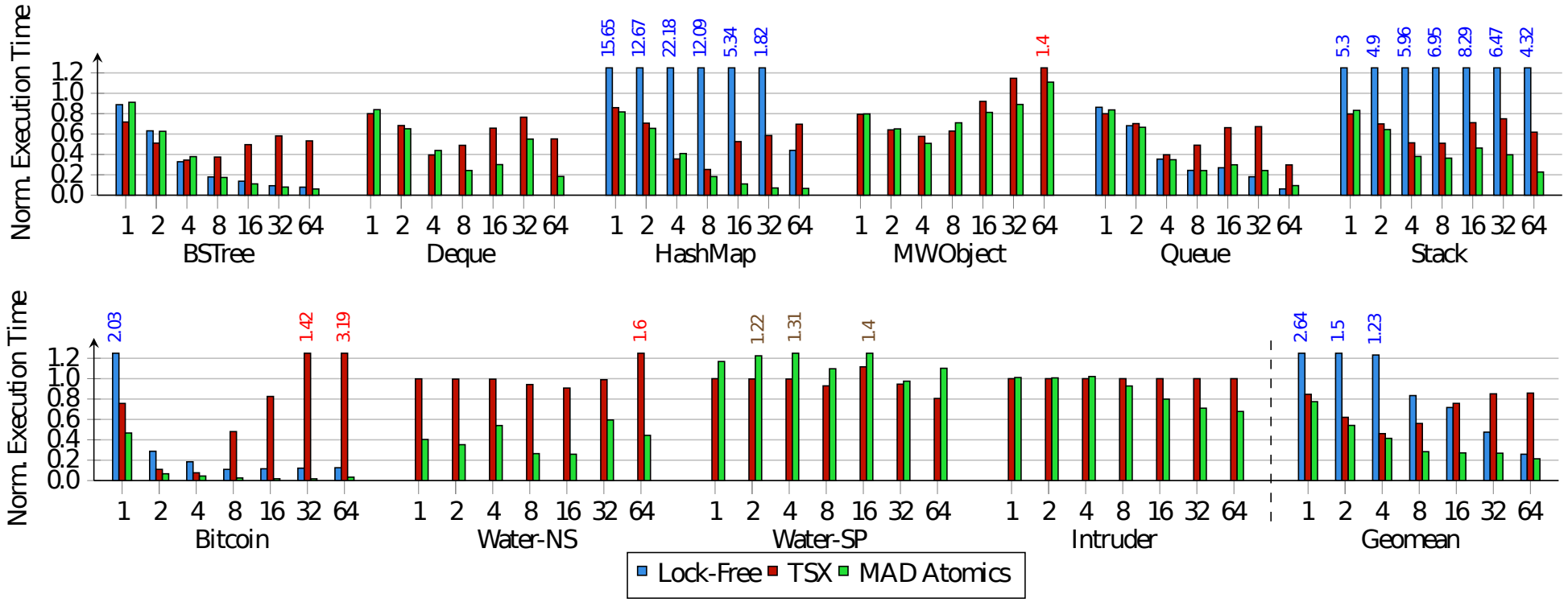
Hardware Multi-Address Atomics – Example (conflict)



Hardware Multi-Address Atomics – Example (conflict)



Hardware Multi-Address Atomics



Multi-Address Atomics reduces up to almost 80% the execution time when compared against the basic implementation done with mutex locks (normalized)

Published at:
- ASPLOS 2024

CLEAR Bounding TM to a single retry

I got it, 4 is not enough, and it is hard to write...
How about n-Addresses but with 1 retry?

cleAR: Bounding TM to a single retry - Objective

- **GOAL**: Enable concurrent and non-speculative execution of ARs, without modifying the software/ISA

CLEAR: Bounding TM to a single retry - Objective

- **GOAL:** Enable concurrent and non-speculative execution of ARs, without modifying the software/ISA

- **KEY QUESTION:** Can the hardware learn the memory footprint of ARs at runtime?

CLEAR: Bounding TM to a single retry - Objective

- **GOAL:** Enable concurrent and non-speculative execution of ARs, without modifying the software/ISA

- **KEY QUESTION:** Can the hardware learn the memory footprint of ARs at runtime?

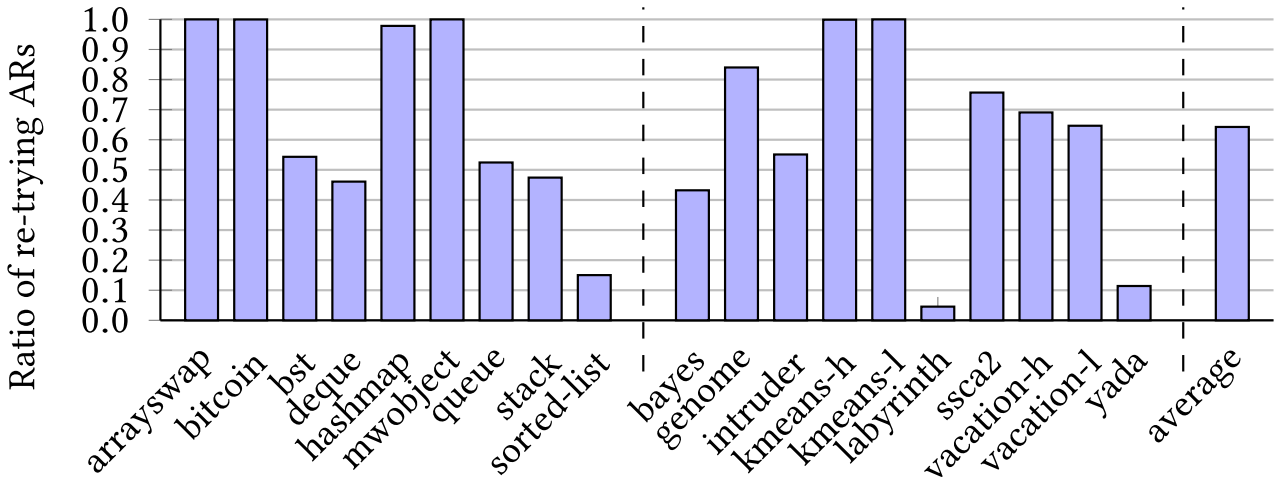
- If so, if a **retry** is needed,
  - **cachelines** can be **locked** following a **non-deadlocking global order**,
  - and the AR can be **executed non-speculatively** (no more retries needed)

cleAR: Bounding TM to a single retry - Memory

- To execute an Atomic Region non-speculatively is to know the memory footprint on advance before executing
- Can the hardware learn the memory footprint?

cleAR: Bounding TM to a single retry - Memory

- To execute an Atomic Region non-speculatively is to know the memory footprint on advance before executing
- Can the hardware learn the memory footprint?
- Most ARs do not change their memory footprint on retries



cleAR: Bounding TM to a single retry - Usage

- What about the previous critical section we have been discussing?

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

CLEAR: Bounding TM to a single retry - Usage

- What about the previous critical section we have been discussing?

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 SECTION_END;
```

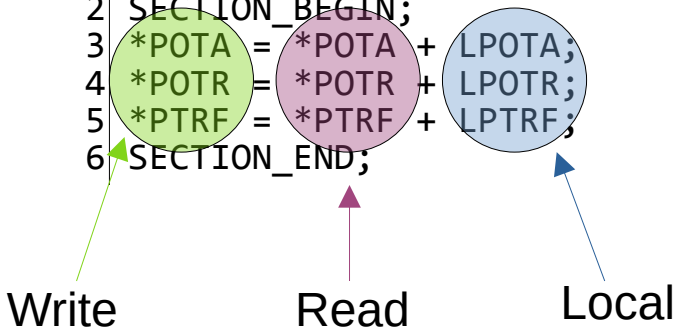
CLEAR: Bounding TM to a single retry - Usage

- What about the previous critical section we have been discussing?

- In the case of a config, the hardware has to take into account 3 addresses for read-write (also depending on the compiler another 3 for read-only)

```
1 /* Lock */
2 LOCK(gl->PotengSumLock);
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 UNLOCK(gl->PotengSumLock);
```

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 SECTION_END;
```



CLEAR: Bounding TM to a single retry – Types of ARs

Immutable

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 SECTION_END;
```

Memory footprint does **never change** between retries.

CLEAR: Bounding TM to a single retry – Types of ARs

Immutable

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 *POTA = *POTA + LPOTA;
4 *POTR = *POTR + LPOTR;
5 *PTRF = *PTRF + LPTRF;
6 SECTION_END;
```

Memory footprint does **never change** between retries.

Mutable

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 data = *ptr;
4 data->POTA = data->POTA + LPOTA;
5 data->POTR = data->POTR + LPOTR;
6 data->PTRF = data->PTRF + LPTRF;
7 SECTION_END;
```

Because of the indirection,

```
1 /* CLEAR? */
2 SECTION_BEGIN;
3 if (*wannabe > 5) {
4     *POTA = *POTA + LPOTA;
5     *POTR = *POTR + LPOTR;
6     *PTRF = *PTRF + LPTRF;
7 }
8 SECTION_END;
```

Because of the branch (depending on a memory location),

memory footprint **may change** between retries

CLEAR: Bounding TM to a single retry – How?

- CLEAR executes the AR in **two** different, but collaborative **steps**:

CLEAR: Bounding TM to a single retry – How?

- CLEAR executes the AR in **two** different, but collaborative **steps**:

1. Discovery
    - An **speculative attempt** to execute the AR
    - **During** the attempt, the processors **learns** the **structure** of the AR
    - If the execution is **successful** (no conflicts), practically **no overhead** over baseline

CLEAR: Bounding TM to a single retry – How?

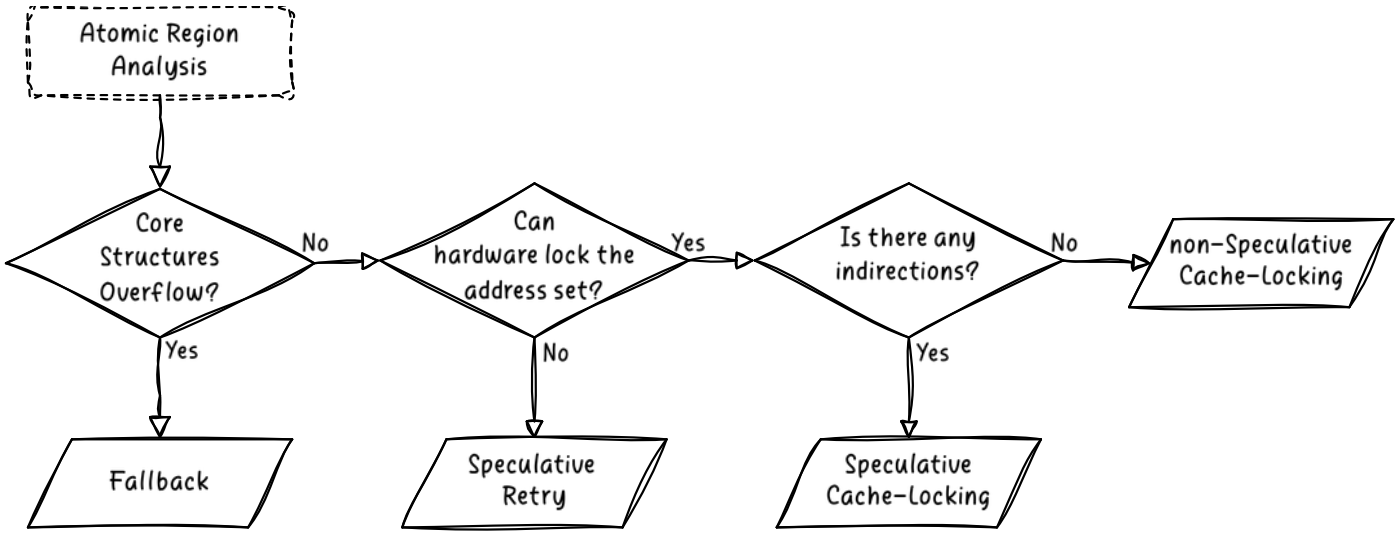
- CLEAR executes the AR in **two** different, but collaborative **steps**:

1. Discovery
   - An **speculative attempt** to execute the AR
   - **During** the attempt, the processors **learns** the **structure** of the AR
   - If the execution is **successful** (no conflicts), practically **no overhead** over baseline

2. Re-Execution
   - **Only** executed if a **confict** was found during discovery
   - **If possible**, the execution can be made **non-speculatively**

cleAR: Bounding TM to a single retry - Discovery

- Aborts are delayed until the end of the AR
  - Because we want to processor to learn the full AR
- Learns the memory accesses and the immutability of the AR
  - This is done by tracking branches and indirections propagated by registers
- When reaching the end of the AR:
  - If no pending abort -> complete!
  - If an abort is pending -> decide!

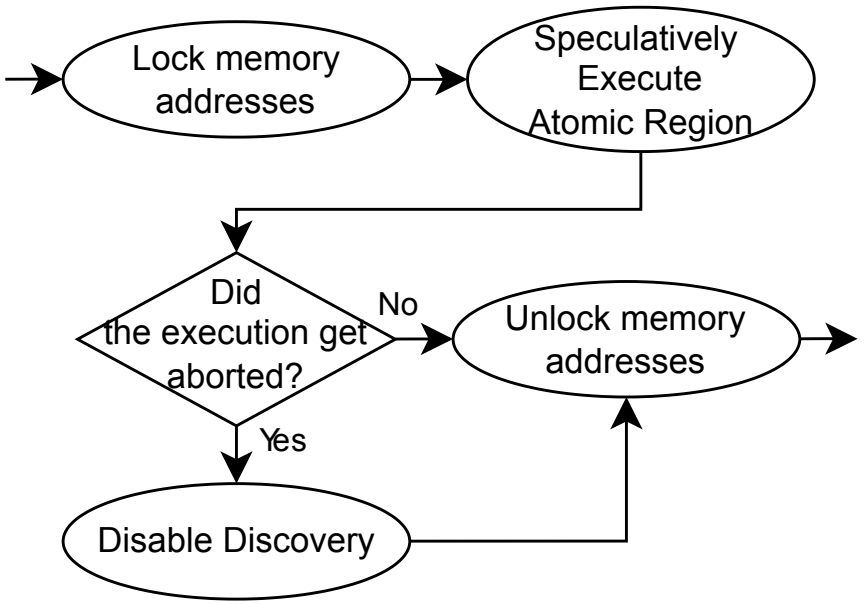


cleAR: Bounding TM to a single retry - non-speculative

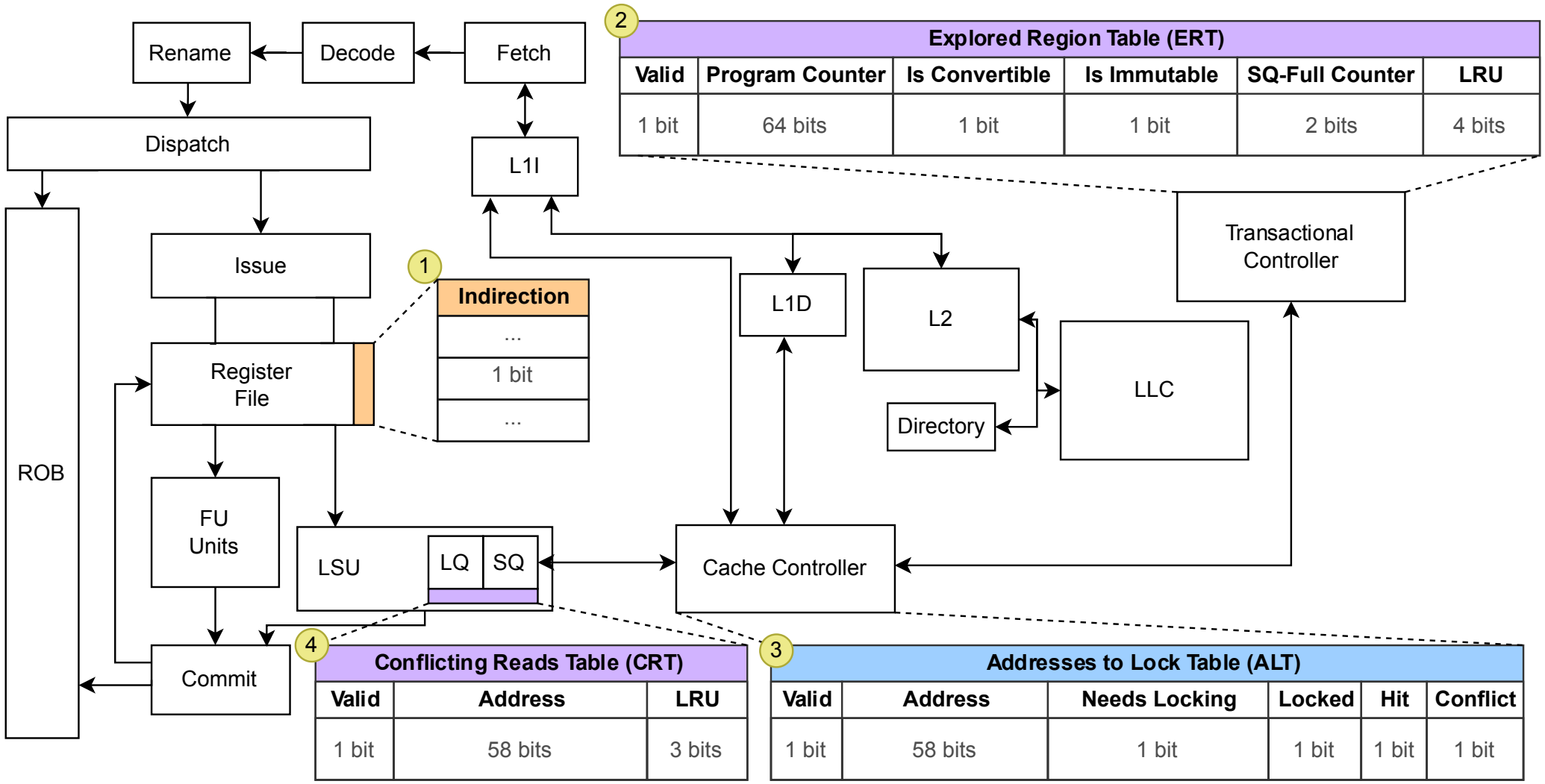
- Execution **without any speculation support** (no more retries are guaranteed)
- Addresses are locked in cache, following a non-deadlocking manner
- When the execution finished, all the locked addresses are unlocked

cleAR: Bounding TM to a single retry - speculative

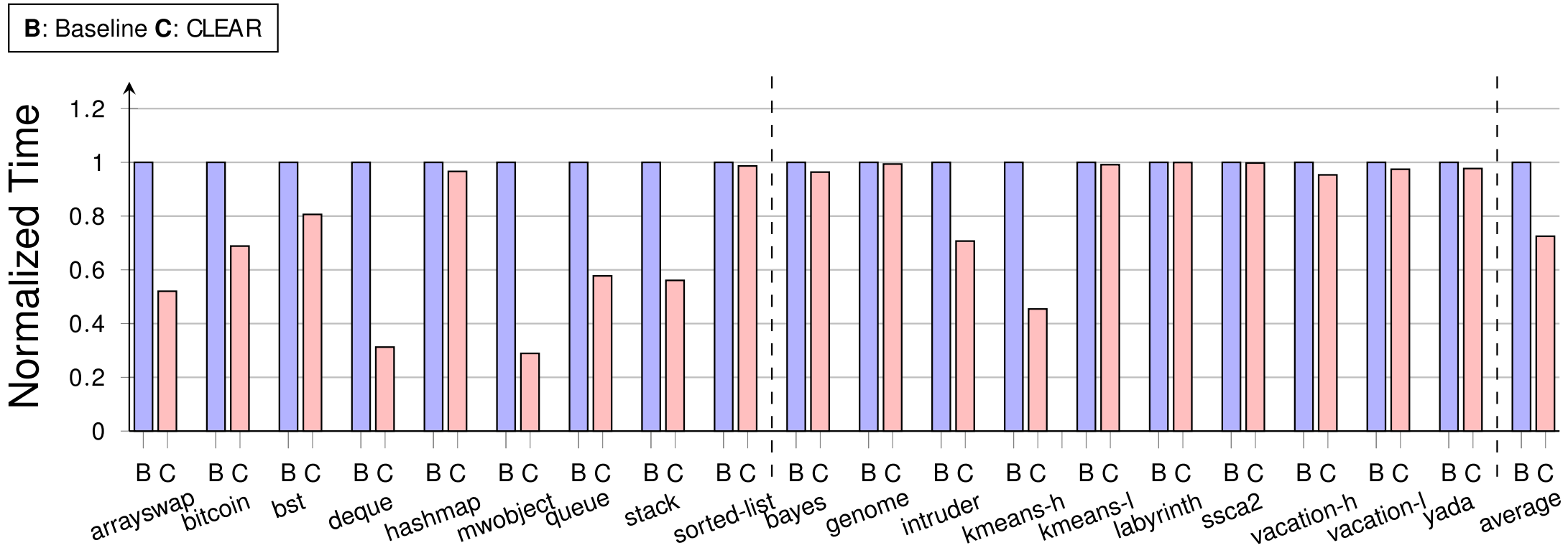
- Execution with **speculation support**
  - to recover in case of a new conflict
- Written (and conflicting) addresses are locked in cache
  - to prevent conflicts
- When the execution finishes (either successfully or not), all the locked addresses are unlocked
- As the section is **mutable**, it is marked to prevent future discovery attempts



cleAR: Bounding TM to a single retry - Implementation

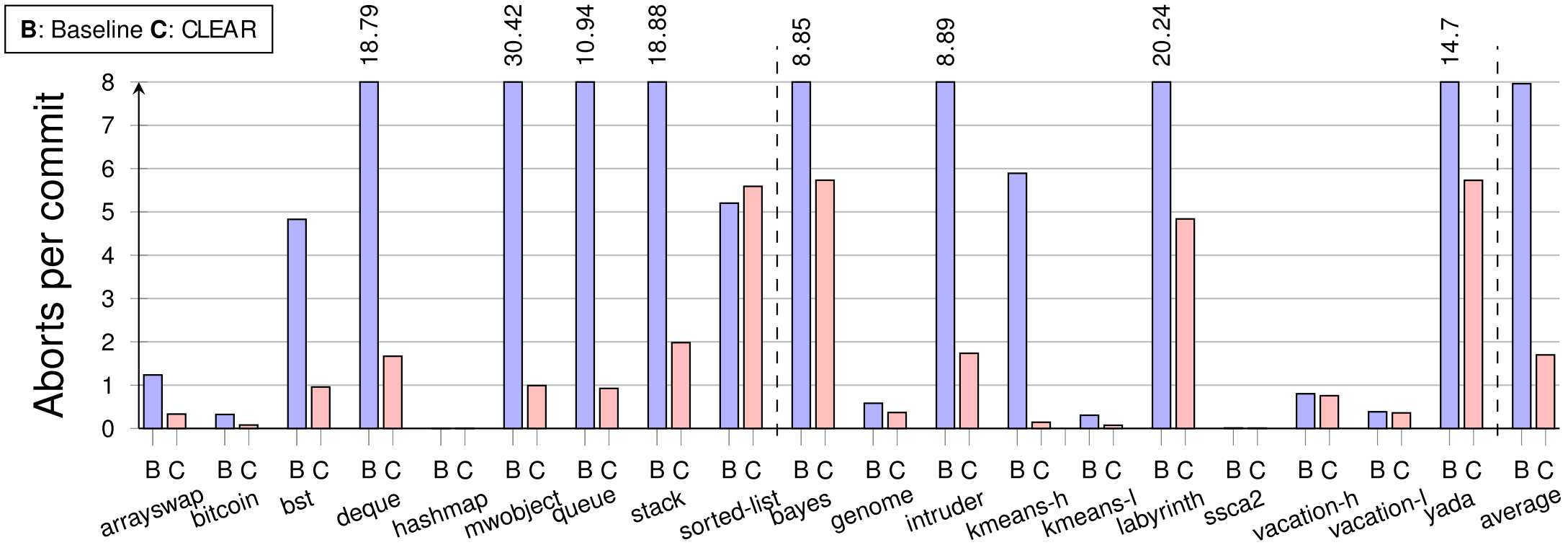


CLEAR: Bounding TM to a single retry



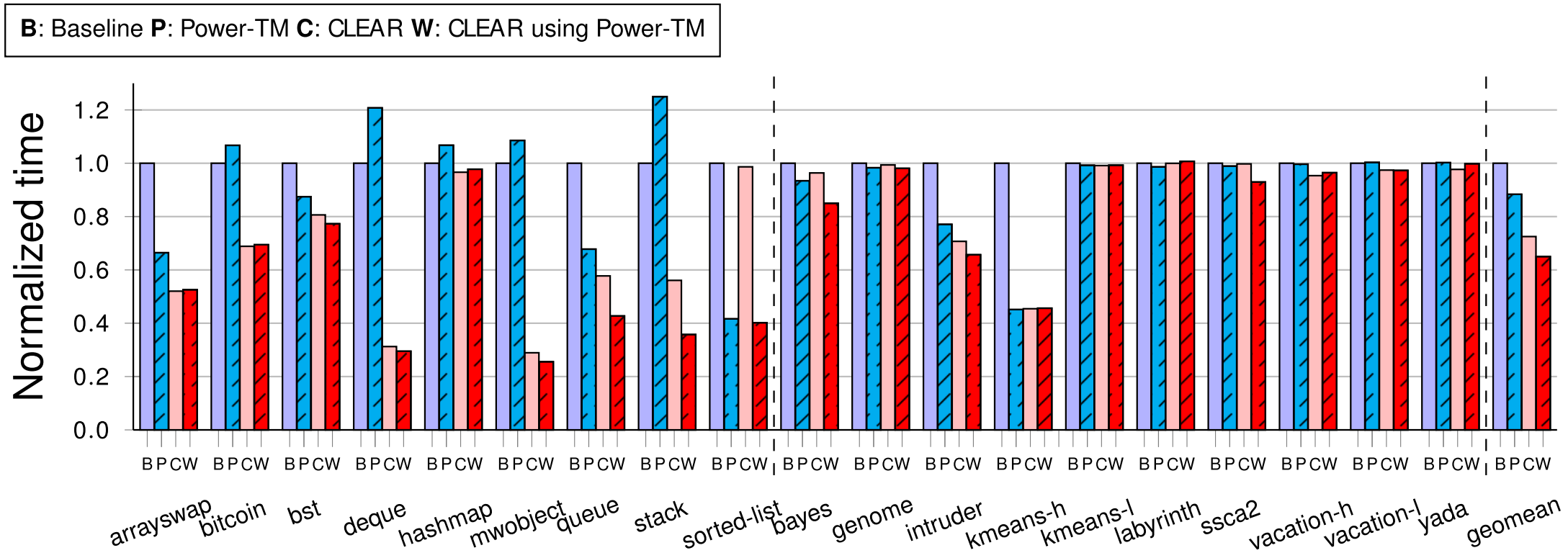
CLEAR shows over a 22% improvement over the baseline Requester Wins

CLEAR: Bounding TM to a single retry



CLEAR reduces the amount of aborts per commit from 8 to 2

CLEAR: Bounding TM to a single retry



CLEAR when implemented atop Power-TM offers 23% improvement over Power-TM

Conclusion

Conclusion

- **Non-speculative and concurrent execution of critical sections is a problem far from being solved**

Conclusion

- Non-speculative and concurrent execution of critical sections is a problem far from being solved

- **Splash-4 try to show why this issue matters and why the community needs to continue updating benchmarks**

Conclusion

- Non-speculative and concurrent execution of critical sections is a problem far from being solved
- Splash-4 try to show why this issue matters and why the community needs to continue updating benchmarks
- **Developed a methodology that allows non-speculative, efficient, and deadlock-free, to lock multiple cachelines at the same time to perform a multiple address atomic operation**

Conclusion

- Non-speculative and concurrent execution of critical sections is a problem far from being solved
- Splash-4 try to show why this issue matters and why the community needs to continue updating benchmarks
- Developed a methodology that allows non-speculative, efficient, and deadlock-free, to lock multiple cachelines at the same time to perform a multiple address atomic operation
- **Introduced a new method that can determine the data used by the section and perform a non-speculative re-execution of the section to guarantee its success in just one retry**

Future Lines

- **The locking is assumed to be executed in order, which may introduce a bottleneck in certain situations.**

Future Lines

- The locking is assumed to be executed in order, which may introduce a bottleneck in certain situations.

- **One lex order may not be enough, this happens with two or more shared structures that have enough capacity but with different indexing policies**

Future Lines

- The locking is assumed to be executed in order, which may introduce a bottleneck in certain situations.
- One lex order may not be enough, this happens with two or more shared structures that have enough capacity but with different indexing policies
- **Explore is the combination of locking and SIMD instructions**

Future Lines

- The locking is assumed to be executed in order, which may introduce a bottleneck in certain situations.
- One lex order may not be enough, this happens with two or more shared structures that have enough capacity but with different indexing policies
- Explore is the combination of locking and SIMD instructions
- **The Splash benchmark suite still has some issues**

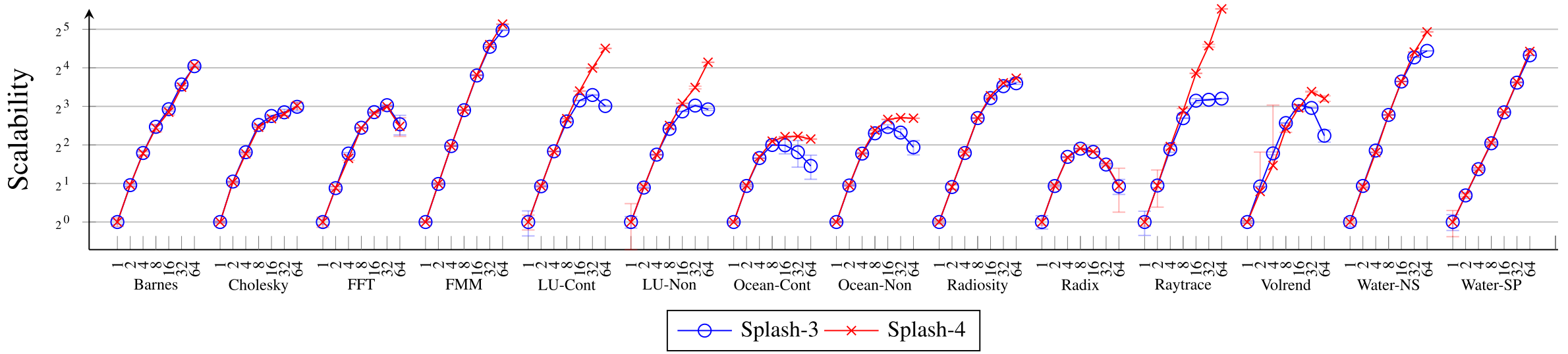
Future Lines

- The locking is assumed to be executed in order, which may introduce a bottleneck in certain situations.
- One lex order may not be enough, this happens with two or more shared structures that have enough capacity but with different indexing policies
- Explore is the combination of locking and SIMD instructions
- The Splash benchmark suite still has some issues
- **We wanted to explore how the compiler could help with the conversion of critical sections and transactions**

Thank you!

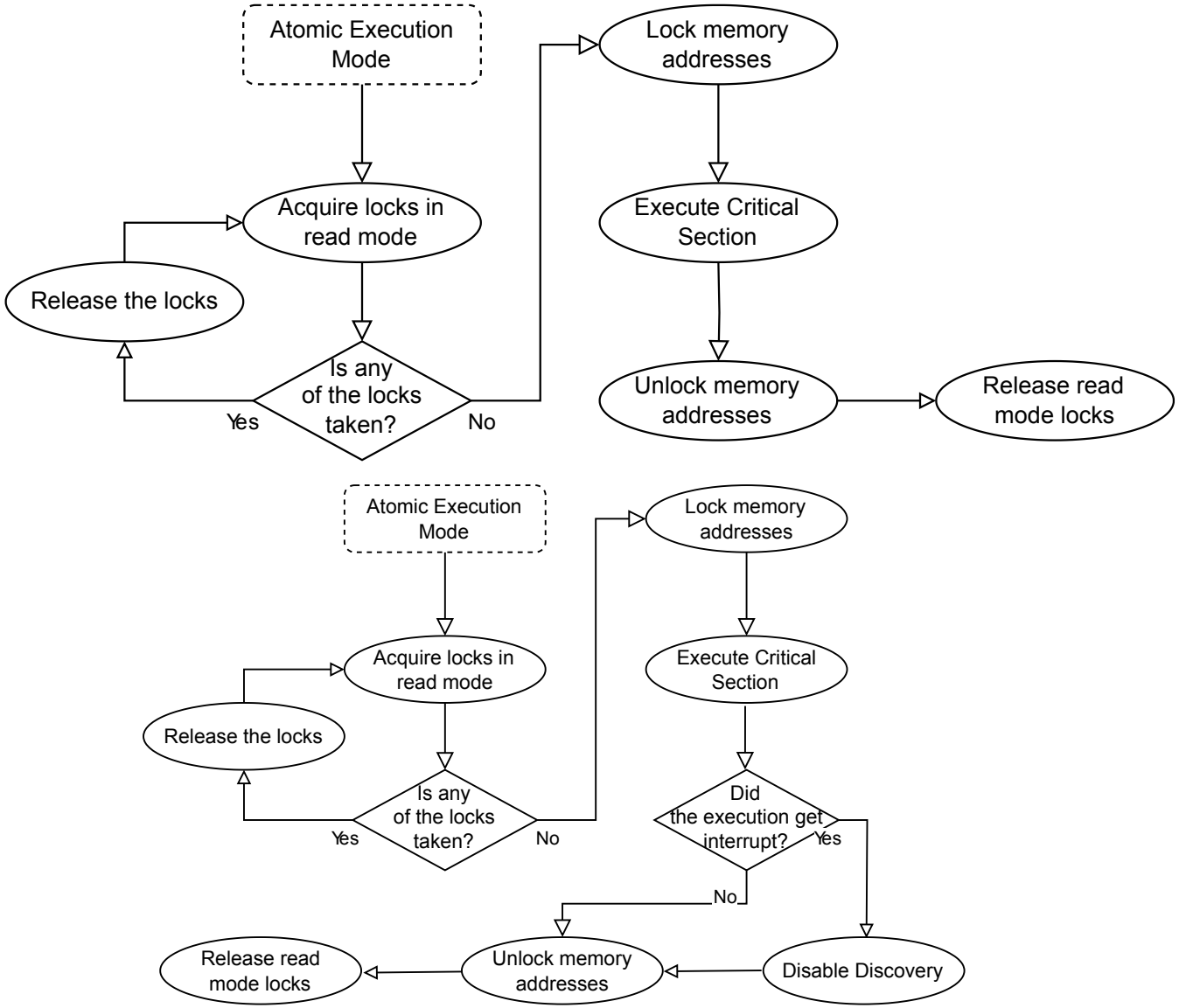
Questions?

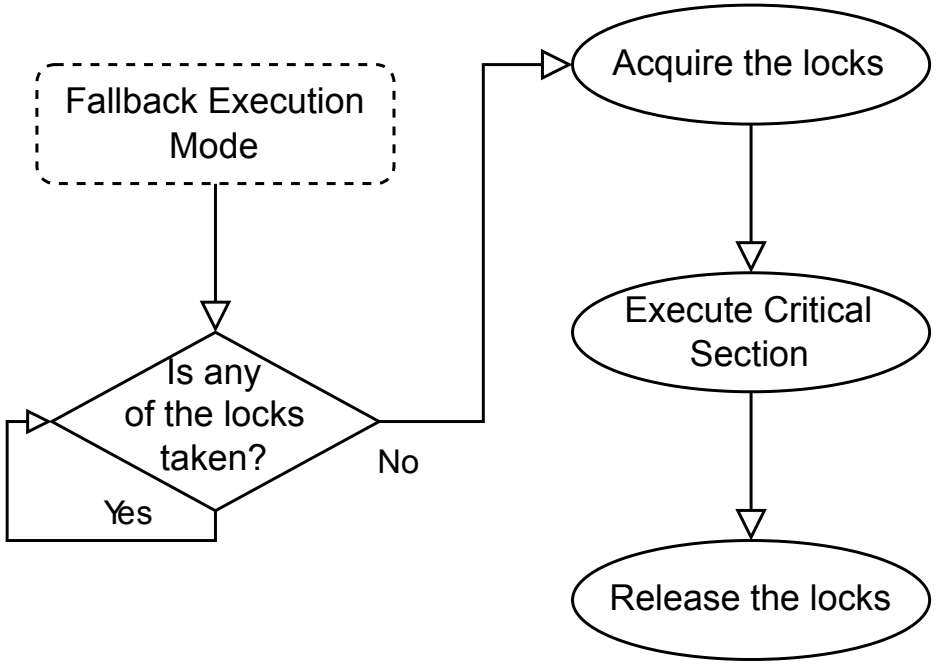
Splash-4



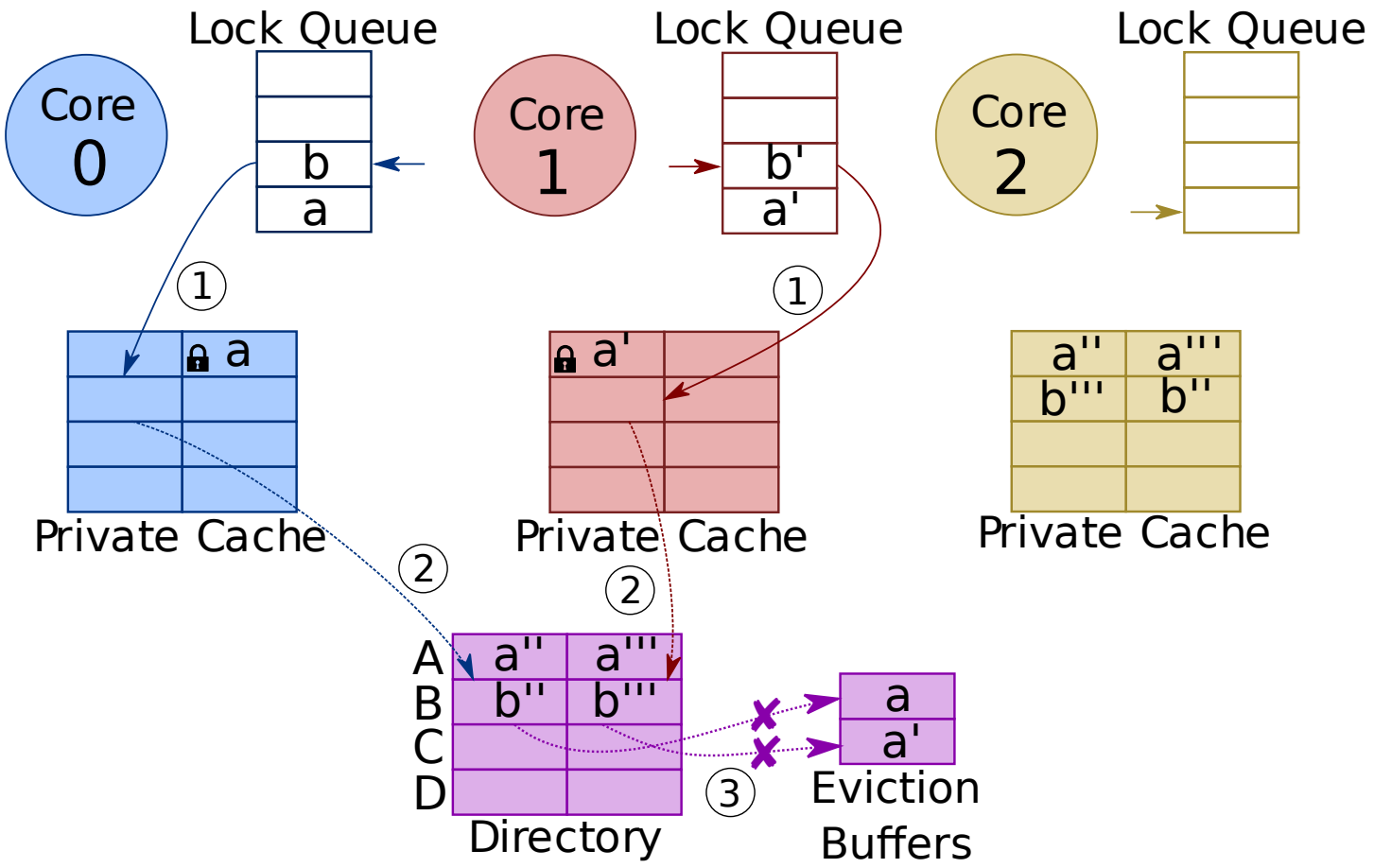
Splash-4 when executed in the gem5 simulator shows significant scalability improvements

CLEAR

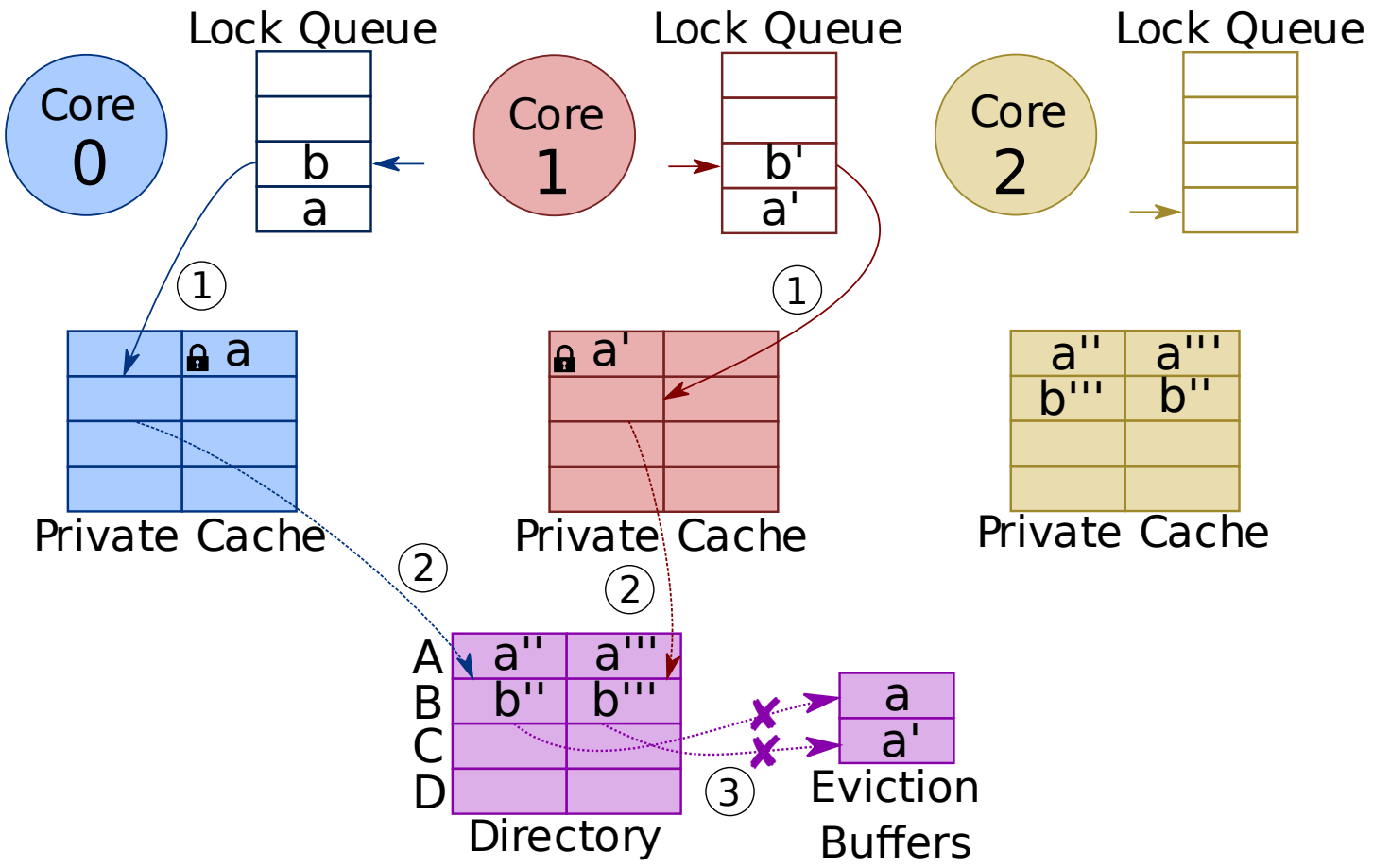

Atomic Execution Mode
Lock memory addresses
Acquire locks in read mode
Execute Critical Section
Release the locks
Is any of the locks taken?
Yes
No
Unlock memory addresses
Release read mode locks
Atomic Execution Mode
Lock memory addresses
Acquire locks in read mode
Execute Critical Section
Release the locks
Is any of the locks taken?
Yes
No
Did the execution get interrupt?
Yes
No
Release read mode locks
Unlock memory addresses
Disable Discovery


Fallback Execution Mode
Is any of the locks taken?
Yes
No
Acquire the locks
Execute Critical Section
Release the locks

Hardware Multi-Address Atomics – Deadlocks - MSHRs

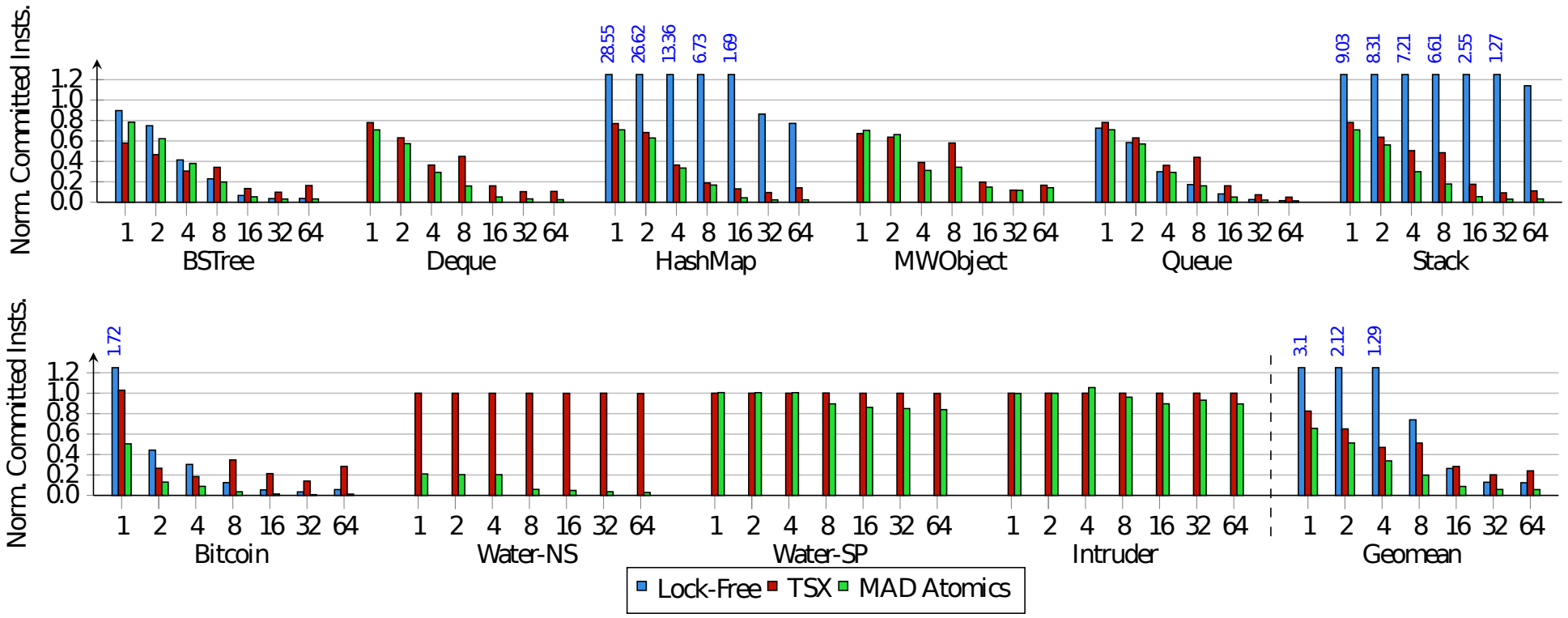


Hardware Multi-Address Atomics – Deadlocks - MSHRs



Replacements when the eviction buffers are occupied must be taken in the cache itself (in-situ)

Hardware Multi-Address Atomics



Multi-Address Atomics reduces the amount of instructions commited even further than the lock-free approaches.