Lazy Release Persistency

Mahesh Dananjaya, Vasilis Gavrielatos, *Arpit Joshi, Vijay Nagarajan

University of Edinburgh, Intel*

Outline

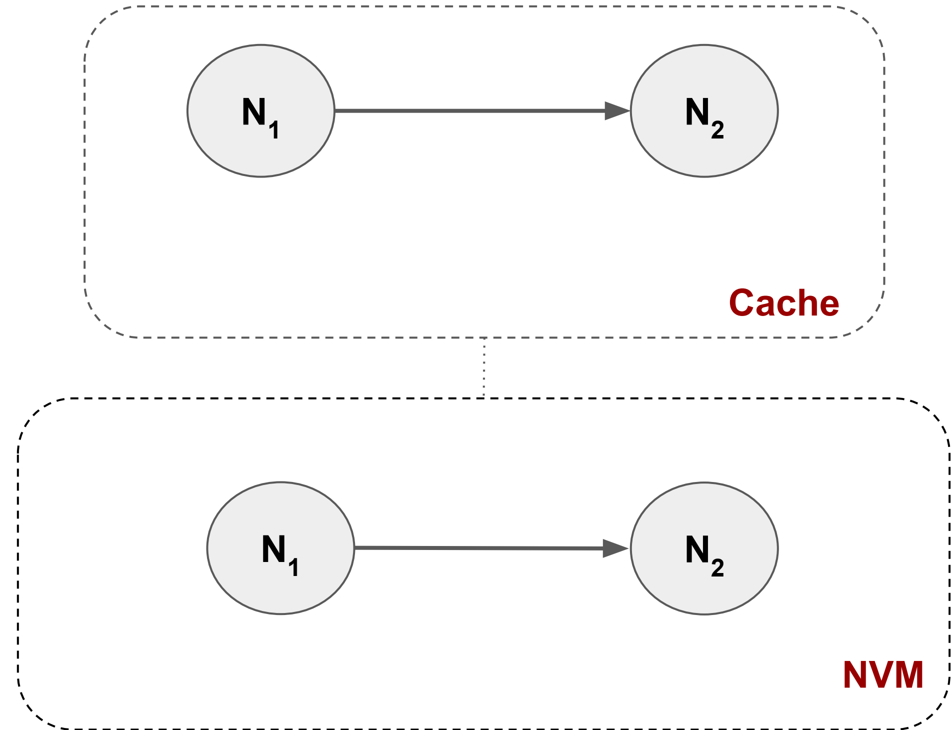
- Release Persistency (RP)
  - Strengthens existing language-level persistency model
  - Strong enough to enable recovery of log-free data structures (LFDs)
- Lazy Release Persistency (LRP)
  - Microarchitectural mechanism to efficiently enforce RP

Persistent Memory (PM)

Non-volatile memory (NVM)

- Fast, byte-addressable persistent storage
- Promises cheap program recovery on a crash
- Program recovery requires crash consistency

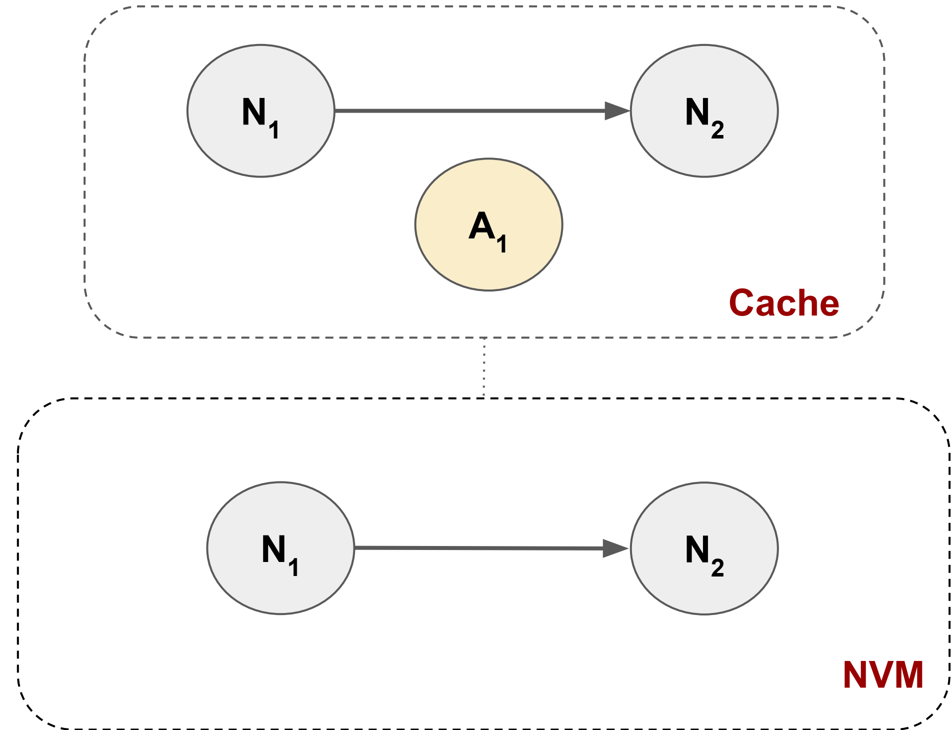
Persistent Memory (PM)



THE UNIVERSITY of EDINBURGH
informatics

Persistent Memory (PM)

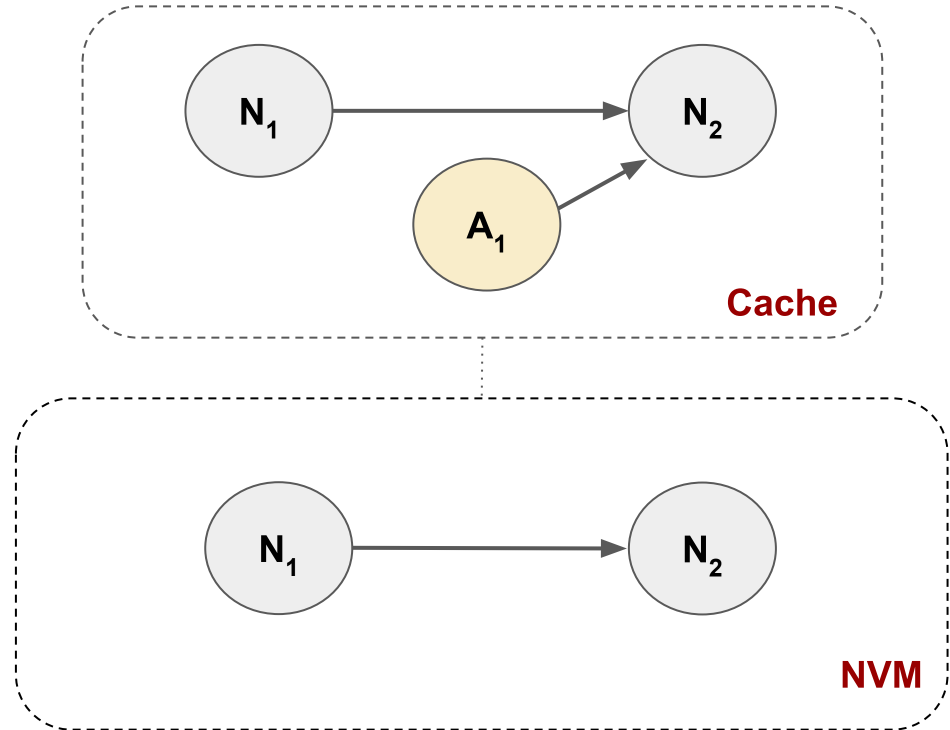
1. Create Node



THE UNIVERSITY of EDINBURGH informatics

Persistent Memory (PM)

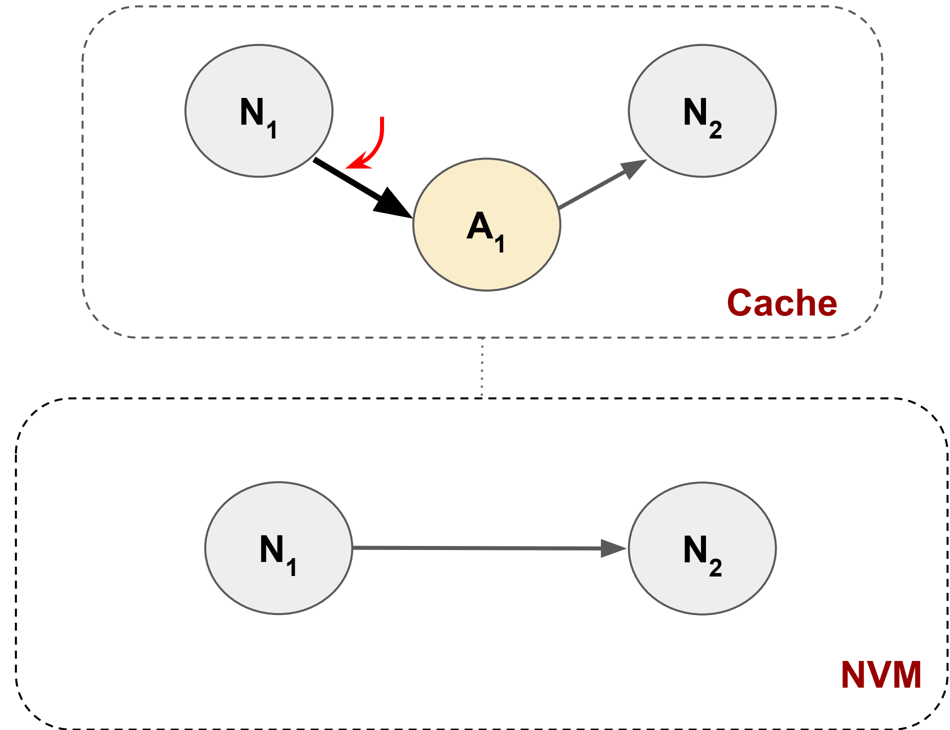
1. Create Node
2. Update Pointer



THE UNIVERSITY of EDINBURGH
informatics

Persistent Memory (PM)

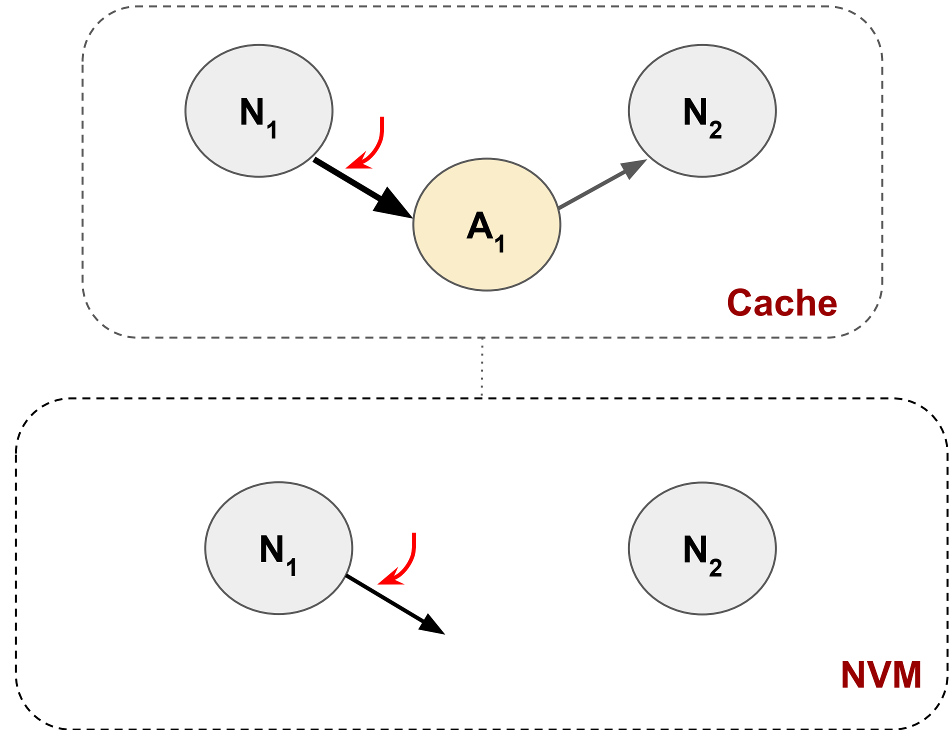
1. Create Node
2. Update Pointer
3. Update Base Pointer



THE UNIVERSITY of EDINBURGH
informatics

Persistent Memory (PM)

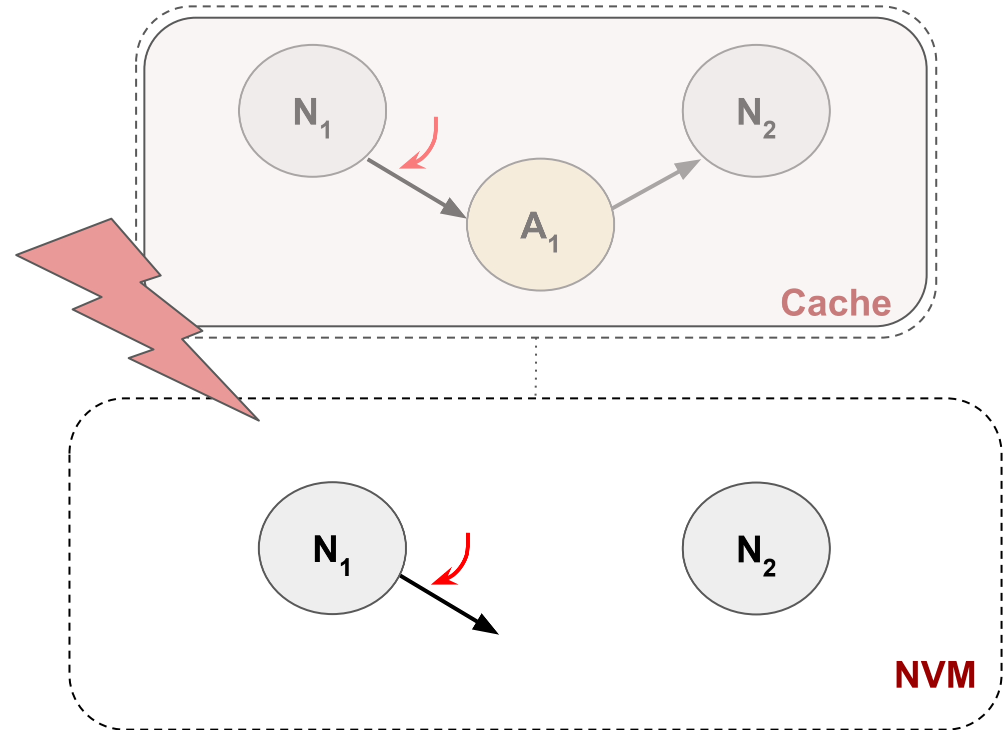
1. Create Node
2. Update Pointer
3. Update Base Pointer



THE UNIVERSITY of EDINBURGH
informatics

Persistent Memory (PM)

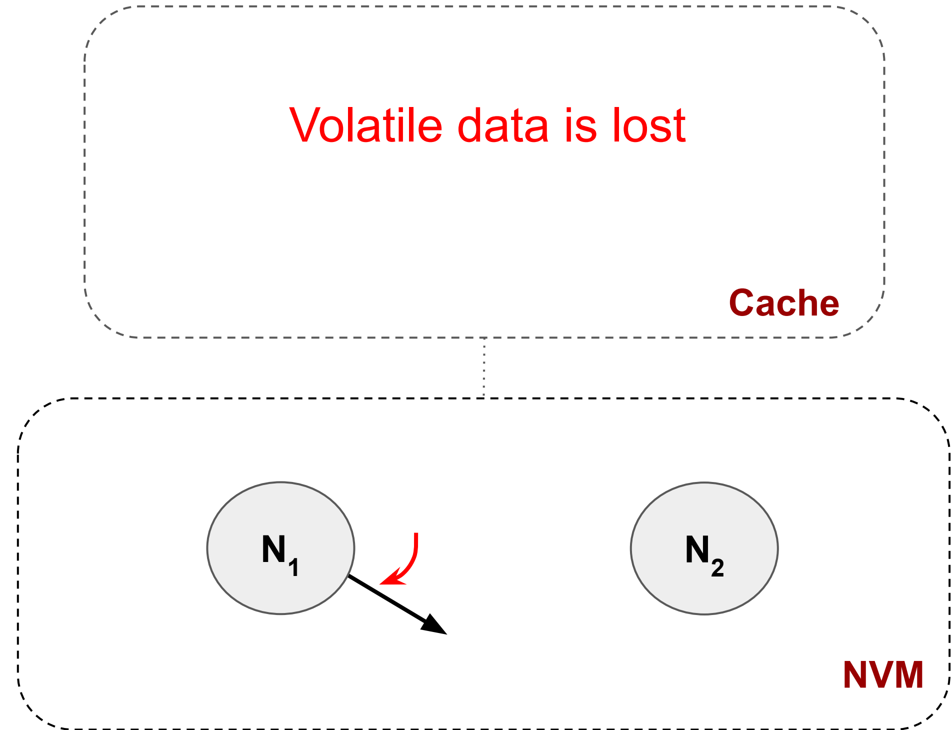
1. Create Node
2. Update Pointer
3. Update Base Pointer



THE UNIVERSITY of EDINBURGH
informatics

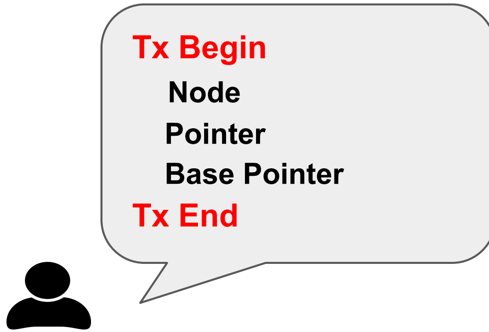
Persistent Memory (PM)

1. Create Node
2. Update Pointer
3. Update Base Pointer



THE UNIVERSITY of EDINBURGH
informatics

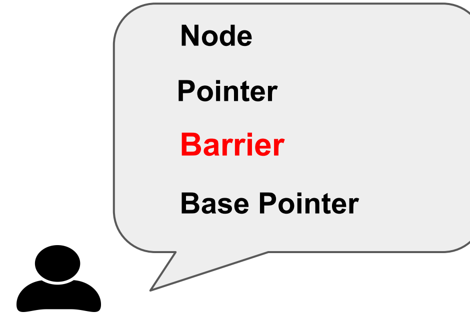
What Primitives for Crash Consistency?



Failure Atomicity

SFR [Gogte et al]

Atlas [Chakrabarti et al]

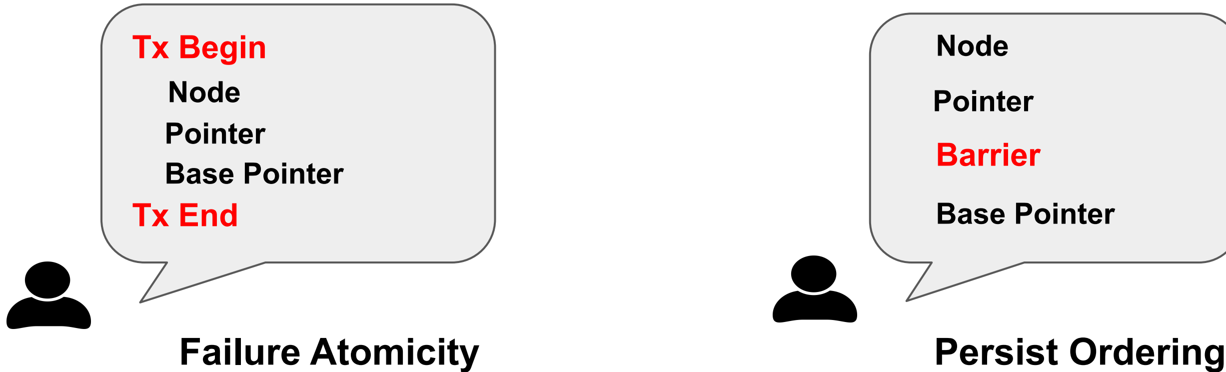


Persist Ordering

ARP [Kolli et al]

THE UNIVERSITY of EDINBURGH
informatics

What Primitives for Crash Consistency?



Goal: Sweet-spot between programmability and performance

What Primitives for Crash Consistency?

Tx Begin
Node
Pointer
Base Pointer
Tx End

Failure Atomicity

What Primitives for Crash Consistency?

Tx Begin
Node
Pointer
Base Pointer
Tx End

Failure Atomicity

Programmability

What Primitives for Crash Consistency?

Tx Begin
Node
Pointer
Base Pointer
Tx End

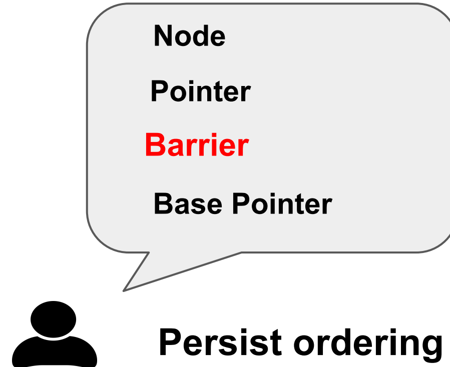
Failure Atomicity

Programmability

Performance

THE UNIVERSITY *of* EDINBURGH
informatics

What Primitives for Crash Consistency?



What Primitives for Crash Consistency?

Node

Pointer

Barrier

Base Pointer

Persist ordering

Programmability

What Primitives for Crash Consistency?

Node

Pointer

Barrier

Base Pointer

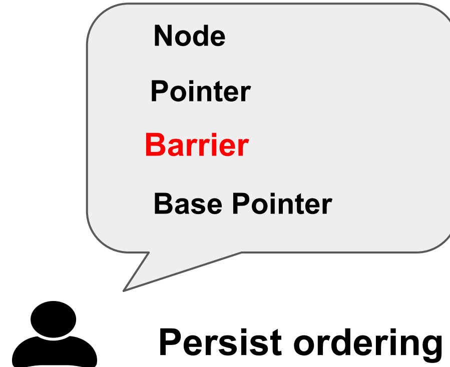
Persist ordering

Programmability

Performance

THE UNIVERSITY of EDINBURGH informatics

What Primitives for Crash Consistency?

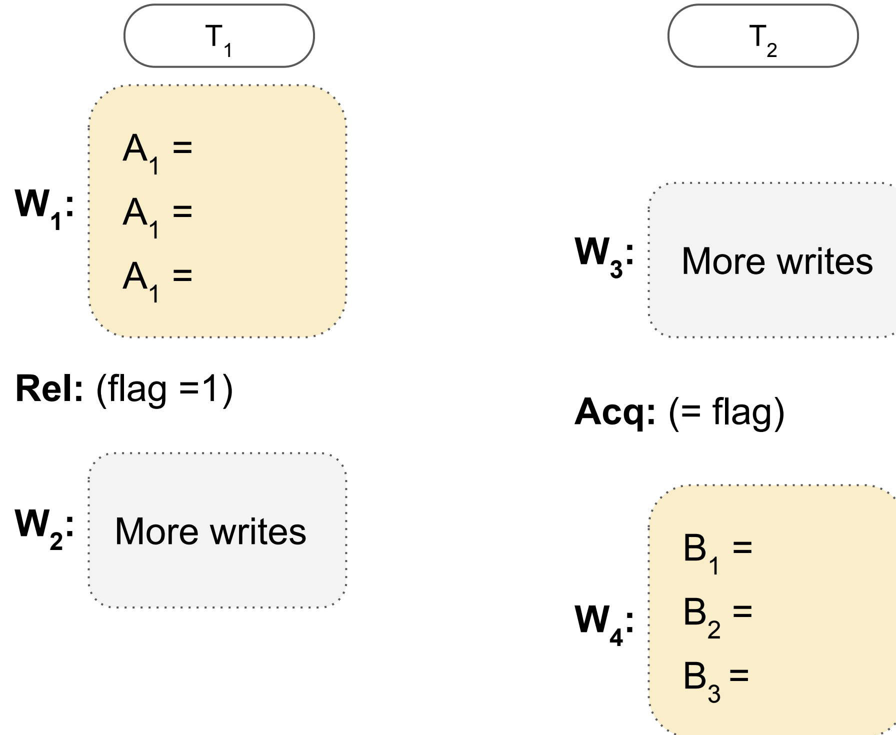


👎 Programmability

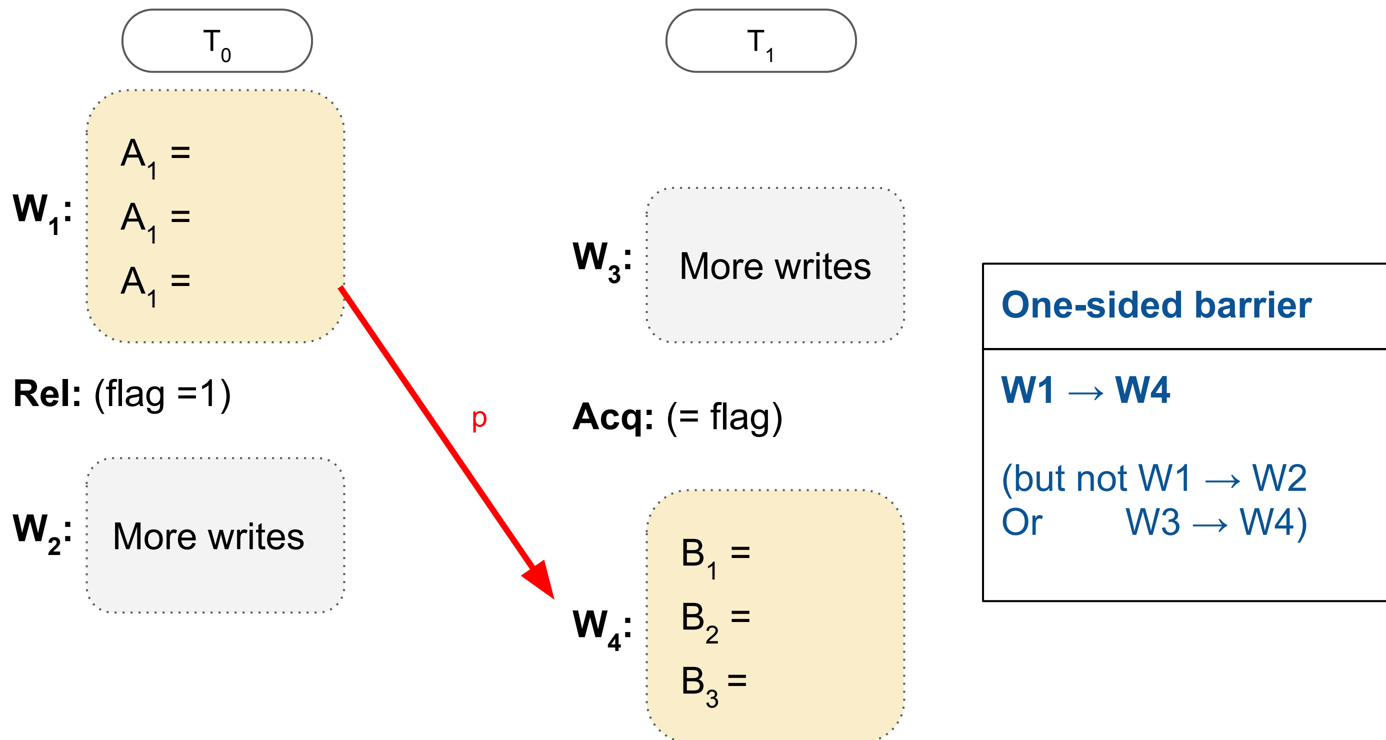
👍 Performance

ARP one-sided barriers [Kolli et al.]

Acquire-Release Persistency (ARP)



Acquire-Release Persistency (ARP)



Our View: Atomicity vs Ordering

Atomicity is programmable but ordering important too
(for ***log-free*** data structures, LFDs)

Our View: Atomicity vs Ordering

Atomicity is programmable but ordering important too
(for ***log-free*** data structures, LFDs)

LFDs are important use-case for persistent memory
[Izraelevitz ‘17, David ‘18, Belloche ‘18]

Our View: Atomicity vs Ordering

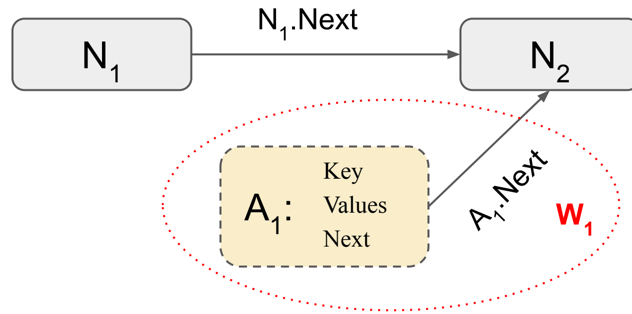
Atomicity is programmable but ordering important too
(for ***log-free*** data structures, LFDs)

LFDs are important use-case for persistent memory
[Izraelevitz '17, David '18, Belloche '18]

But ARP is not strong enough for enabling recovery of LFDs!

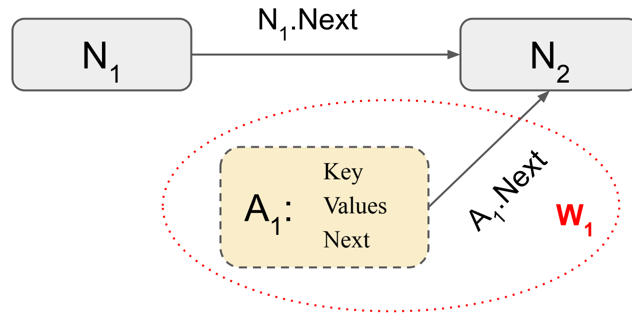
Multithreaded Log-free Linked List: Thread 1

step 1:

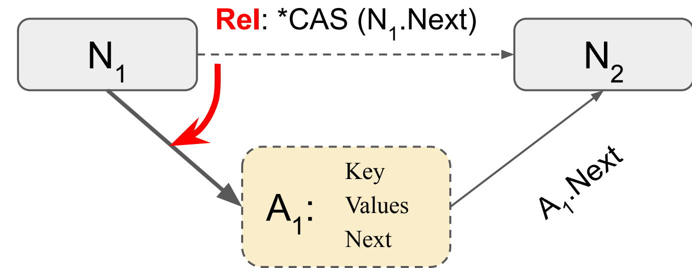


Multithreaded Log-free Linked List: Thread 1

step 1:



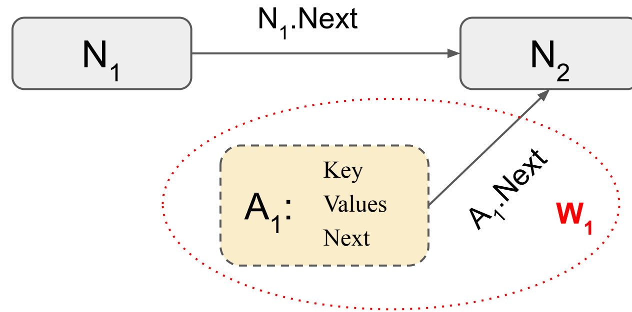
step 2:



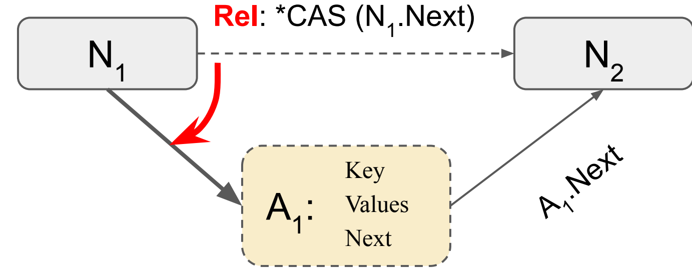
***CAS :** Compare-and-swap is a **read-modify-write** (RMW) instruction.

Multithreaded Log-free Linked List: Thread 1

step 1:



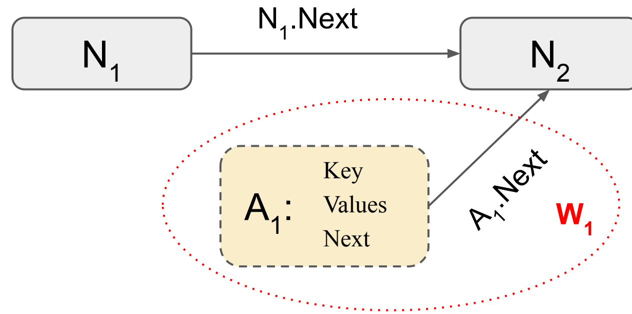
step 2:



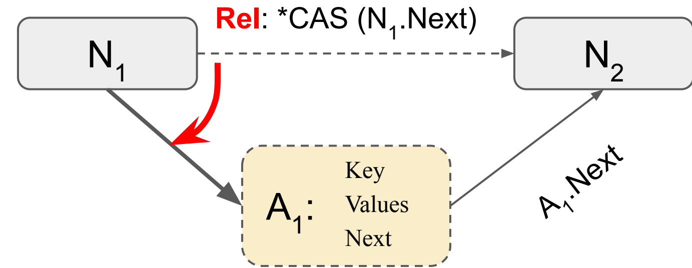
Inconsistent PM

Multithreaded Log-free Linked List: Thread 1

step 1:



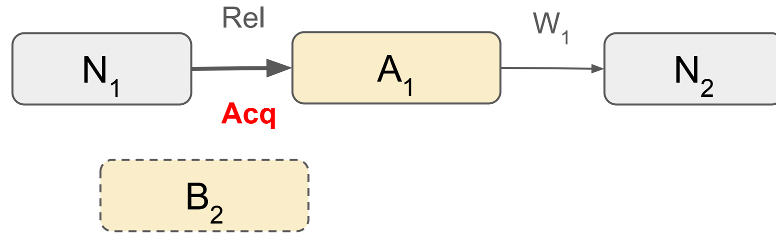
step 2:



$W_1 \rightarrow Rel$

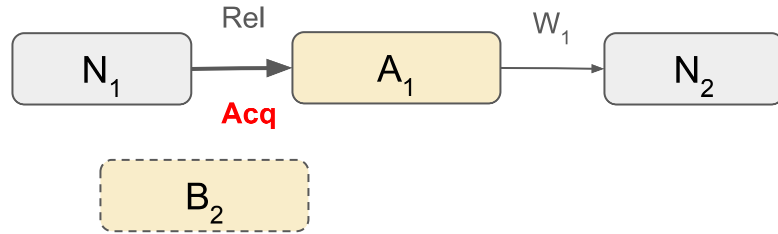
Multithreaded Log-free Linked List: Thread 2

step 3:

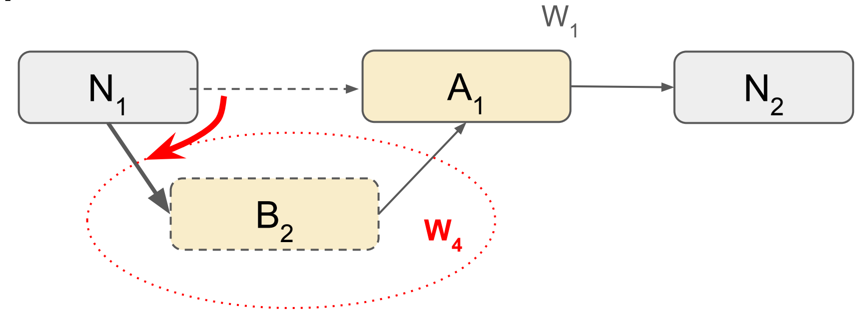


Multithreaded Log-free Linked List: Thread 2

step 3:

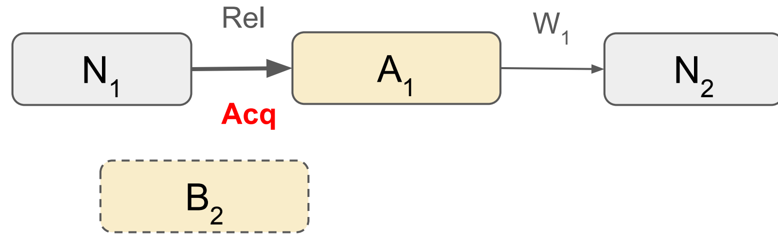


step 4:

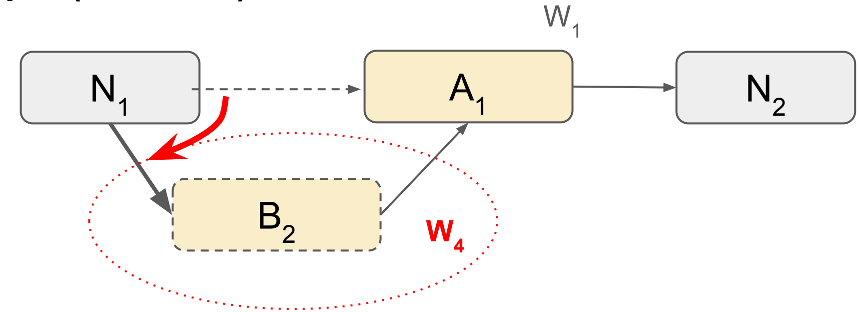


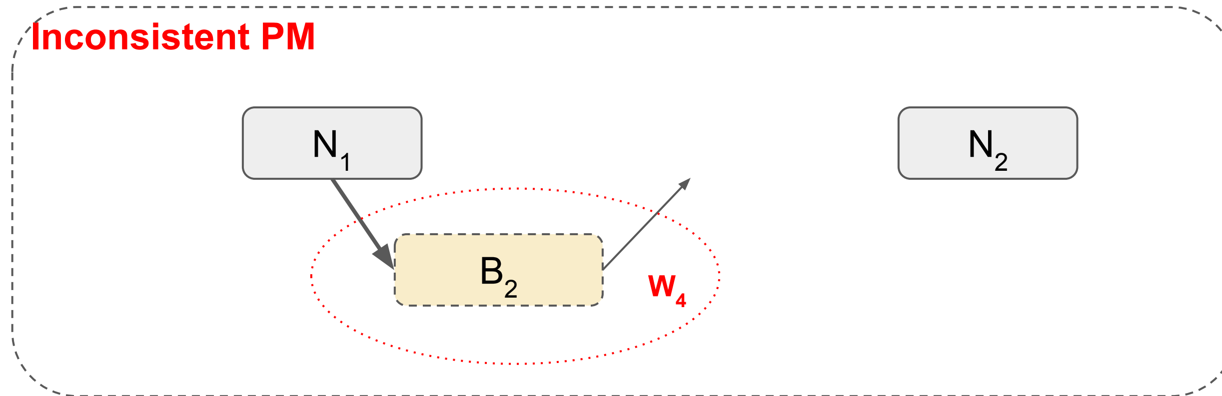
Multithreaded Log-free Linked List: Thread 2

step 3 (Thread 2):



step 4 (Thread 2):

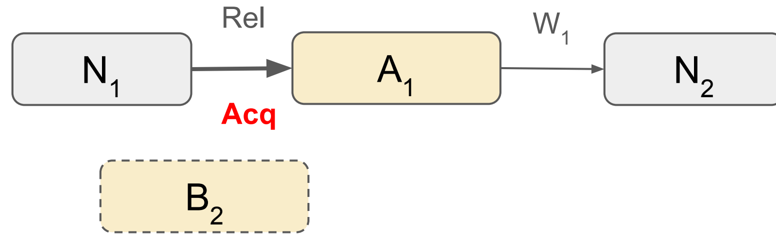




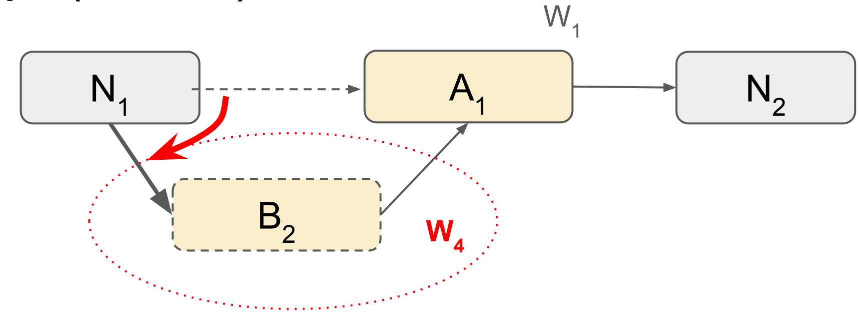
THE UNIVERSITY of EDINBURGH
informatics

Multithreaded Log-free Linked List: Thread 2

step 3 (Thread 2):

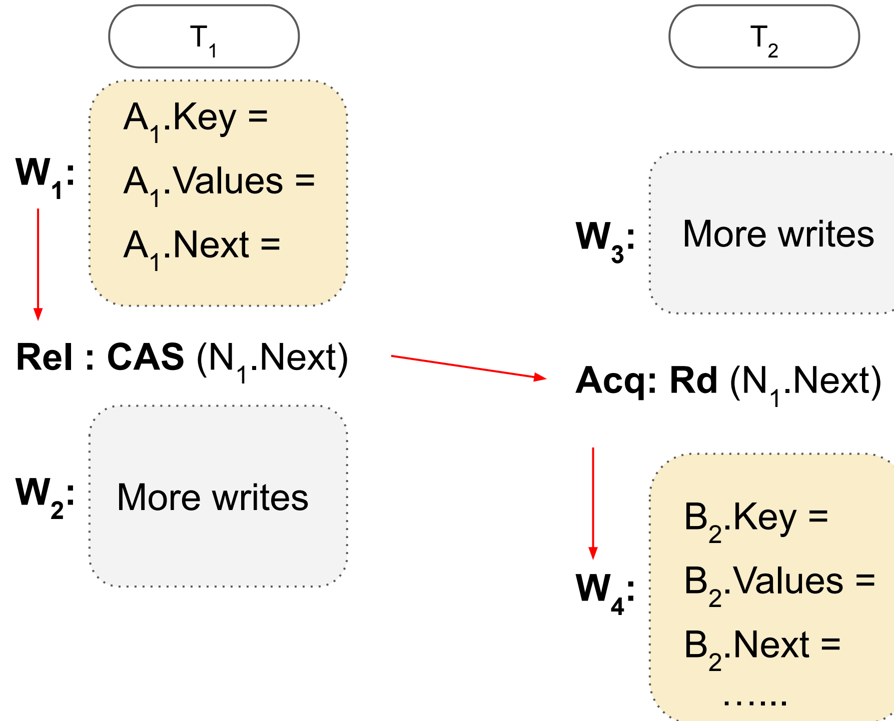


step 4 (Thread 2):



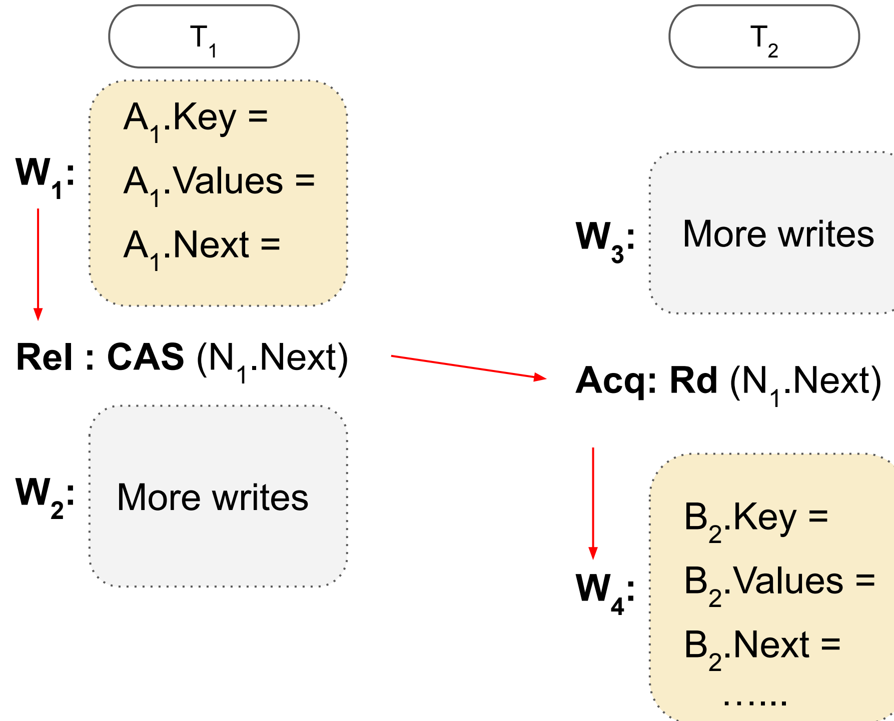
$W_1 \rightarrow W_4$

Persistent Orderings for Recovery



THE UNIVERSITY of EDINBURGH informatics

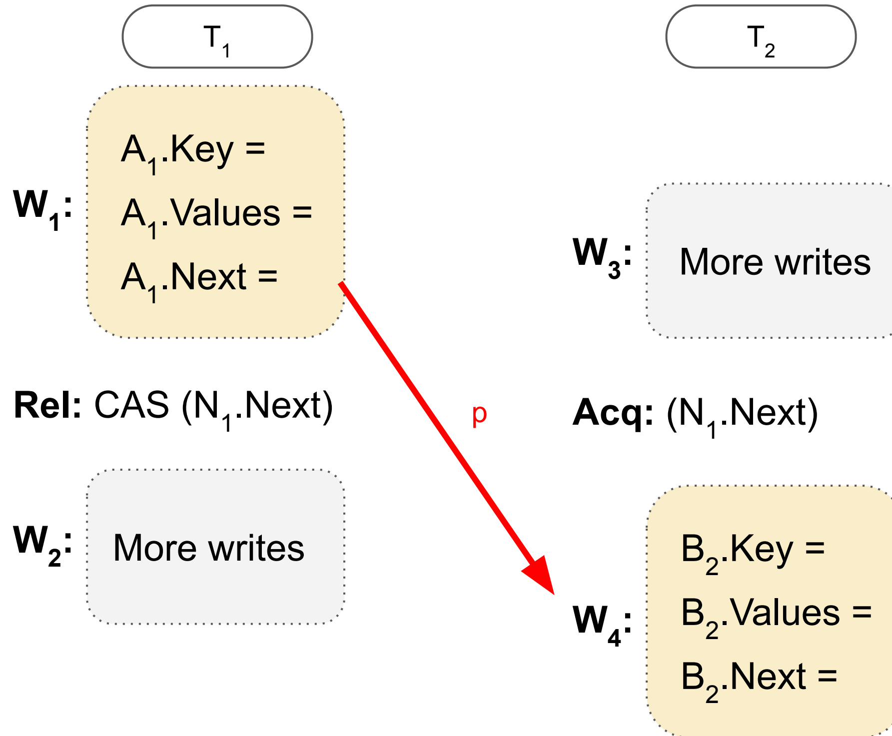
Persistent Orderings for Recovery



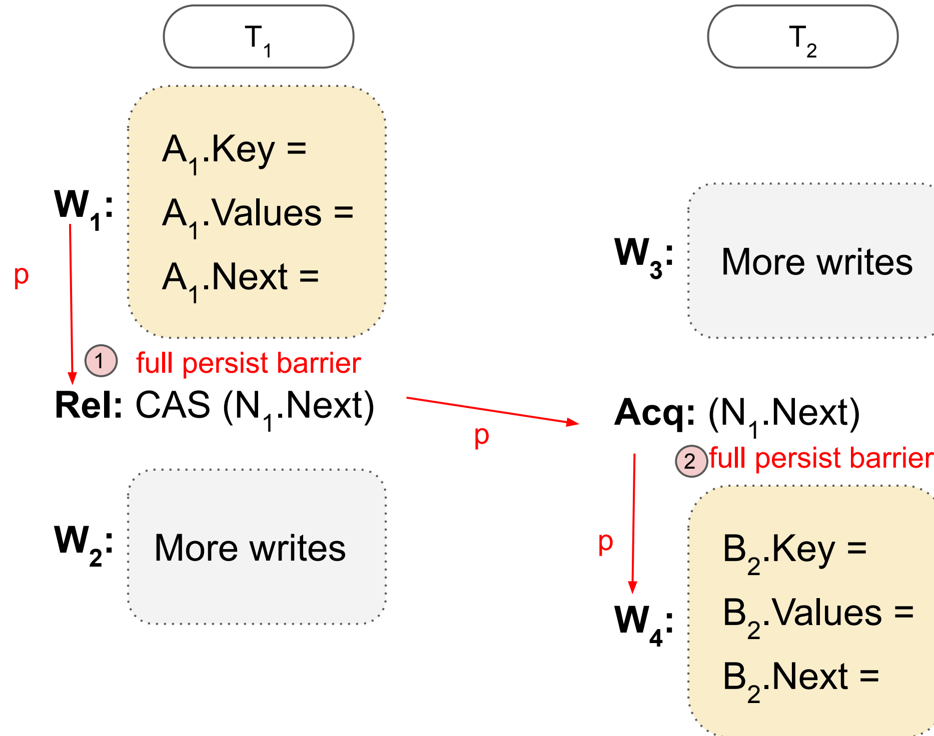
$W_1 \rightarrow \text{Rel} \rightarrow W_4$

THE UNIVERSITY of EDINBURGH
informatics

Acquire-Release Persistency (ARP) is not strong enough!

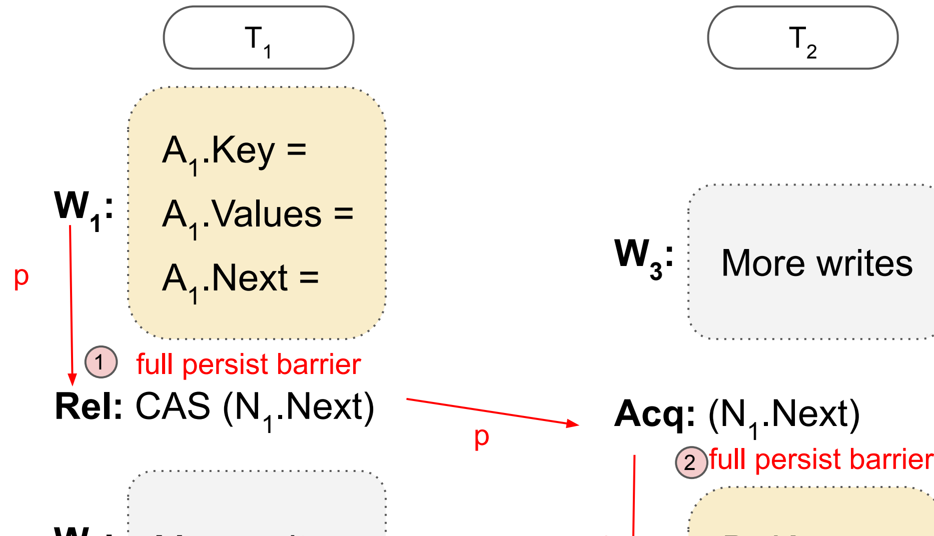


Acquire-Release Persistency (ARP) is not strong enough!



THE UNIVERSITY of EDINBURGH informatics

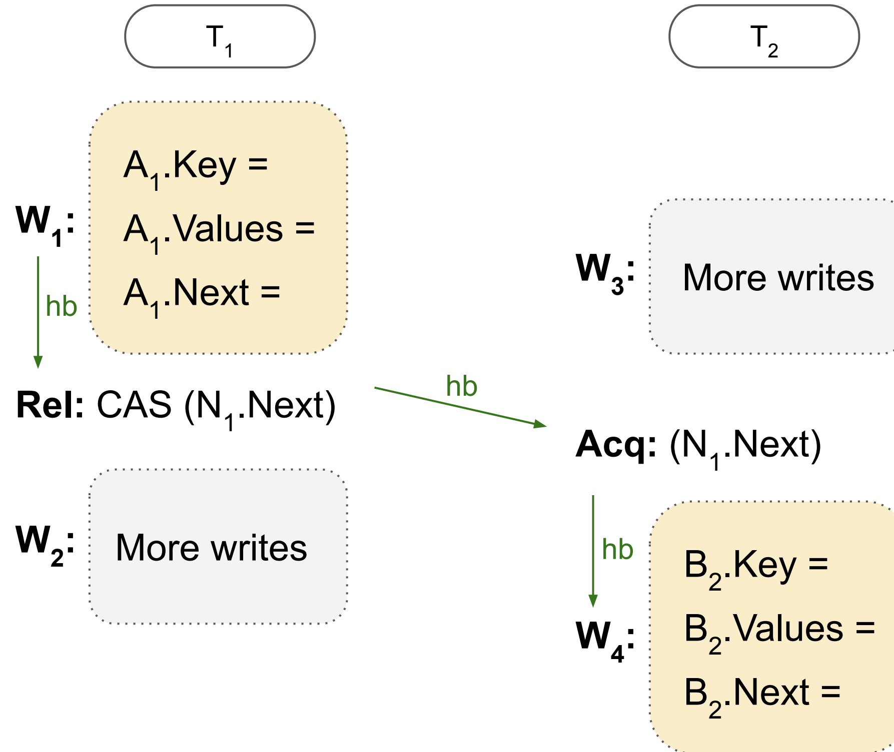
Acquire-Release Persistency (ARP) is not strong enough!



Full barriers annul the performance benefits of ARP!

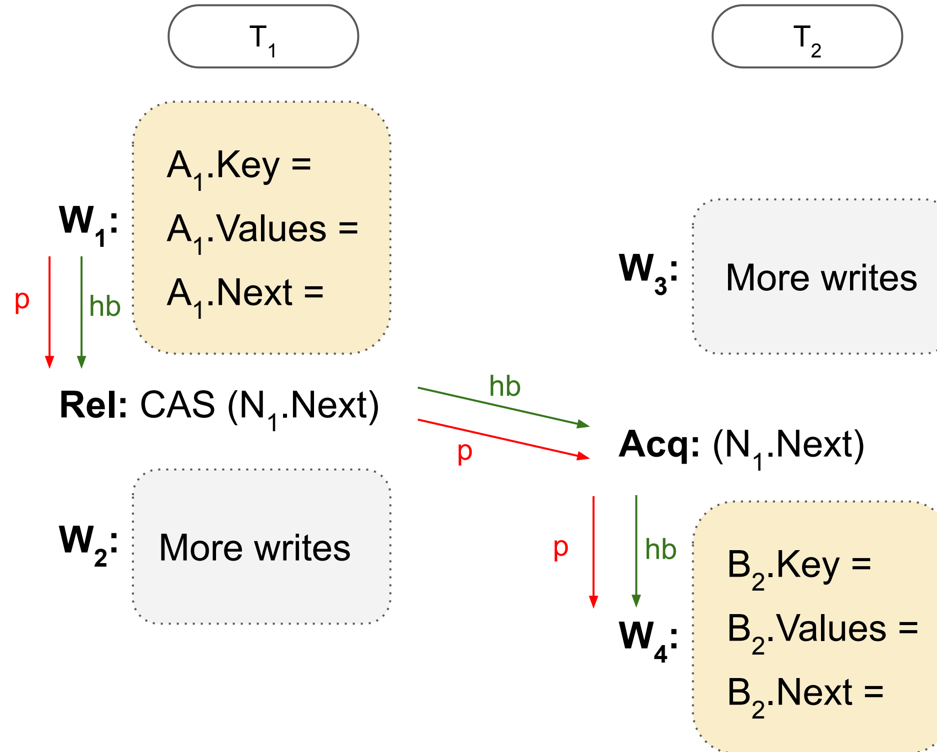
THE UNIVERSITY of EDINBURGH informatics

Release Persistency



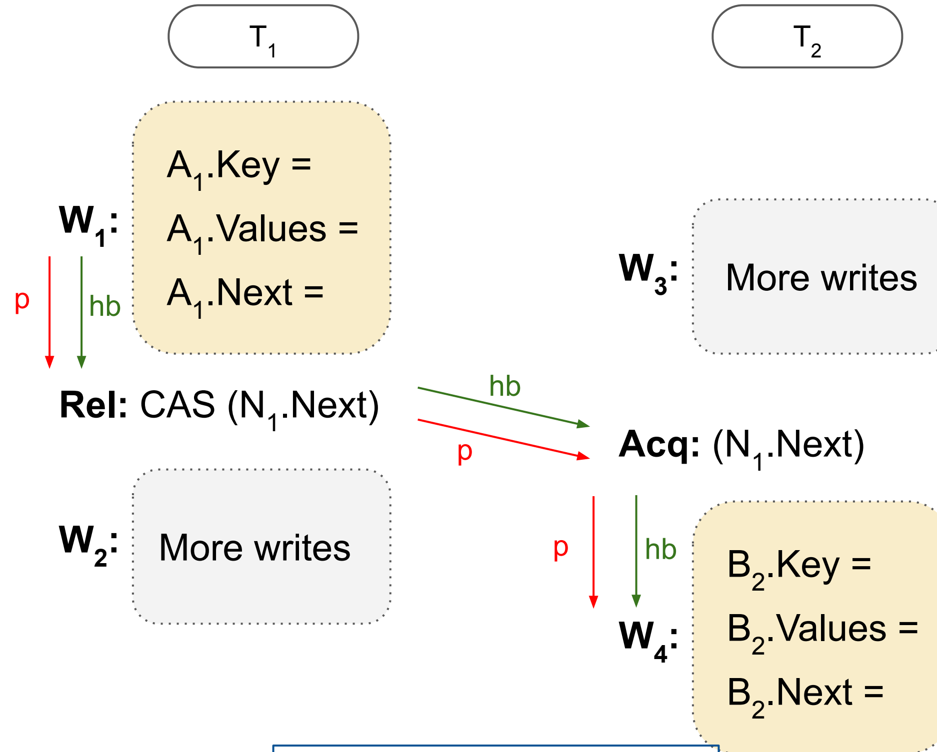
THE UNIVERSITY of EDINBURGH informatics

Release Persistency



THE UNIVERSITY of EDINBURGH
informatics

Release Persistency



$W_1 \rightarrow$ Rel $\rightarrow W_4$

Release Persistency

T_1

T_2

A_1.Key =

Persist ordering mirrors consistency happens-before ordering

(sufficient for LFD recoverability [Izraelevitz and Scott '16])

W_2: More writes

p

hb

W_4:

B_2.Key =

B_2.Values =

B_2.Next =

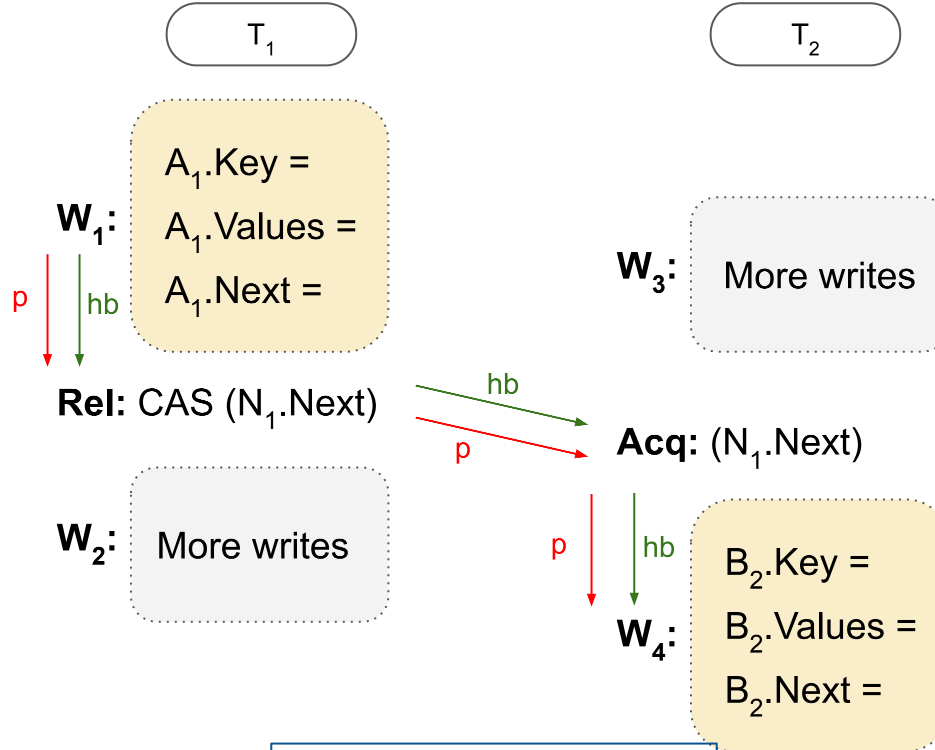
$W_1 \rightarrow Rel \rightarrow W_4$

THE UNIVERSITY of EDINBURGH
informatics

Release Persistency

One-sided

LFDs



$W_1 \rightarrow Rel \rightarrow W_4$

THE UNIVERSITY of EDINBURGH
informatics

How to Enforce RP Efficiently?

- Requirements
  - $\mathbf{W_1 \rightarrow Rel \rightarrow W_4}$
  - But the implementation needs to be buffered

    (Suboptimal for W_1 to persist when the Rel performs)

THE UNIVERSITY *of* EDINBURGH
informatics

How to Enforce RP Efficiently?

- Requirements
  - **$W_1 \rightarrow$ Rel $\rightarrow W_4$**
  - But the implementation needs to be buffered
    (Suboptimal for W_1 to persist when the Rel performs)

These requirements match a 30-year old protocol for enforcing RC lazily!

THE UNIVERSITY of EDINBURGH
informatics

Lazy Release Consistency (LRC)

- On a release
  - Write to the local cache (no need to propagate writes before the release)
- On an acquire
  - Find matching release and write-back dirty blocks in the releasing processor

Lazy Release Consistency (LRC)

- On a release
  - Write to the local cache (no need to propagate writes before the release)
- On an acquire
  - Find matching release and write-back dirty blocks in the releasing processor

Propagate data lazily to enforce RC

Lazy Release Persistency (LRP)

- On a release
  - Write to the local cache (no need to persist writes before the release)
- On an acquire
  - Find matching release and persist dirty blocks in the releasing processor

Lazy Release Persistency (LRP)

- On a release
  - Write to the local cache (no need to persist writes before the release)
- On an acquire
  - Find matching release and persist dirty blocks in the releasing processor

Persist data lazily to enforce RP

Evaluation

Benchmarks

- SynchroBench LFD suite from [*Gramoli '15*]

Comparison points.

- Strict Barrier (SB)
- Buffered two-sided Barrier (BB) *[Joshi '15]*

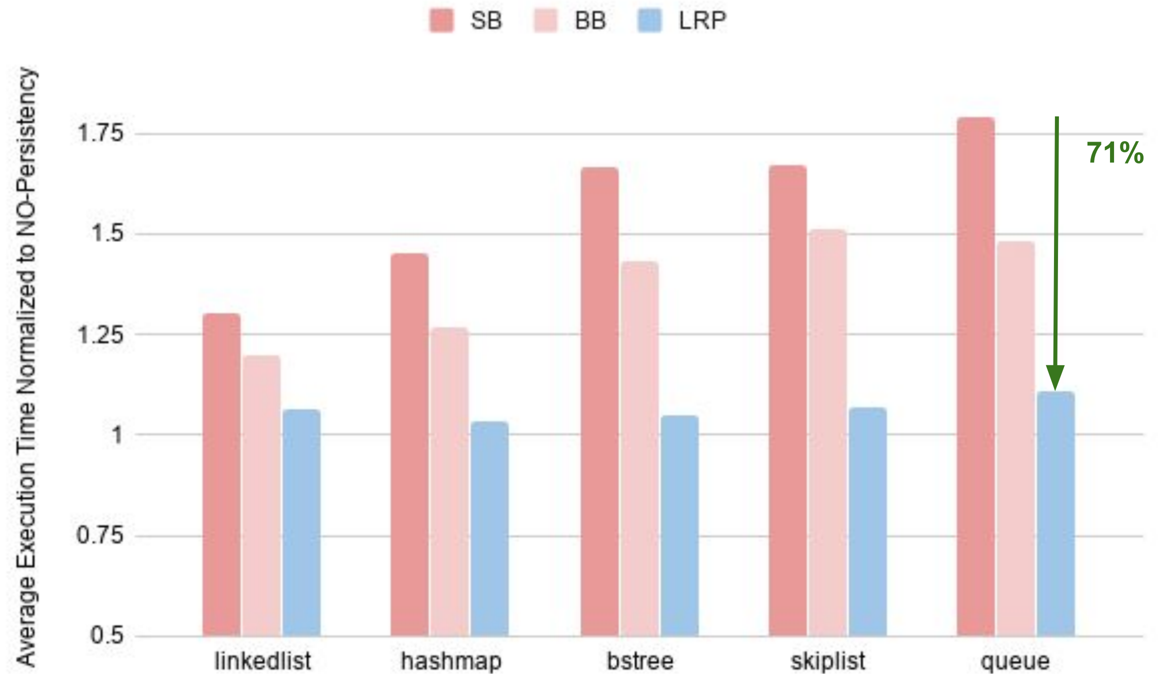
Simulation infrastructure

- PRiME simulator
- Processor: 64-core, 2.5 GHz, x86-64 (TSO)
- Caches: L1 (32K), LLC-banked (1MB per bank)
- Coherence: Directory-based MESI
- Interconnection: 2D-Mesh network
- Memory : PCM-NVRAM

Execution Time

LRP vs SB

- Upto 71% Improvement
- Average of 52%



THE UNIVERSITY of EDINBURGH
informatics

Execution Time

LRP vs SB

- Upto 71% Improvement
- Average of 52%

LRP vs BB *[Joshi '15]*

- Upto 44% Improvement
- Average of 33%

THE UNIVERSITY *of* EDINBURGH
informatics

Execution Time

LRP vs SB

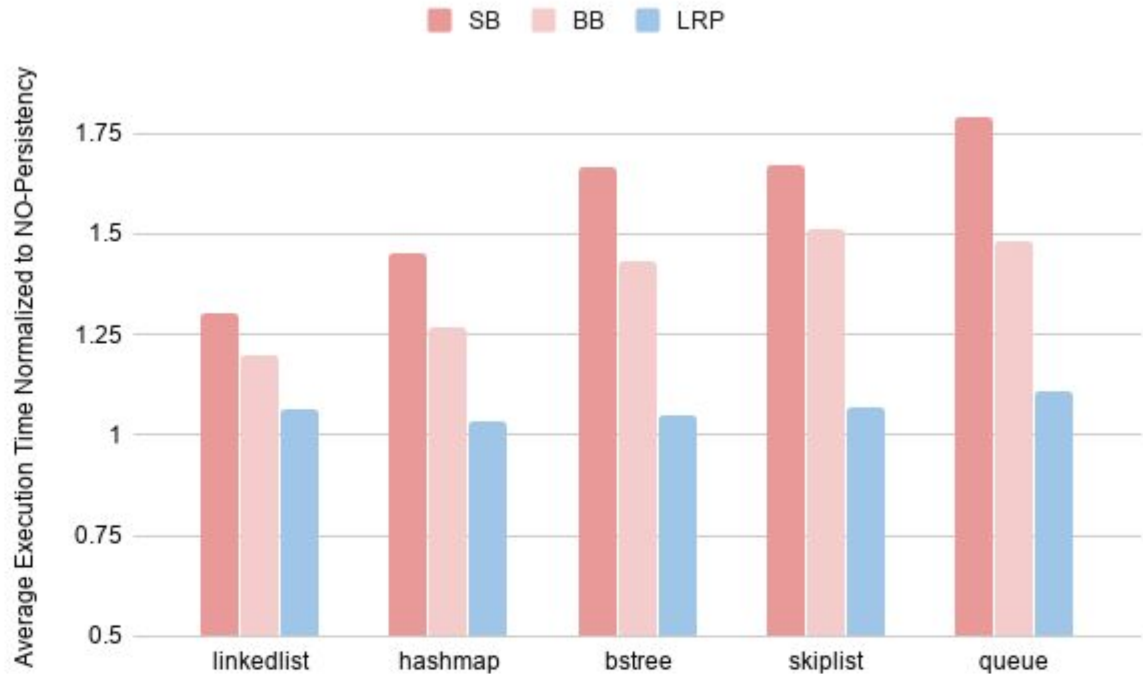
- Upto 71% Improvement
- Average of 52%

LRP vs BB *[Joshi '15]*

- Upto 44% Improvement
- Average of 33%

LRP over NO-Persistency

- Only upto 8% overhead



THE UNIVERSITY of EDINBURGH informatics

Summary

- Crash consistency requires persistency primitives: Ordering or Atomicity?
- Languages should support atomicity but also ordering!
- Why? Log-free data structures can recover without needing atomicity
- State-of-the-art language primitive for ordering ARP is not strong enough for LFDs

Contribution

- Release Persistency a language-level persistency model
- Lazy release persistency (LRP) its microarchitectural implementation
- LRP shows 14%-44% improvement over 5 commonly used LFDs.

THE UNIVERSITY *of* EDINBURGH
informatics

Backup Slides

Execution Time

LRP vs SB

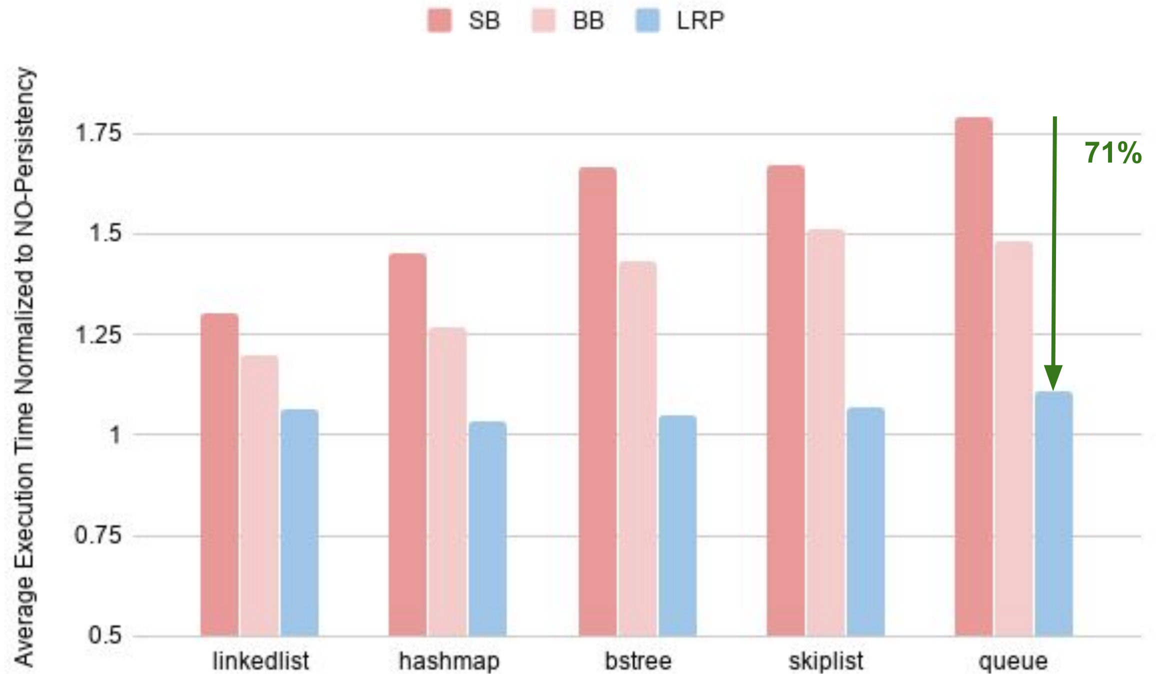
- Upto 71% Improvement
- Average of 52%

LRP vs BB *[Joshi '15]*

- Upto 44% Improvement
- Average of 33%

LRP over NO-Persistency

- Only upto 8% overhead



THE UNIVERSITY of EDINBURGH
informatics

Execution Time

LRP vs SB

- Upto 71% Improvement
- Average of 52%

LRP vs BB *[Joshi '15]*

- Upto 44% Improvement
- Average of 33%

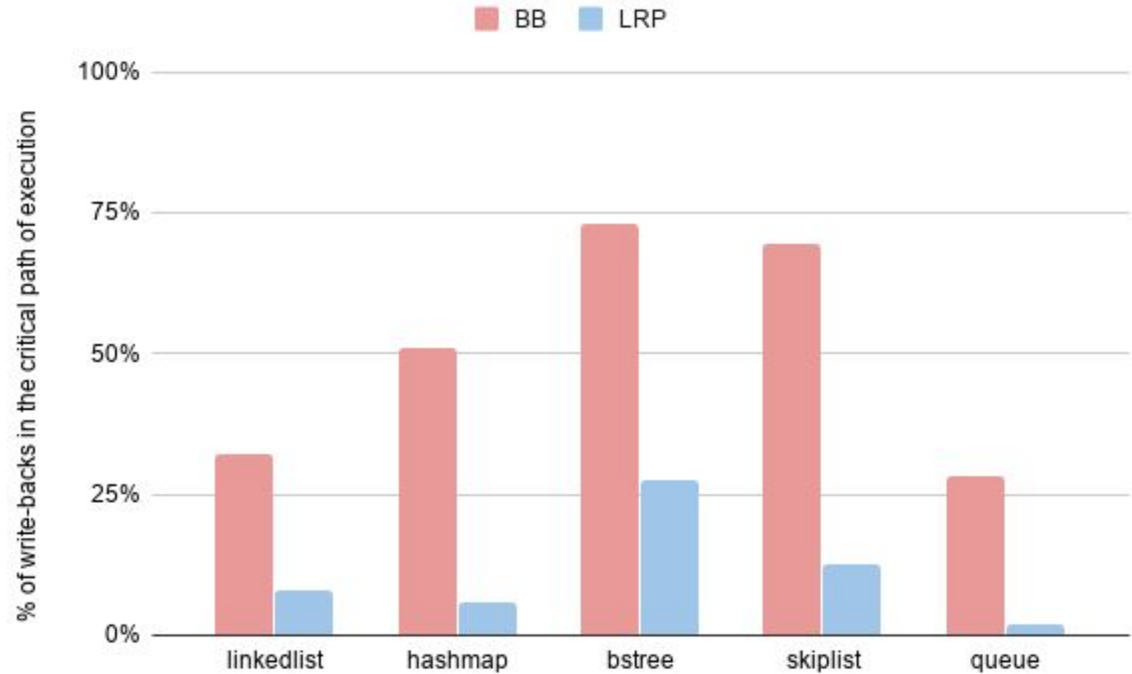
LRP over NO-Persistency

- Only upto 8% overhead

Writebacks

LRP vs BB

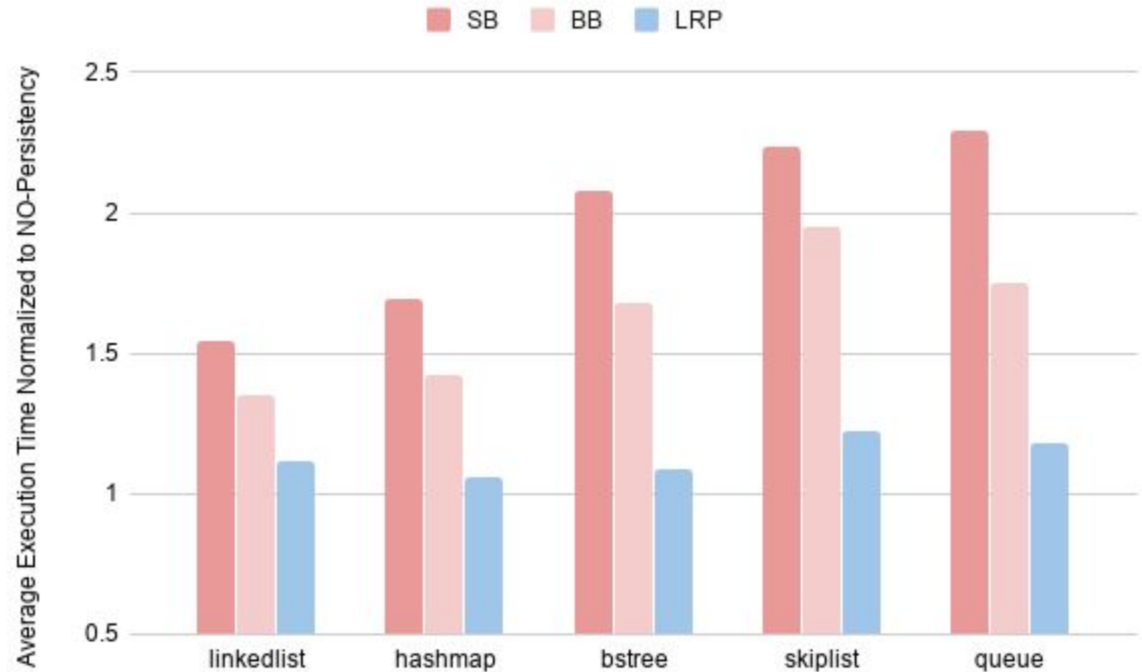
- ~41% improvement in the critical path write-backs.



Uncached Mode

LRP vs BB

- 24%-68% Improvement
- Average 52%



LFD Recovery Guarantees

Buffered Durable Linearizability (BDL)

BDL and DL require real time ordering

Durable Linearizability (DL)

Detectable Execution (DE)

Stronger Guarantees

LFD Recovery Guarantees

LRP

Null Recovery

BDL and DL require real time ordering

Buffered Durable Linearizability (BDL)

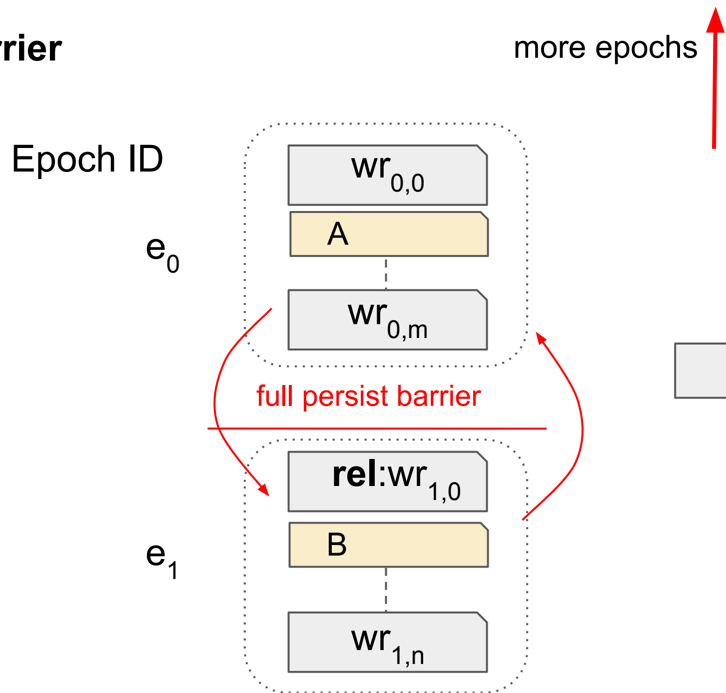
Durable Linearizability (DL)

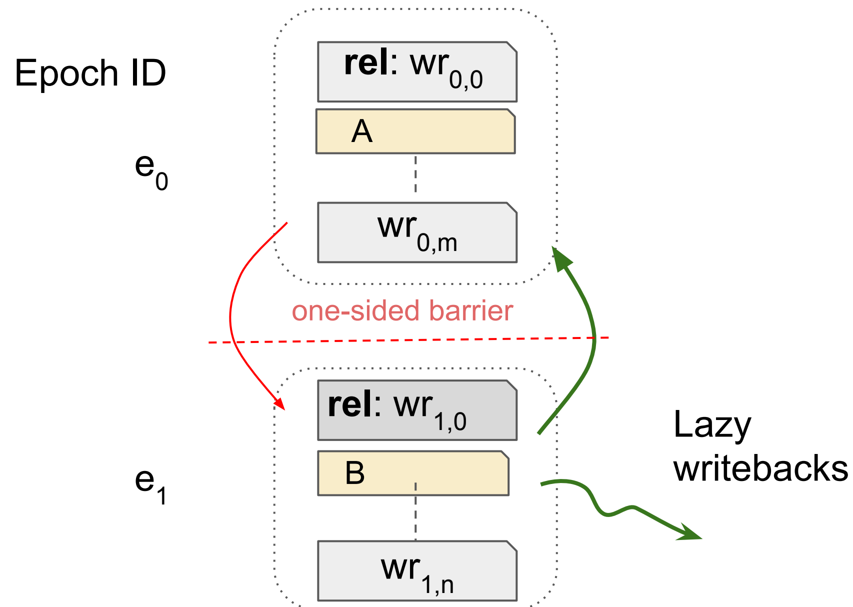
Detectable Execution (DE)

Stronger Guarantees

LRP's null recovery is easy to be extended into stronger guarantees without real time ordering

One-Sided Release Barrier





- No coalescing and reordering across barriers
- Flushing data **eagerly** (*all previous epochs*)

- Coalescing and reordering across barriers
- Flushing data **lazily**

Buffered Epoch Persistency

Buffered Epoch Persistency (BEP):

~~Rule 1: writes from different epochs can be buffered.~~

Rule 2: writes persist in their epoch order.

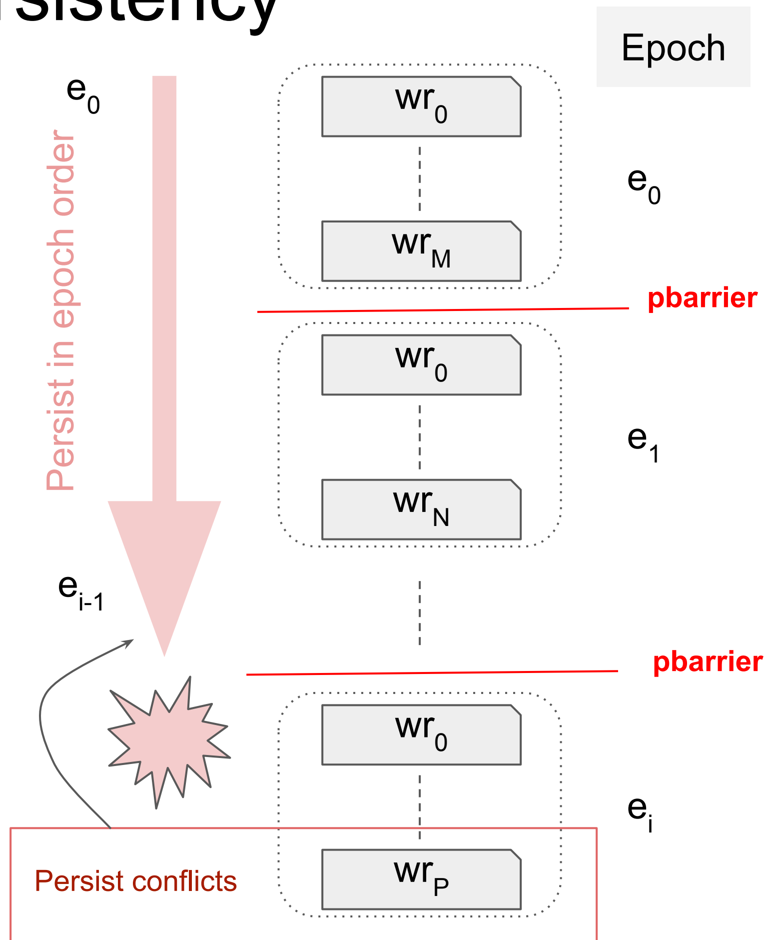
Persist Barriers have strong semantics:

- ➢ No reordering across barriers
- ➢ Persistency overhead is significant!

Implementation artifacts:

- ➢ Cache-line eviction/writeback conflicts
- ➢ Writing to the same cacheline conflicts

Eagerly flushing cache-lines in the critical path



Release Persistency

Epoch

$wr_{0,0}$

A

$wr_{0,m}$

e_0

Full pbarrier

rel: $wr_{1,0}$

B

$wr_{1,n}$

e_1

BEP

Epoch

rel: $wr_{0,0}$

$wr_{0,m}$

e_0

release pbarrier

rel: $wr_{1,0}$

$wr_{1,n}$

e_1

RP

Epoch

$wr_{0,0}$

A

acq: $wr_{0,m}$

e_0

acquire pbarrier

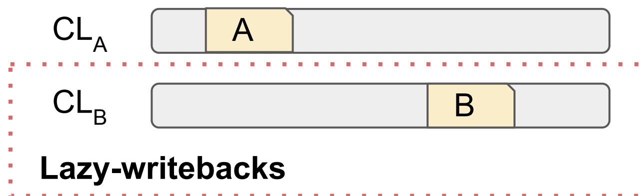
$wr_{1,0}$

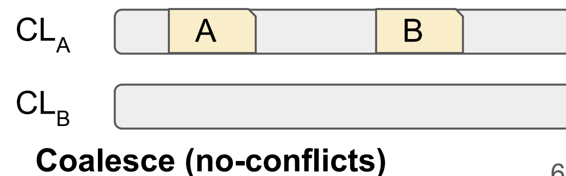
B

$wr_{1,n}$

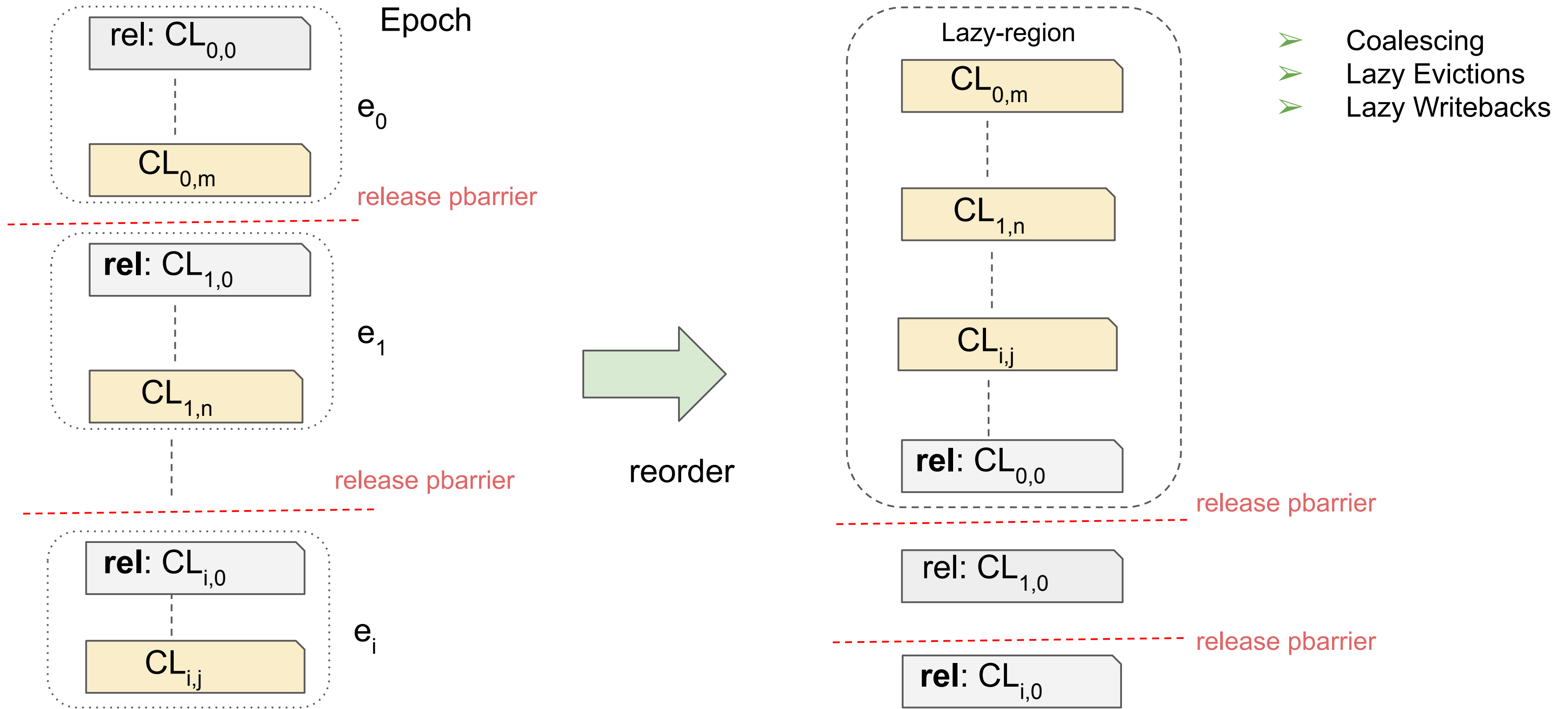
e_1

*RP avoids EOP conflicts



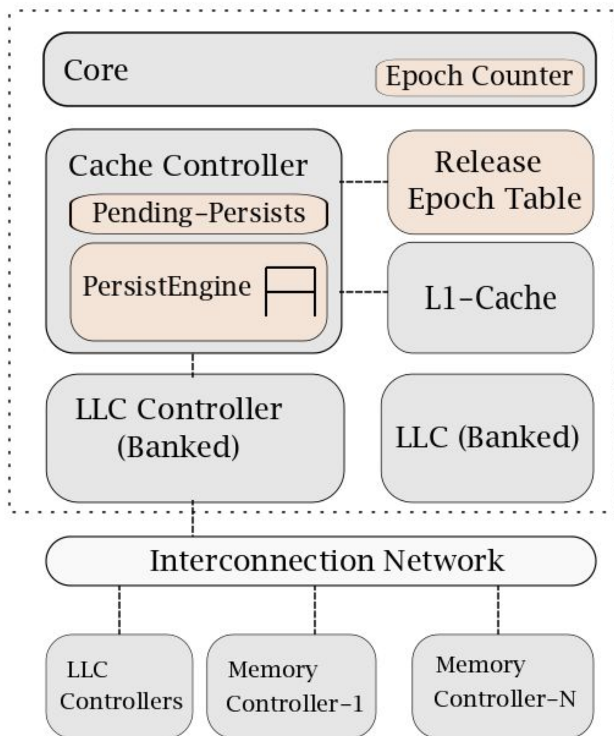


LRP-Examples



*LRP one-sided barriers aggressively reorder writes, persisting stores lazily

Hardware Extensions



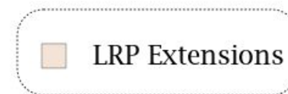
a) Overall Architetcure

min-epoch	R	Tag	Data

b) Cache-line metadata

min-epoch	Address
xx	xxx
xx	xxx
...	...

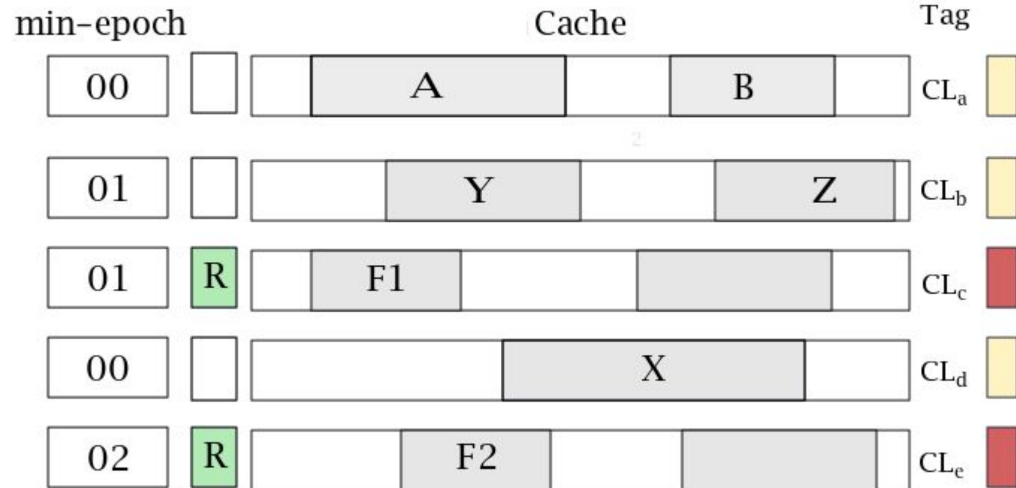
c) Release Epoch Table (RET)



Release Example

Epoch	Execution
00	Write (A)
00	Write (X)
01	Release (F1)
01	Write (B)
01	Write (Y)
02	Release (F2)
02	Write(Z)

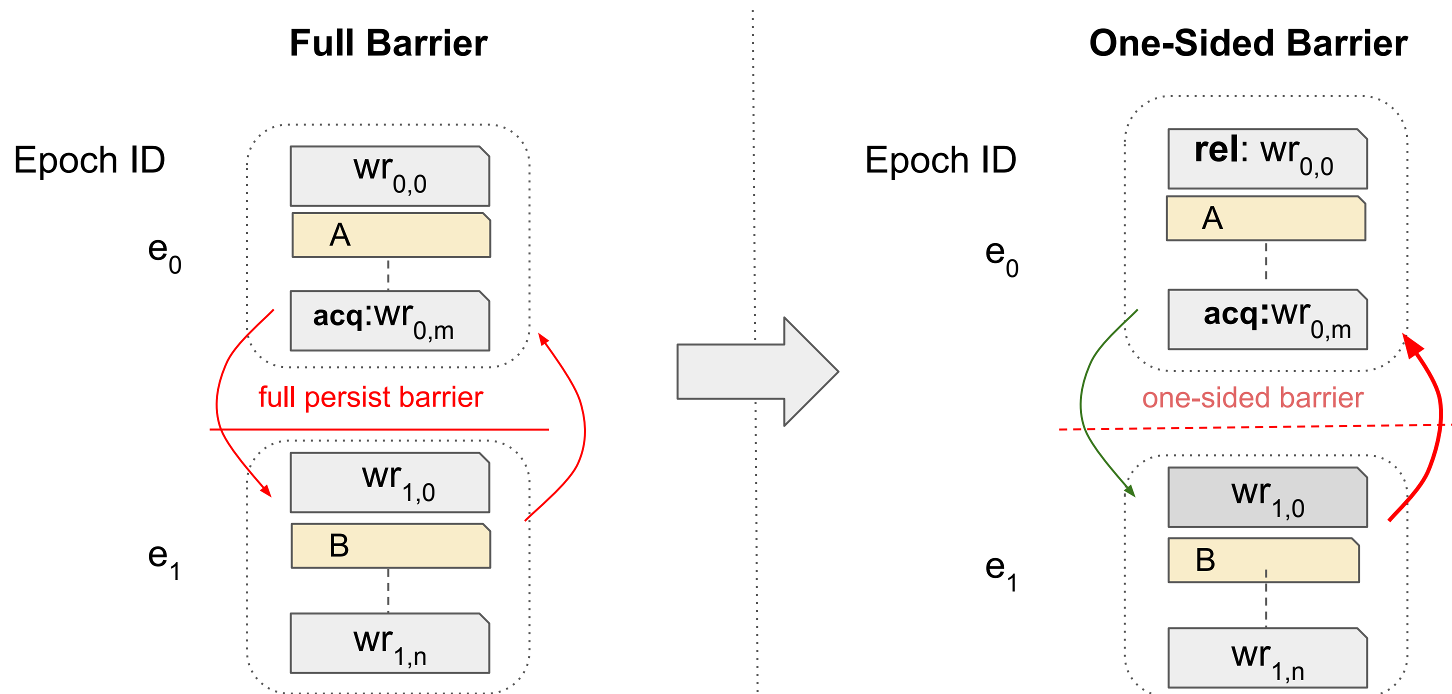
Only–Written
Released
Release Bit



release–epoch	Address
01	CL_c
02	CL_e

Release Epoch Table

One-Sided Acquire Barrier



- No coalescing and reordering across barriers
- Flushing data eagerly (***all** previous epochs*)

- Coalescing and reordering across barriers
- Flushing data lazily

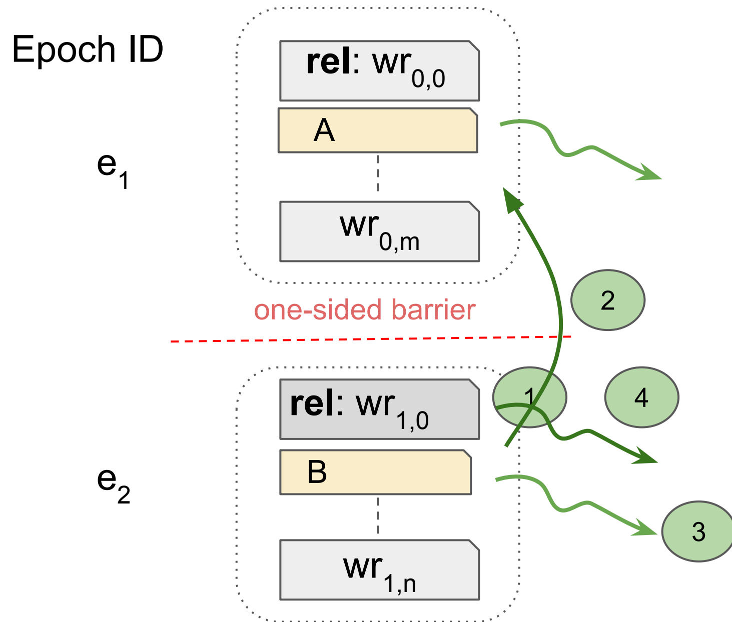
One-Sided Acquire Barrier

- Acquire can be enforced very easily than the Release barriers.
- If a read is annotated with an Acquire, no additional action is required (can easily be enforced with the min epoch Id mentioned in the Release).
- If a RMW is annotated with Acquire, *write* inside RMW is persisted first.
- Regular Write can never be annotated with acquire.

All Cases Handled

- **Wr → Wr** : Write after write is no issue.
- **Rel → Wr** : Write after release can be coalesced. unlimitedly.
- **Wr → Rel** : Release after write can be coalesces only if min epoch id is equal to rel epoch id -1.
- **Rel → Rel** : Release after release to a cacheline can be coalesced only if they are ***consecutive*** releases (epoch id+1). RET table is used, but can be optimized with/without the RET table as well.
- Different consistency guarantees: **RC-SC** or **RC-PC**.

LRP One-Sided Release Barrier



Key Points

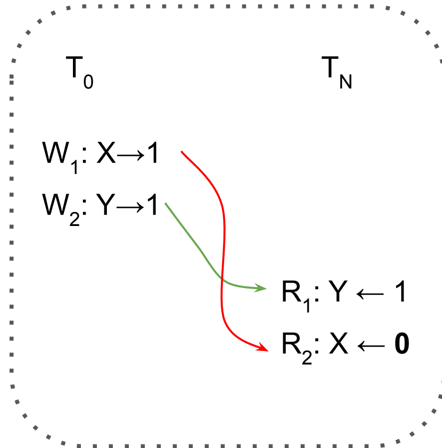
1. Release barrier is flagged using a bit
2. Cache-lines store *only* **min** epoch Id
3. "*Pending-persist*" counter to track write-backs
4. Persist engine to enforce the RC semantics

*__min__ epoch Id: First epoch that writes the cache-line.

THE UNIVERSITY of EDINBURGH informatics

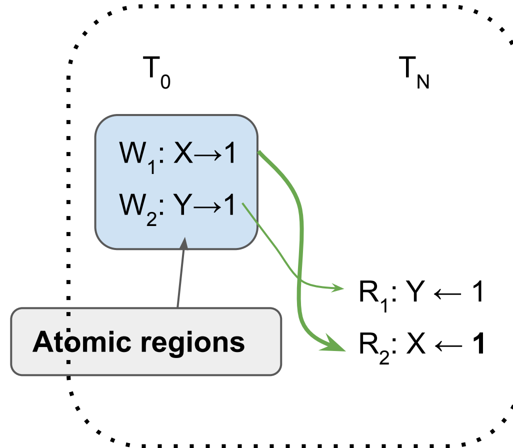
Crash Consistency

- Content of memory after a crash needs to be consistent



Inconsistent!!

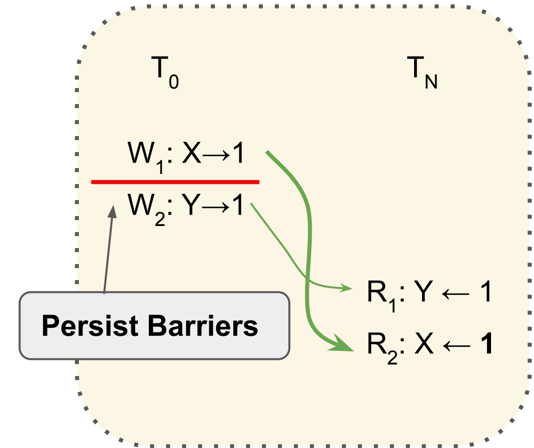
Failure-atomicity
(atomic durability)



Consistent

Transaction **+ Logging**

Ordering



Consistent

clwb **+ sfence**

Crash Consistency

- Content of memory after a crash needs to be consistent