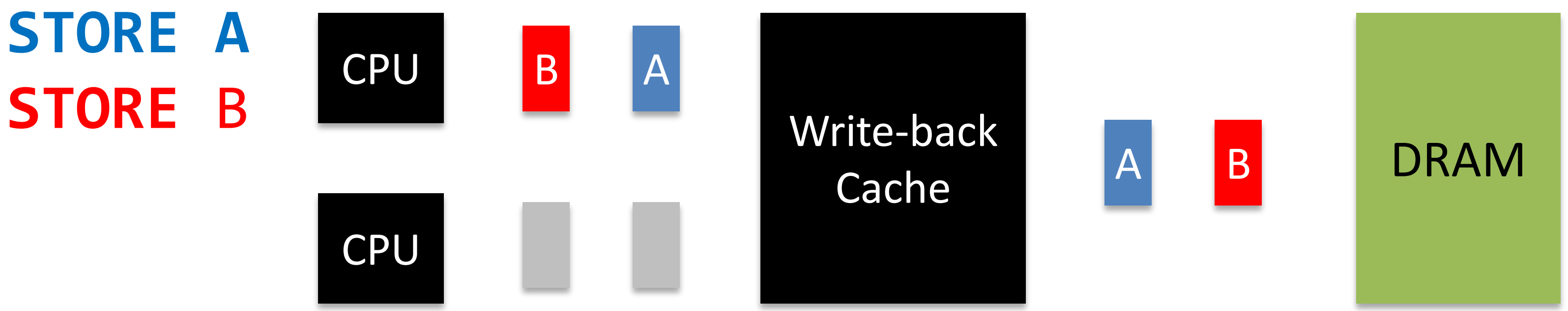
Hardware Support for NVM Programming

Outline

- **Ordering**
- Transactions
- Write endurance

Volatile Memory Ordering

- Write-back caching
  - Improves performance
  - Reorders writes to DRAM

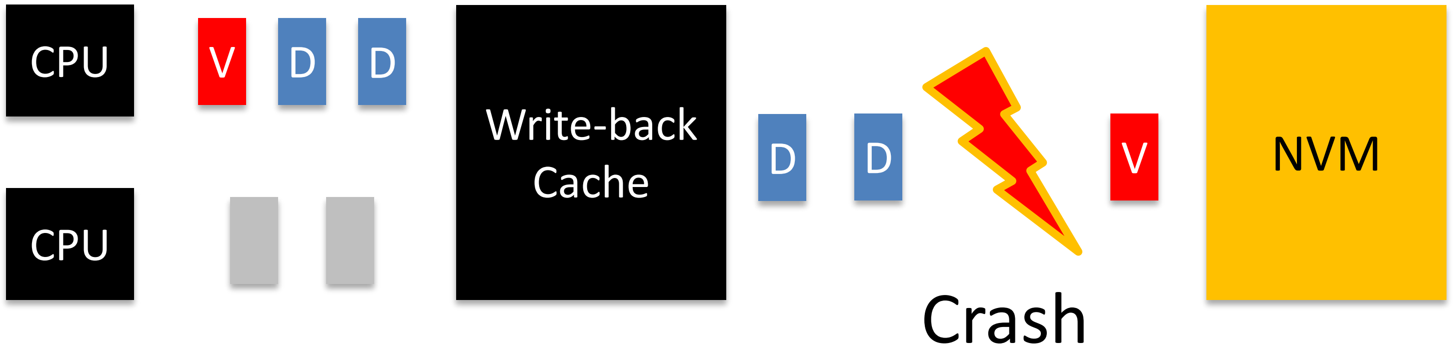


- Reordering to DRAM does not break correctness
- Memory consistency orders stores between CPUs

Persistent Memory (PM) Ordering

- Recovery depends on write ordering

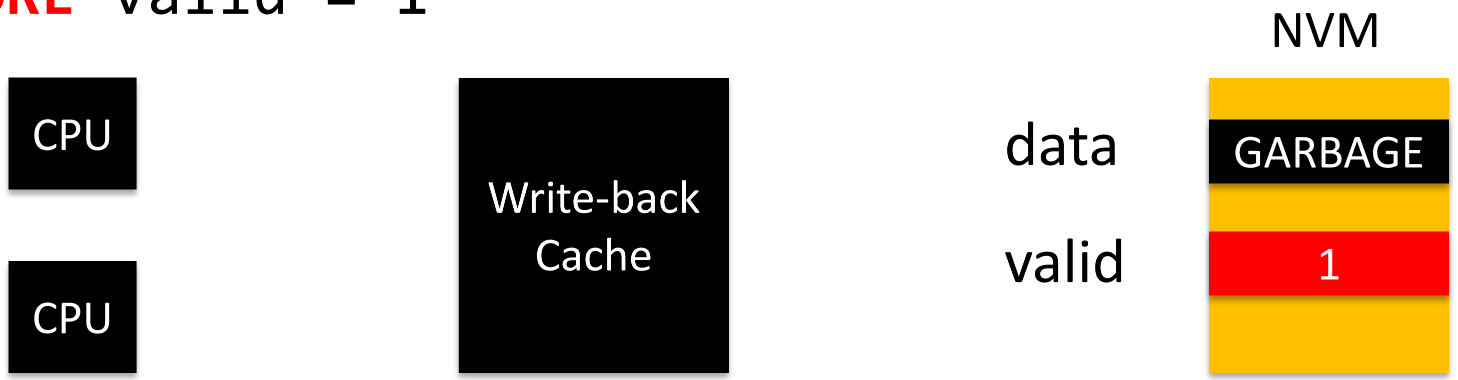
```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
STORE valid = 1
```



Persistent Memory (PM) Ordering

- Recovery depends on write ordering

```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
STORE valid = 1
```



Reordering breaks recovery

Recovery incorrectly considers garbage as valid data

Simple Solutions

- Disable caching
- Write-through caching
- Flush entire cache at commit

Generalizing PM Ordering

```
1: STORE data[0] = 0xFOOD
2: STORE data[1] = 0xBEEF
3: STORE valid = 1
```

Generalizing PM Ordering

```
1: PERSIST data[0]
2: PERSIST data[1]
3: PERSIST valid
```

$$\begin{tikzcd} 1 \arrow[r] & 2 \arrow[r] & 3 \end{tikzcd}$$

Program order implies unnecessary constraints

Need interface to describe necessary constraints

Generalizing PM Ordering

```
1: PERSIST data[0]
2: PERSIST data[1]
3: PERSIST valid
```

$$\begin{tikzcd} 1 \arrow[dr] & \\ & 3 \\ 2 \arrow[ur] & \end{tikzcd}$$

Need interface to expose necessary constraints

Expose persist concurrency; sounds like consistency!

Memory Persistency: Memory Consistency for NVM

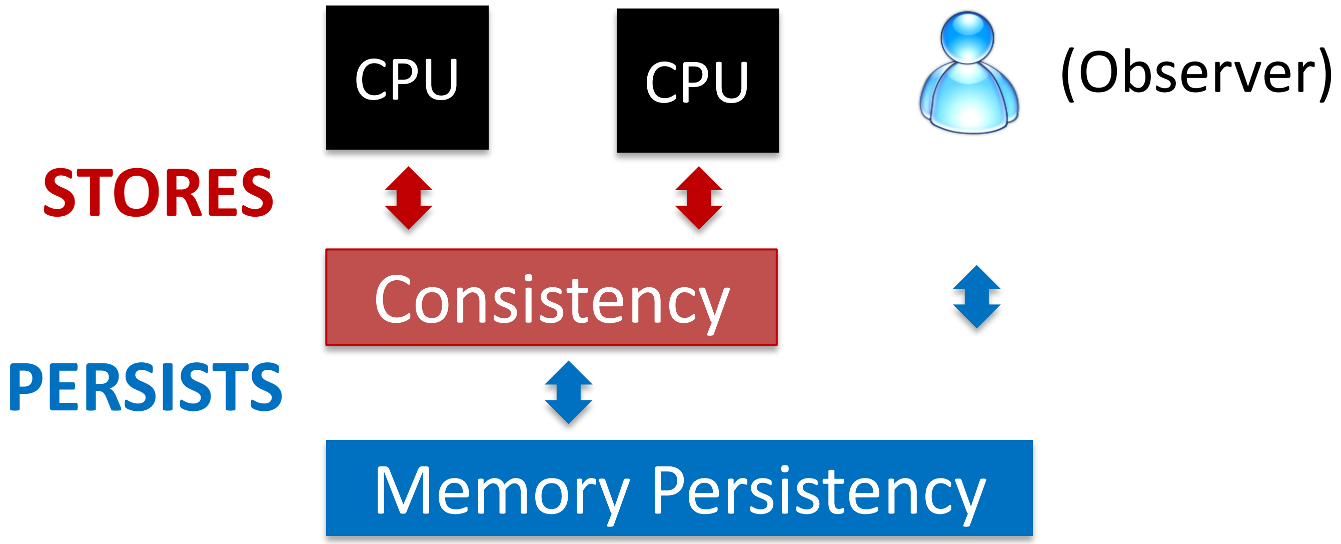
[Pelley, ISCA14]

- Framework to reason about persist order while maximizing concurrency

- Memory consistency
  - Constrains order of loads and stores between CPUs
- Memory persistency
  - Constrains order of writes with respect to failure

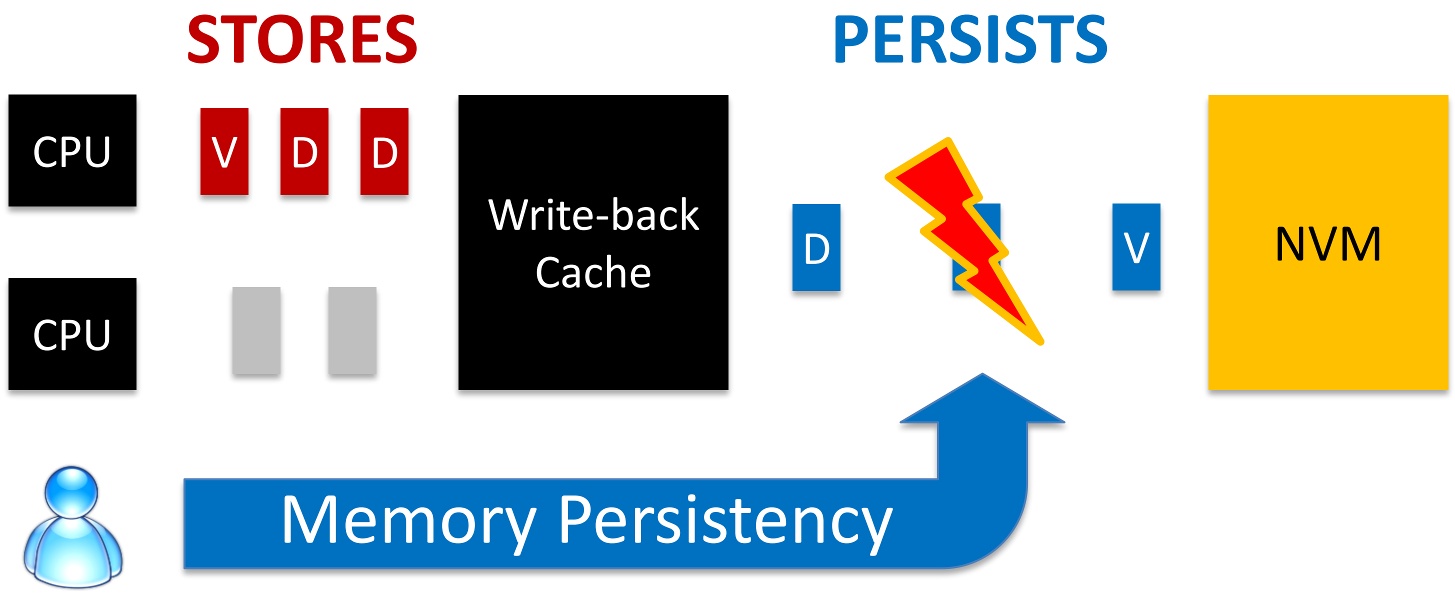
Memory Persistency = Consistency + Recovery Observer

- Abstract failure as recovery observer
  - Observer sees writes to NVM
- Memory persistency
  - Constrains order of writes with respect to **observer**



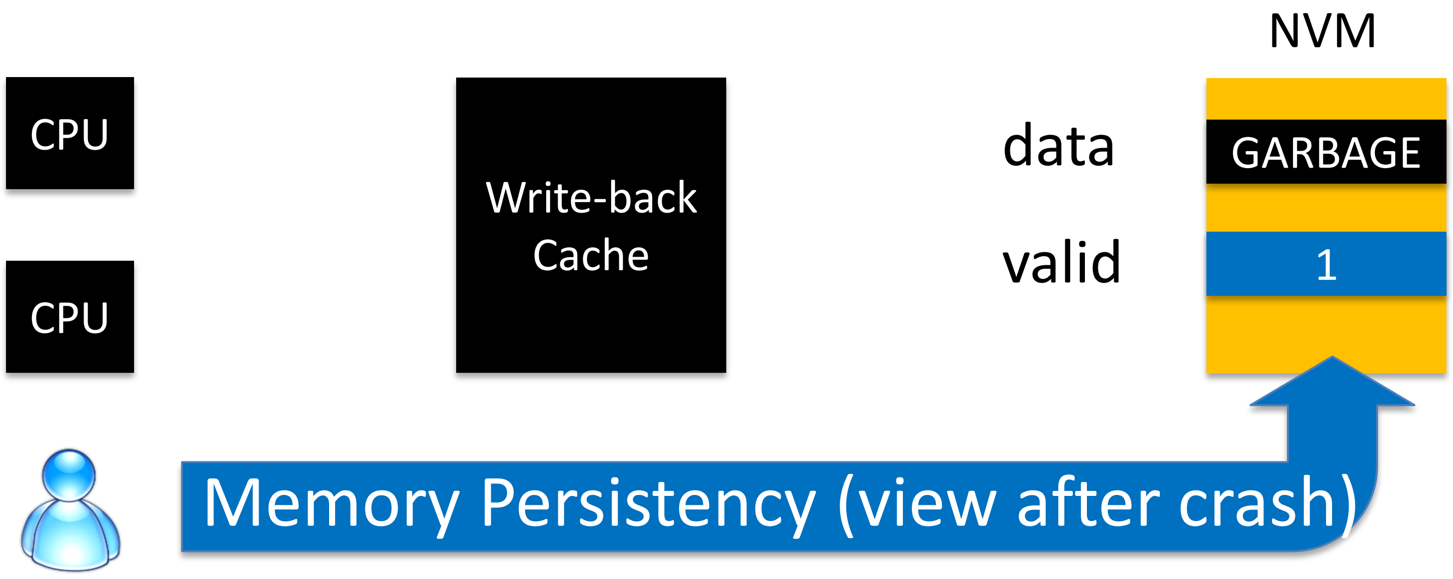
Ordering With Respect to Recovery Observer

```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
STORE valid = 1
```

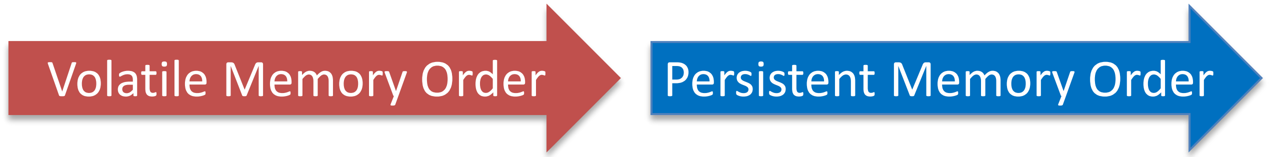


Ordering With Respect to Recovery Observer

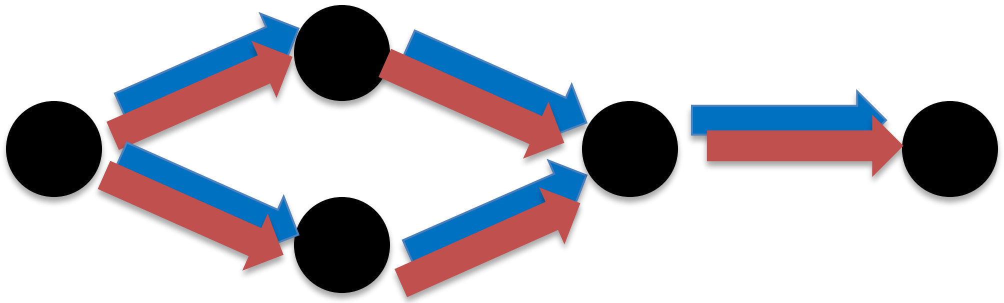
```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
STORE valid = 1
```



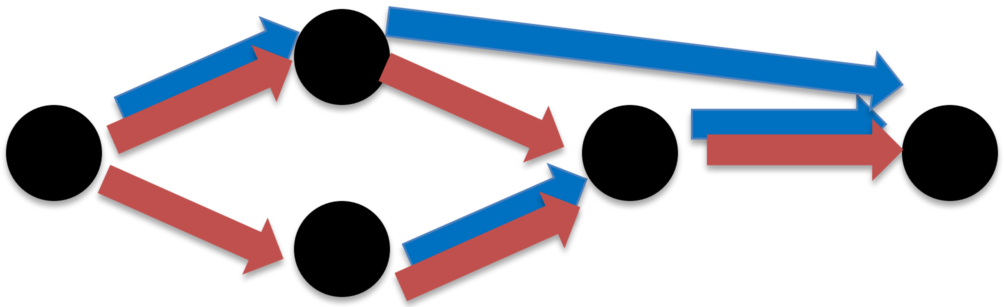
Persistency Design Space

Happens Before: 

- Strict persistency: single memory order



- Relaxed persistency: separate volatile and (new) persistent memory orders



Outline

- **Ordering**
  - **Intel x86 ISA extensions [Intel14]**
  - **BPFS epochs barriers [Condit, SOSP09]**
  - **Strand persistency [Pelley, ISCA14]**
- Transactions
- Write endurance

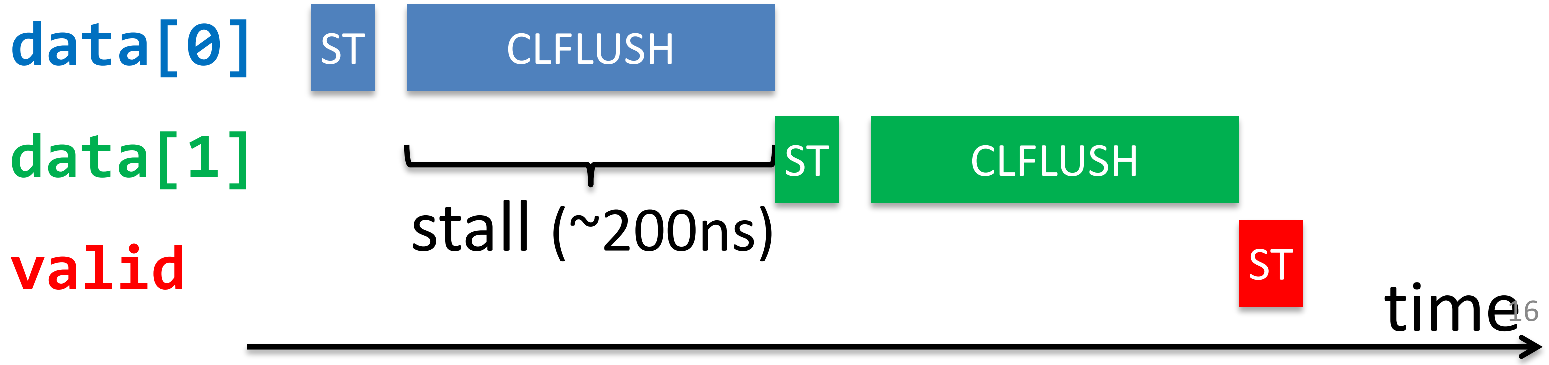
Relax persistency

Ordering with Existing Hardware

- Order writes by flushing cachelines via CLFLUSH

```
STORE data[0] = 0xF00D
STORE data[1] = 0xBEEF
CLFLUSH data[0]
CLFLUSH data[1]
STORE valid = 1
```

- But CLFLUSH:
  - Stalls the CPU pipeline and serializes execution



Ordering with Existing Hardware

- Order writes by flushing cachelines via CLFLUSH

```
STORE data[0] = 0xF00D
STORE data[1] = 0xBEEF
CLFLUSH data[0]
CLFLUSH data[1]
STORE valid = 1
```

- But CLFLUSH:
  - Stalls the CPU pipeline and serializes execution
  - Invalidates the cacheline
  - Only sends data to the memory subsystem – **does not commit** data to NVM

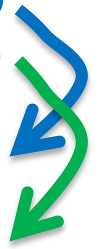
Fixing CLFLUSH: Intel x86 Extensions

- CLFLUSHOPT
- CLWB
- PCOMMIT

CLFLUSHOPT

- Provides unordered version of CLFLUSH
- Supports efficient cache flushing

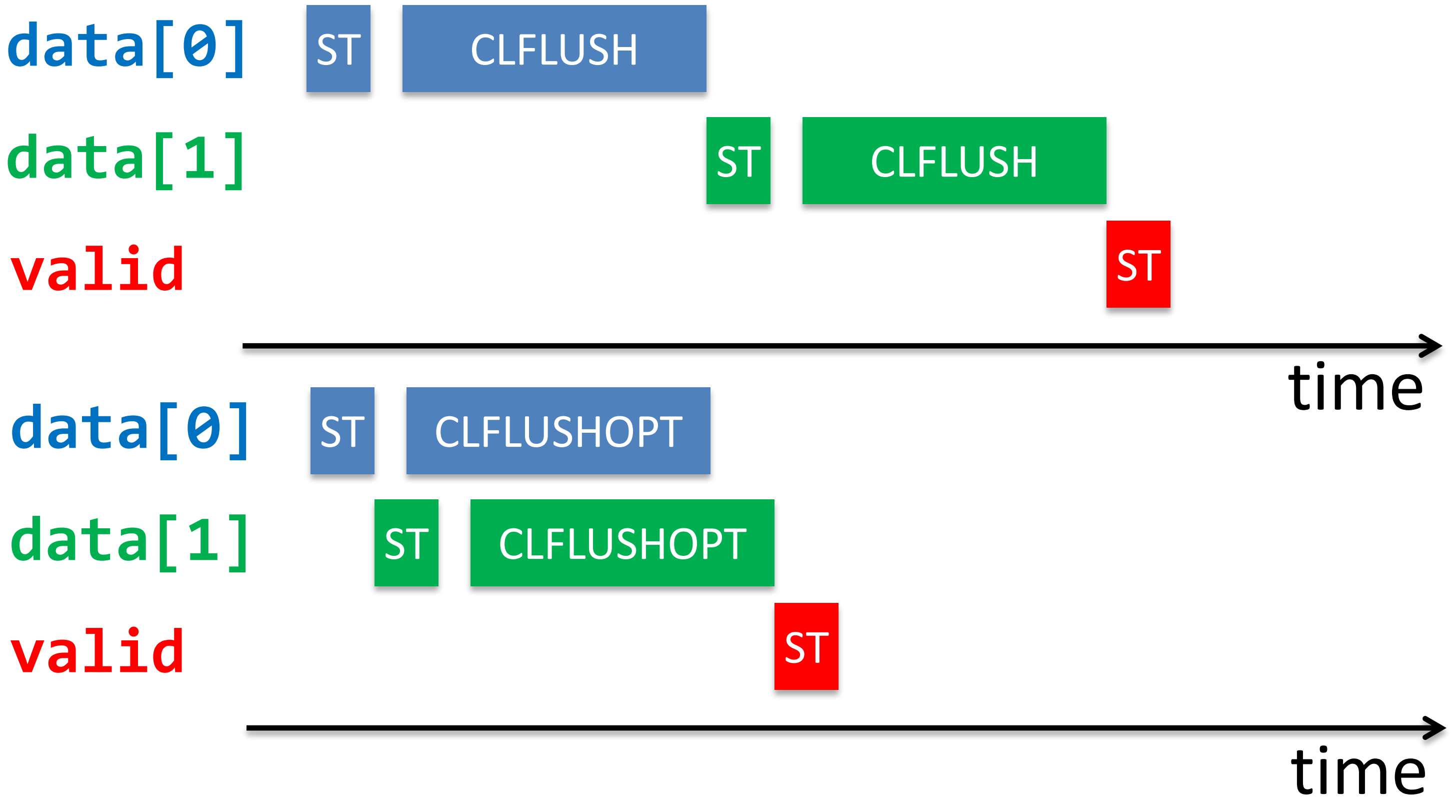
```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
CLFLUSHOPT data[0]
CLFLUSHOPT data[1]
SFENCE // explicit ordering point
STORE valid = 1
```



Implicit orderings

CLFLUSHOPT

- Provides unordered version of CLFLUSH
- Supports efficient cache flushing



CLWB

- Write backs modified data of a cacheline
- **Does not** invalidate the line from the cache
  - Marks the line as non-modified

- Note: Following examples use CLWB

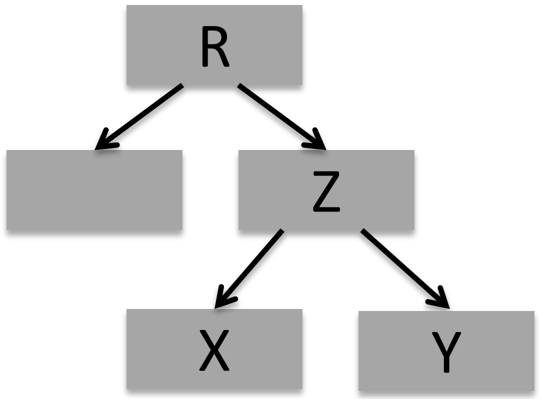
PCOMMIT

- Commits data writes queued in the memory subsystem to NVM

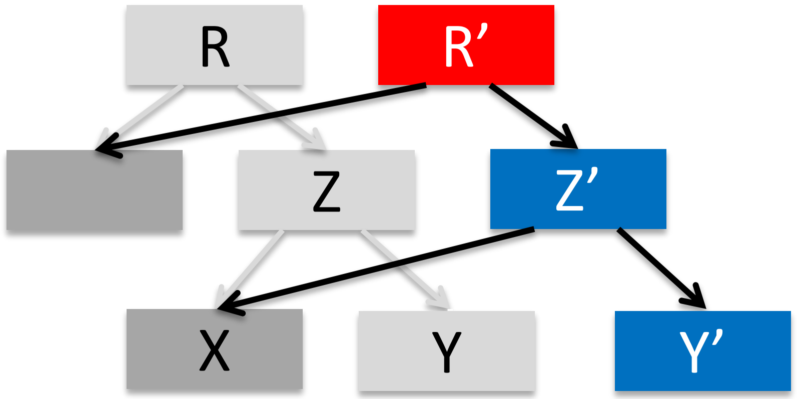
```
STORE data[0] = 0xFOOD
STORE data[1] = 0xBEEF
CLWB data[0]
CLWB data[1]
SFENCE // orders subsequent PCOMMIT
PCOMMIT // commits data[0], data[1]
SFENCE // orders subsequent stores
STORE valid = 1
```

Limitation: PCOMMITs execute serially

Example: Copy on Write

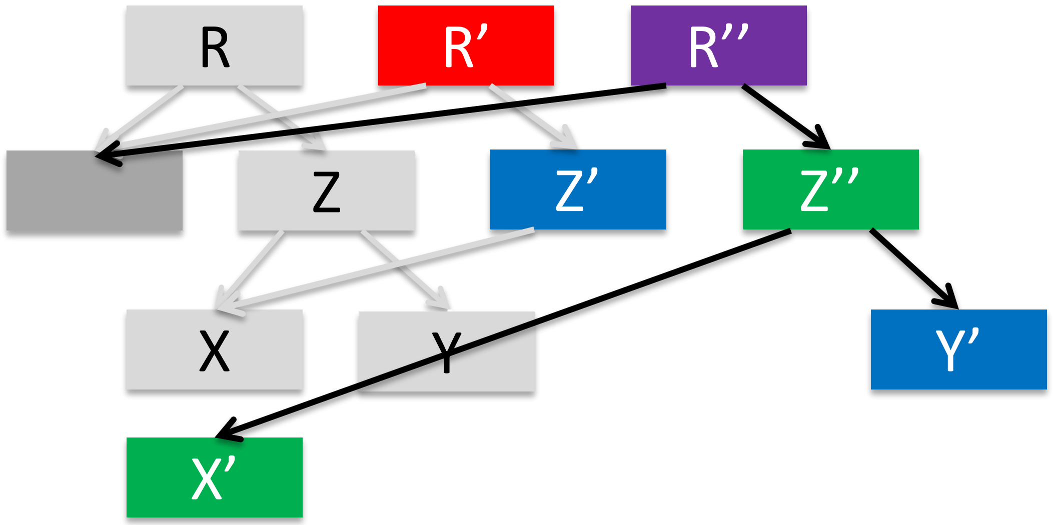


Example: Copy on Write



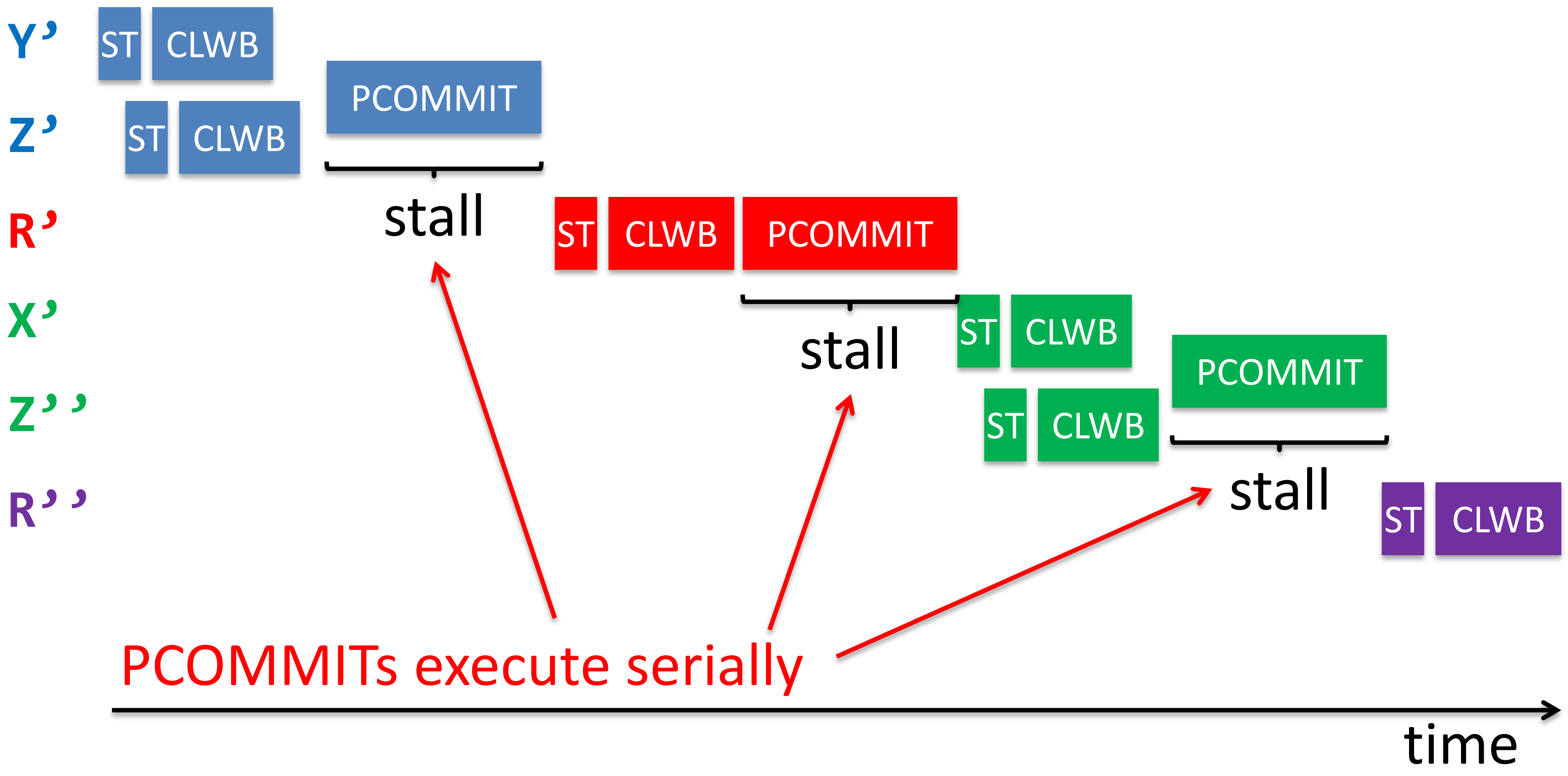
```
STORE Y’
STORE Z’
CLWB Y’
CLWB Z’
PCOMMIT
STORE R’
CLWB R’
```

Example: Copy on Write



```
STORE Y’
STORE Z’
CLWB Y’
CLWB Z’
PCOMMIT
STORE R’
CLWB R’
PCOMMIT
STORE X’
STORE Z’’
CLWB X’
CLWB Z’’
PCOMMIT
STORE R’
CLWB R’
```

Example: Copy on Write – Timeline



Outline

- Ordering
  - Intel x86 ISA extensions [Intel14]
  - **BPFS epochs barriers [SOSP09]**
  - Strand persistency [ISCA14]
- Transactions
- Write endurance

Relax persistency

BPFS Epochs Barriers

[Condit, SOSP09]

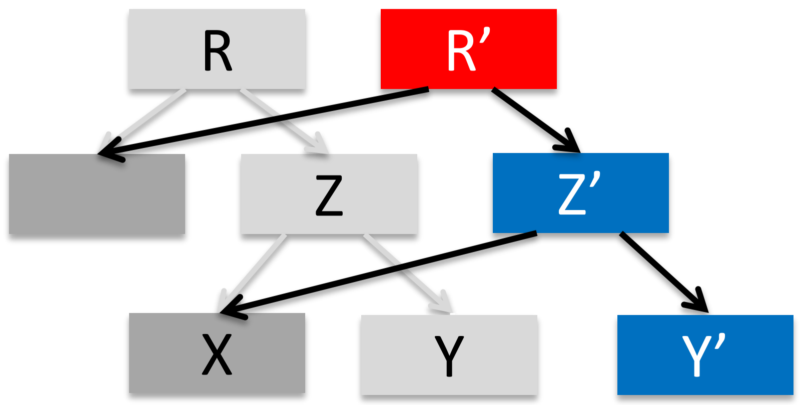
- Barriers separate execution into epochs: sequence of writes to NVM from the same thread

```
Epoch [ STORE …
        STORE …
        EPOCH_BARRIER
Epoch [ STORE …
        EPOCH_BARRIER
Epoch [ STORE …
```

Writes within **same** epoch are **unordered**

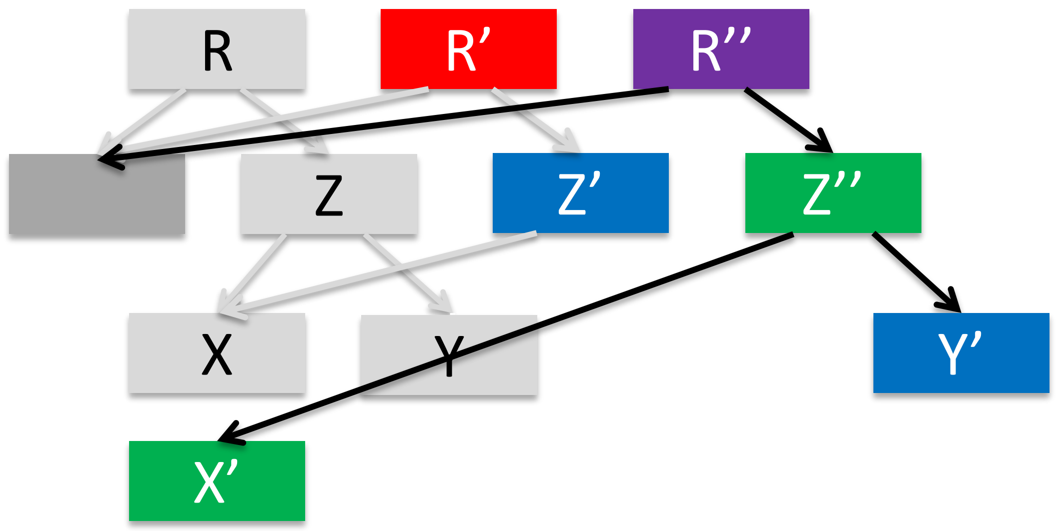
A younger write is issued to NVM **only after all previous** epochs commit

Example: Copy on Write



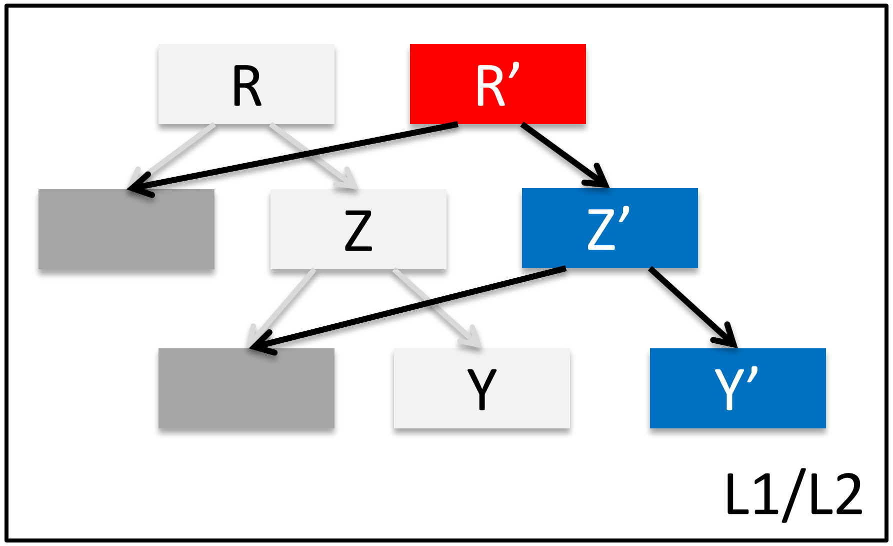
```
STORE Y’  ⌶ Epoch
STORE Z’  ⌶
EPOCH_BARRIER
STORE R’  ⌶ Epoch
```

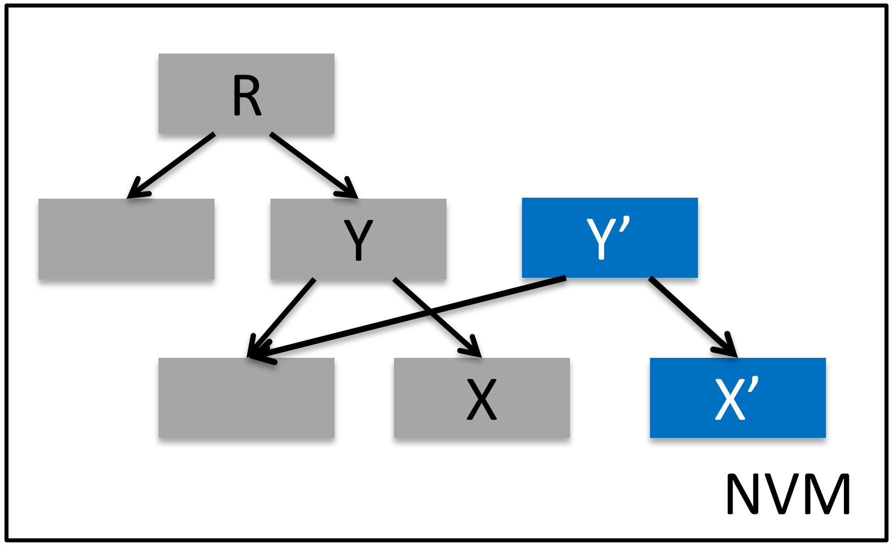
Example: Copy on Write



```
STORE Y’
STORE Z’
EPOCH_BARRIER
STORE R’
EPOCH_BARRIER
STORE X’
STORE Z’’
EPOCH_BARRIER
STORE R’
```

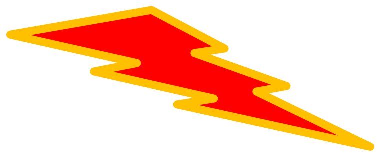
Example: Copy on Write - Failure





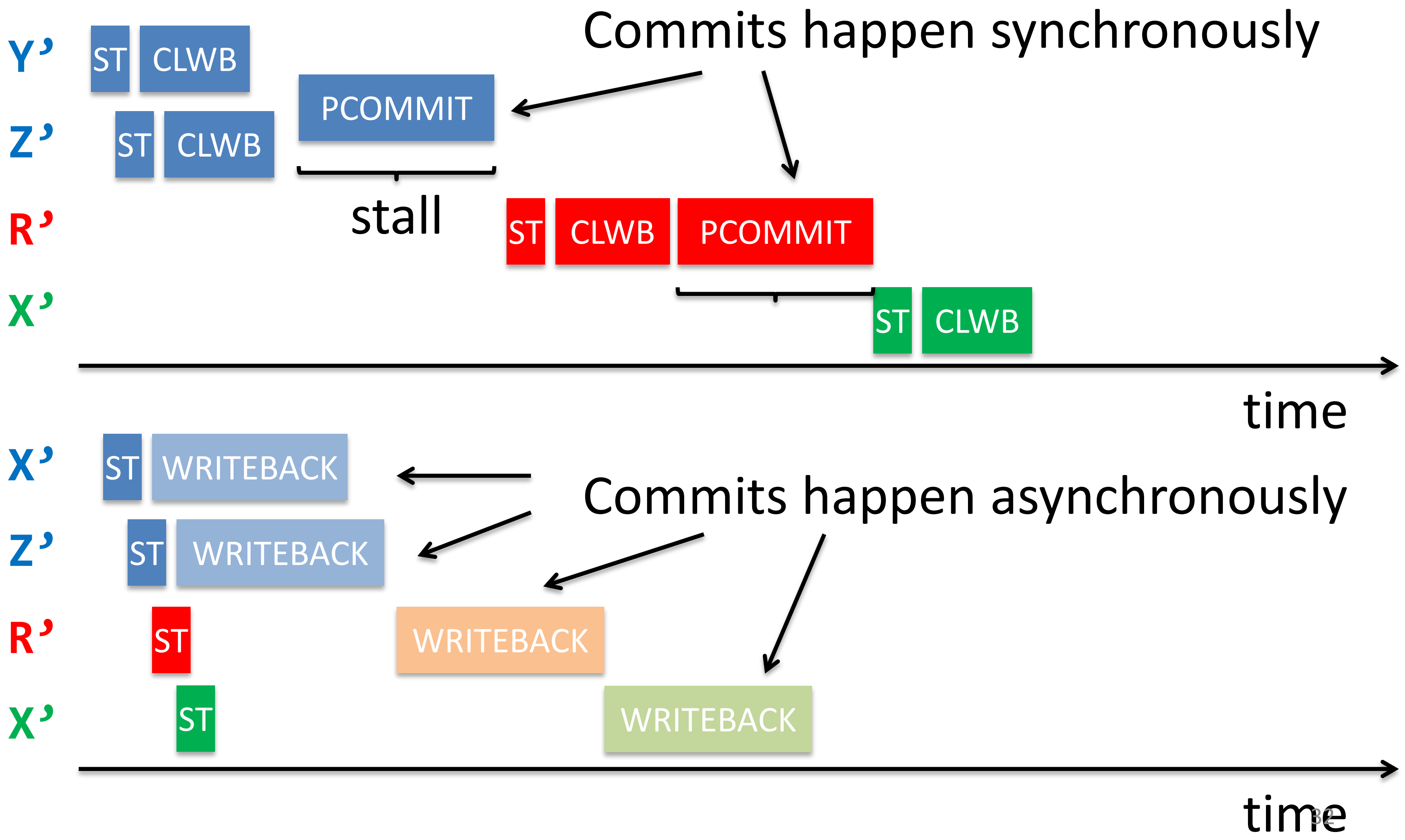
```
STORE Y'
STORE Z'
EPOCH_BARRIER
STORE R'
```

```
WRITEBACK X'
WRITEBACK Y'
```



```
WRITEBACK R'
```

PCOMMIT/CLWB VS Epochs Barriers



BPFS Epochs Barriers: Ordering between threads

- Epochs also capture read-write dependencies between threads

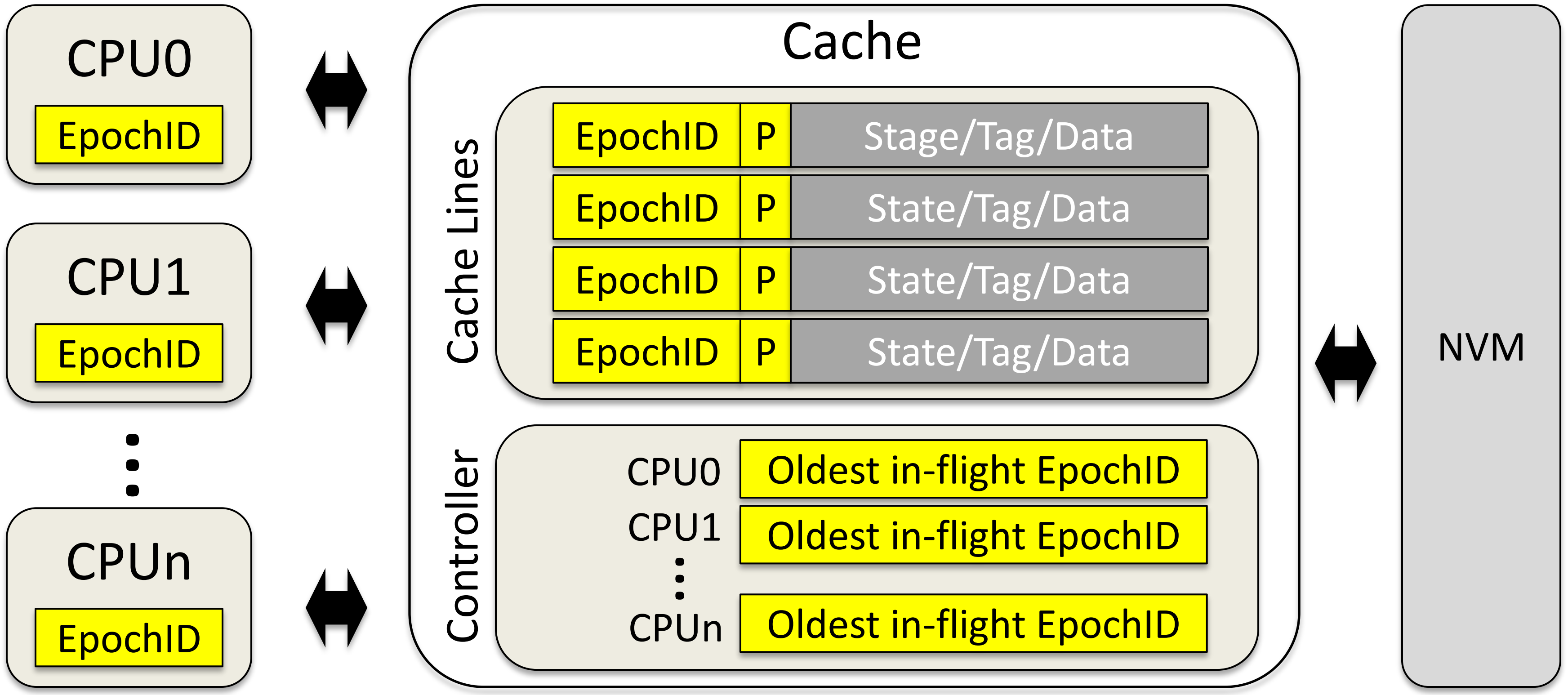
	Thread 0	Thread 1	Recovery Observer
Epoch	STORE …		PERSIST …
	STORE …		PERSIST …
	EPOCH_BARRIER		
Epoch	STORE R		PERSIST R
	EPOCH_BARRIER	LOAD R	
Epoch	STORE …	STORE V	PERSIST V

Memory Consistency makes this dependency visible to thread 1

Missing persists

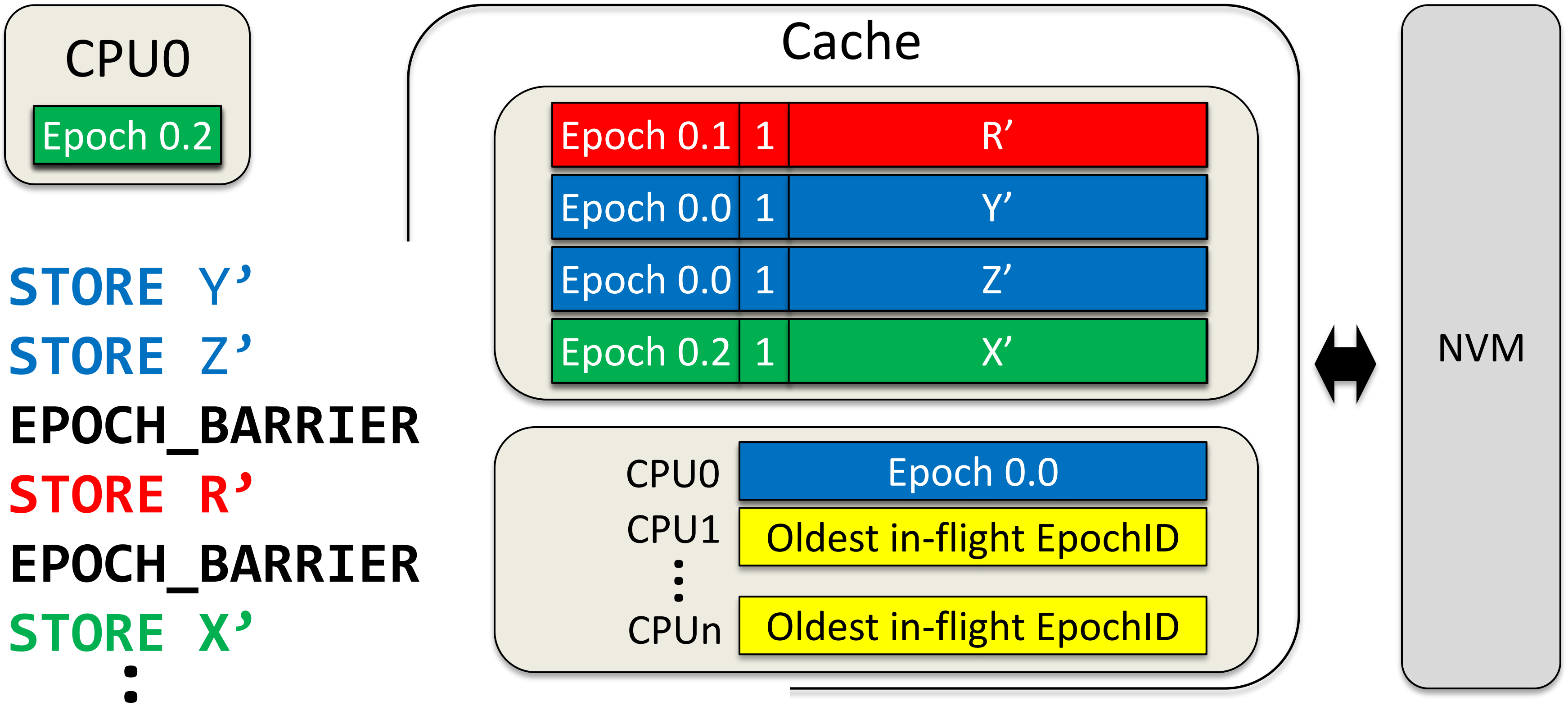
Must make dependency visible to NVM to ensure crash consistency

Epoch Hardware Proposal



- Per-processor epoch ID tags writes
- Cache line stores epoch ID when it is modified
- Cache tracks oldest in-flight epoch per CPU

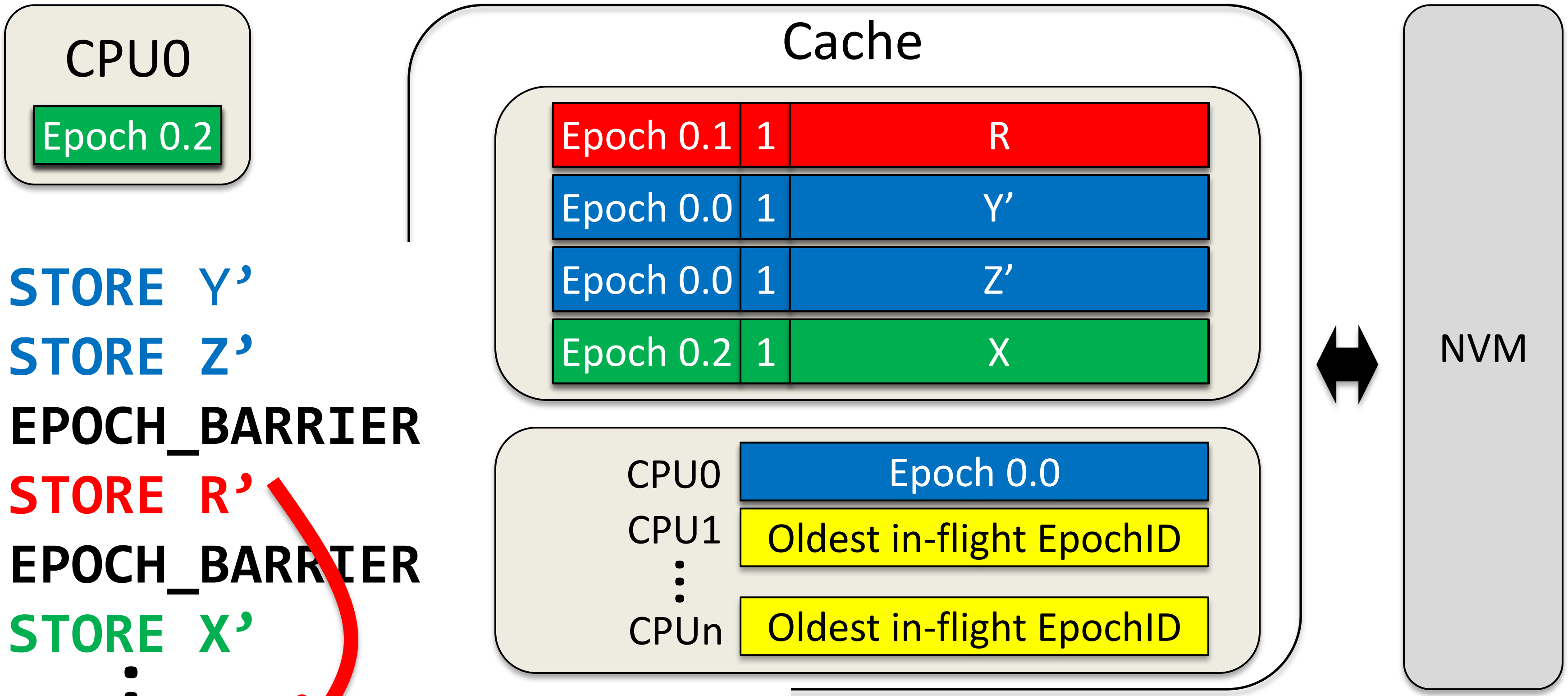
Epoch HW: Ordering Within a Thread
Cascading Writebacks



```
STORE Y’
STORE Z’
EPOCH_BARRIER
STORE R’
EPOCH_BARRIER
STORE X’
     ⋮
EVICT R’
   WRITEBACK Y’
   WRITEBACK Z’
   WRITEBACK R’
```

Epoch 0.1 is not oldest → evict all earlier epochs

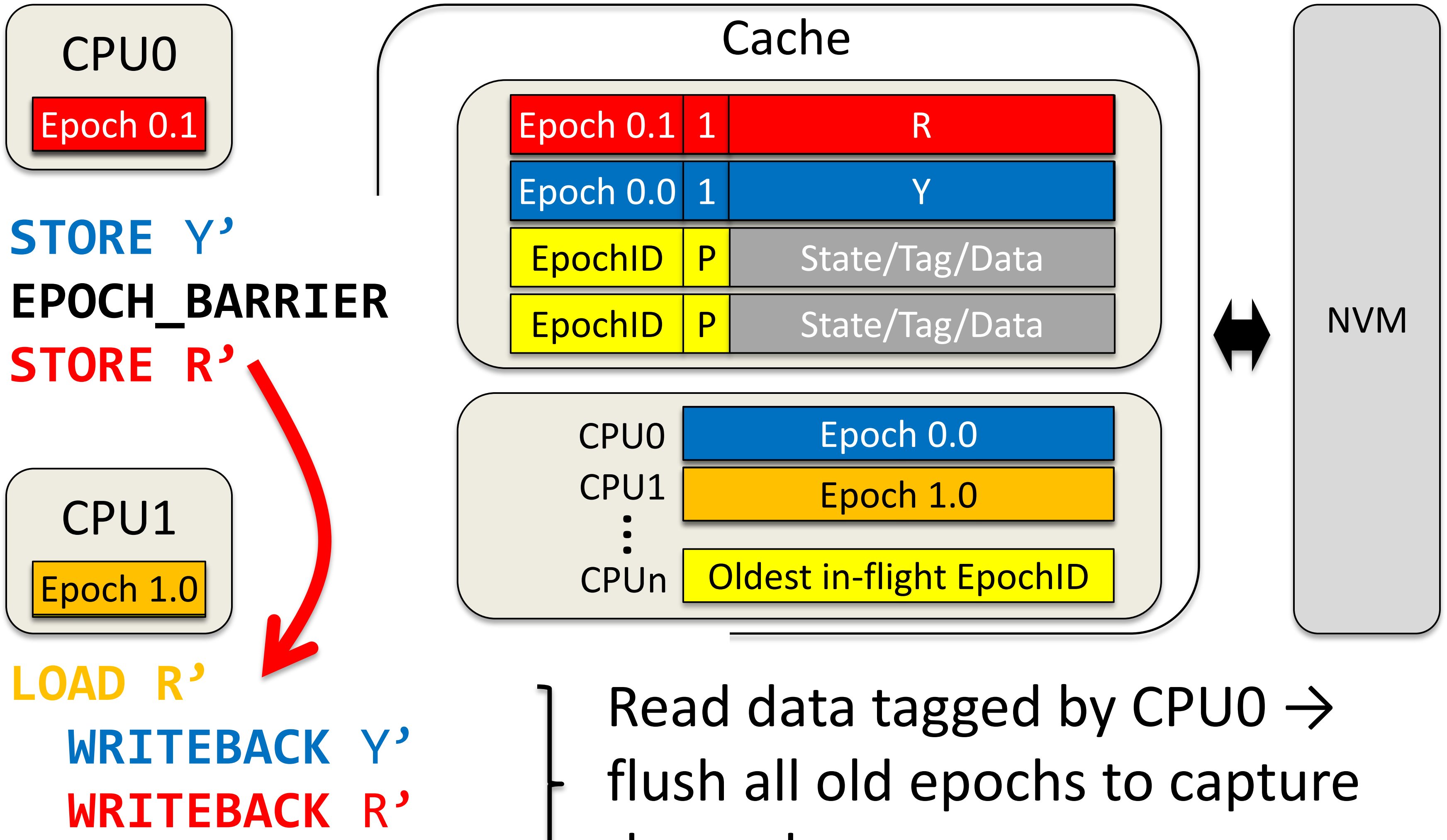
Epoch HW: Ordering Within a Thread Overwrites



```
STORE Y'
STORE Z'
EPOCH_BARRIER
STORE R'
EPOCH_BARRIER
STORE X'
    ⋮
STORE R'
   WRITEBACK Y'
   WRITEBACK Z'
```

} Flush epoch 0.1 (containing Y) and older epochs

Epoch HW: Ordering Between Threads



```
STORE Y'
EPOCH_BARRIER
STORE R'

LOAD R'
   WRITEBACK Y'
   WRITEBACK R'
```

Read data tagged by CPU0 → flush all old epochs to capture dependency

Summary

Ordering primitive	Persists	Commits
CLFLUSH	Serial	N/A
PCOMMIT/CLWB	Parallel	Synchronous
Epochs	Parallel	Asynchronous

Outline

- Ordering
  - Intel x86 ISA extensions
  - BPFS epochs barriers [Condit, SOSP09]
  - **Strand persistency [Pelley, ISCA14]**
- Transactions
- Write endurance

Limitation of Epochs

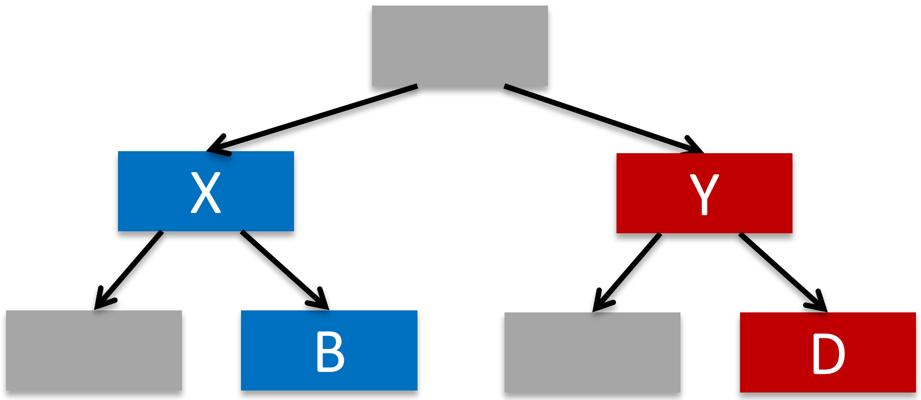
```
seek (fd, 1024, SEEK_SET);
write (fd, data, 128);

            :

seek (fd, 2048, SEEK_SET);
write (fd, data, 128);
```

Non conflicting writes

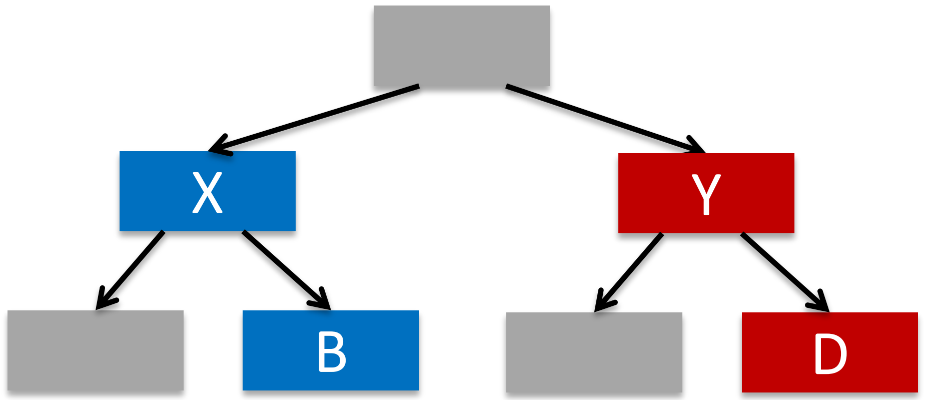
Limitation of Epochs



```
seek (fd, 1024, …);
write (fd, data, 128);

seek (fd, 2048, …);
write (fd, data, 128);
```

Limitation of Epochs

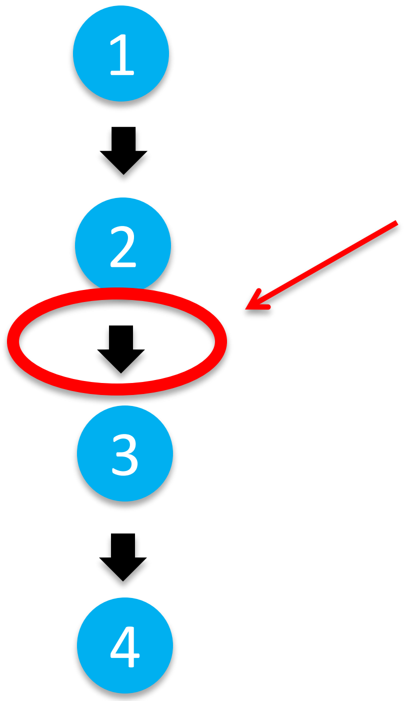


```
STORE B
EPOCH_BARRIER
STORE X

EPOCH_BARRIER
STORE D
EPOCH_BARRIER
STORE Y
```

Limitation of Epochs

```
1:PERSIST B
2:PERSIST X
3:PERSIST D
4:PERSIST Y
```



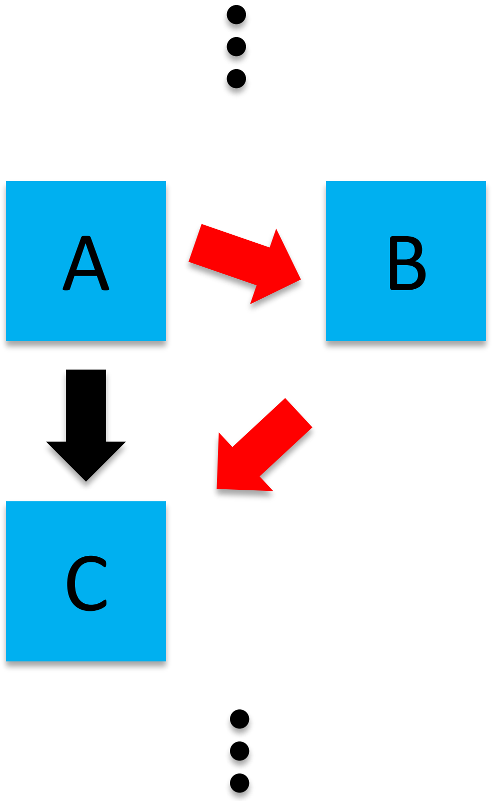
Epoch order implies unnecessary constraint

Can we expose more persist concurrency?

Strand Persistency

- Divide execution into strands
- Each strand is an independent set of persists
  - All strands initially unordered
  - Conflicting accesses establish persist order
- *NewStrand* instruction begins each strand
- Barriers continue to order persists within each strand as in epoch persistency

Strand Persistency: Example



Epoch

```
A
BARRIER
B
C
```

or

```
A
B
BARRIER
C
```

Strand

```
NEWSTRAND
A
BARRIER
C
NEWSTRAND
B
```

Strands remove unnecessary ordering constraints

Strands Expose More Persist Concurrency

```
seek (fd, 1024, …);
write (fd, data, 128);
```

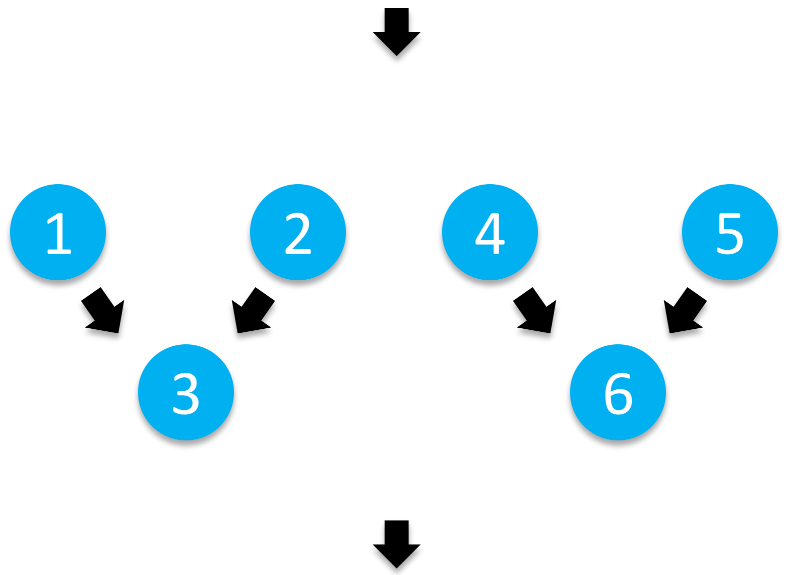
```
NEW_STRAND
STORE B
EPOCH_BARRIER
STORE X
```

```
NEW_STRAND
STORE D
EPOCH_BARRIER
STORE Y
```

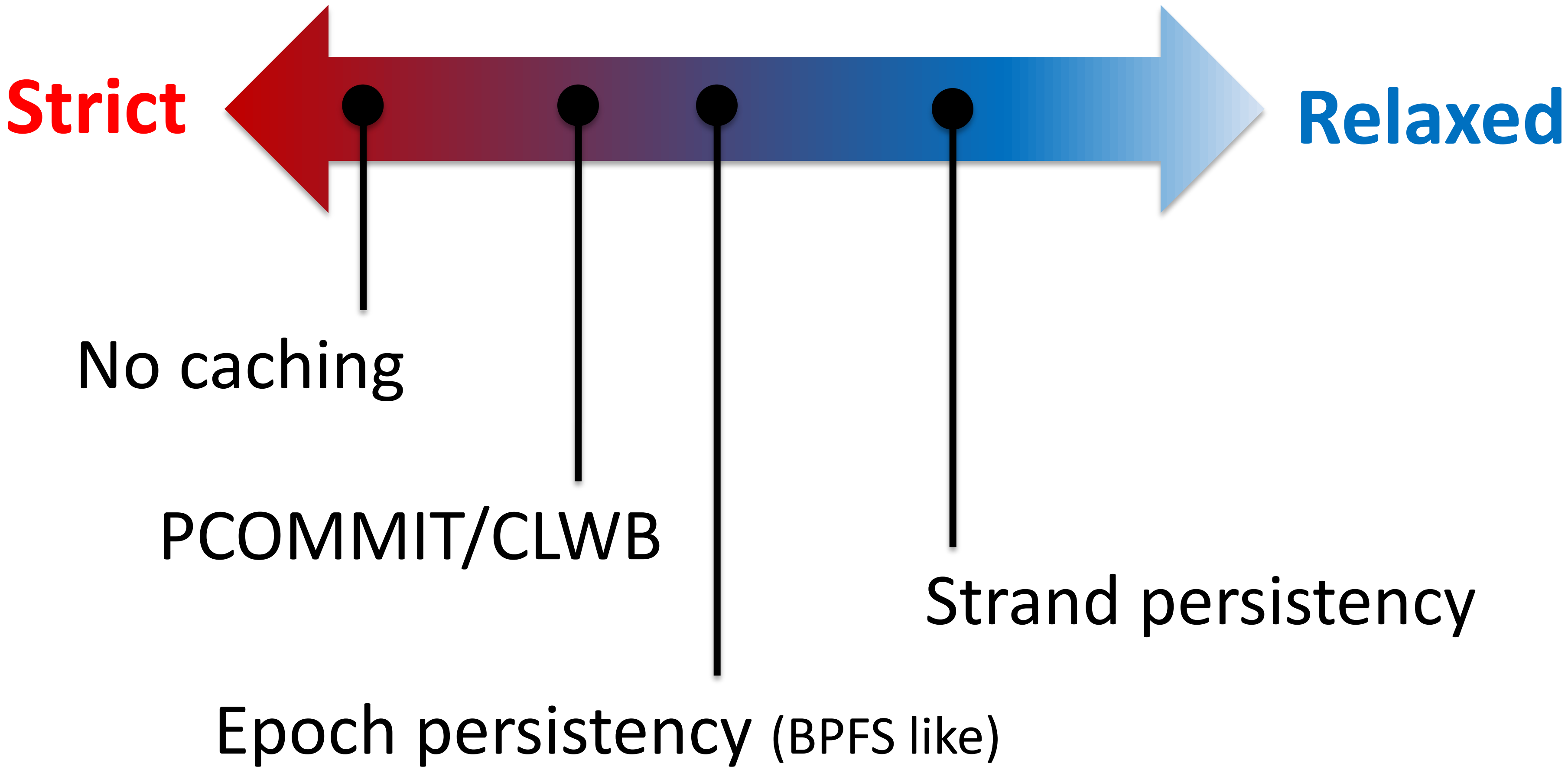
```
seek (fd, 2048, …);
write (fd, data, 128);
```

Strands Expose More Persist Concurrency

```
1:PERSIST B[0]
2:PERSIST B[1]
3:PERSIST X
4:PERSIST D[0]
5:PERSIST D[1]
6:PERSIST Y
```


$$\begin{tikzcd}
1 \arrow[dr] & & 2 \arrow[dl] & 4 \arrow[dr] & & 5 \arrow[dl] \\
& 3 & & & 6 &
\end{tikzcd}$$

Persistency Spectrum



Summary

Ordering primitive	Persists	Commits
CLFLUSH	Serial	N/A
PCOMMIT/CLWB	Parallel	Synchronous
Epochs	Parallel	Asynchronous
Strands	Parallel	Asynchronous + Parallel

Outline

- Ordering
- **Transactions**
  - **Restricted transactional memory [Dulloor, EuroSys14]**
  - **Multiversioned memory hierarchy [Zhao, MICRO13]**
- Write endurance

Software-based Atomicity is Costly

- Atomicity relies on multiple data copies (versions) for recovery
  - Write-ahead logging: write intended updates to a log
  - Copy on write: write updates to new locations

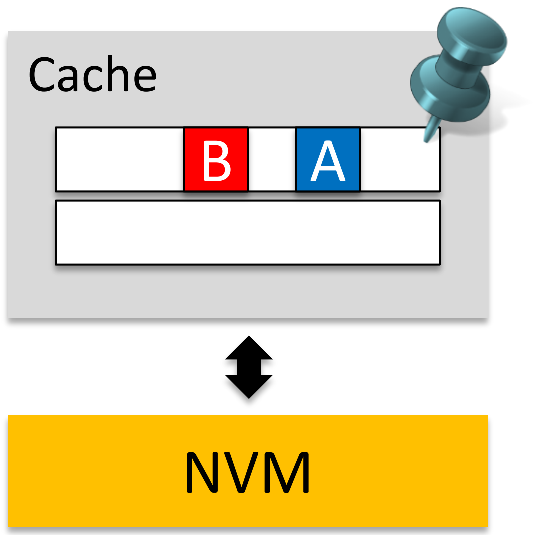
- Software cost
  - Mem-copying for creating multiple data versions
  - Bookkeeping information for maintaining versions

Restricted Transactional Memory (RTM)

- Intel's RTM supports failure-atomic 64-byte cache line writes

```
XBEGIN
STORE  A
STORE  B
XEND
```

A, **B** can be now written back to NVM atomically

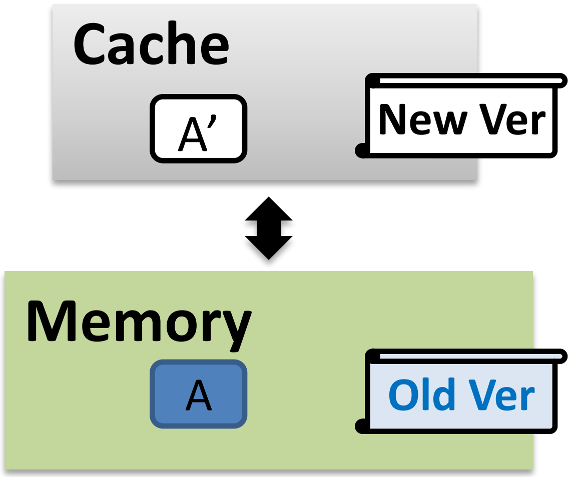


RTM prevents **A**, **B** from leaving cache **before** commit (for isolation)

Existence proof that PM can leverage hardware TM

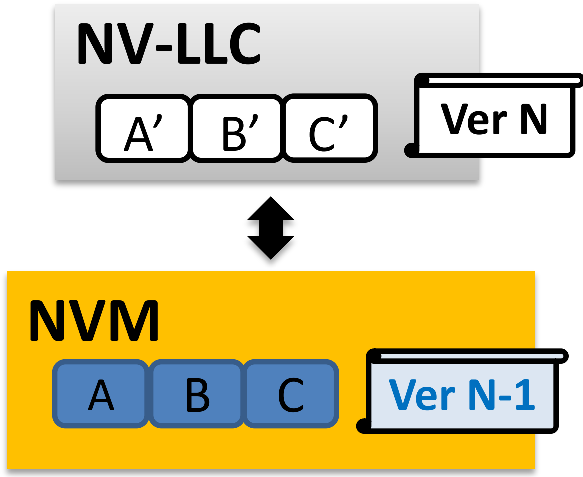
Multiversioning: Leveraging Caching for In-place Updates

- How does a write-back cache work?
  - A processor writes a value
  - Old values remain in lower levels
  - Until the new value gets evicted



Multiversioning: Leveraging Caching for In-place Updates

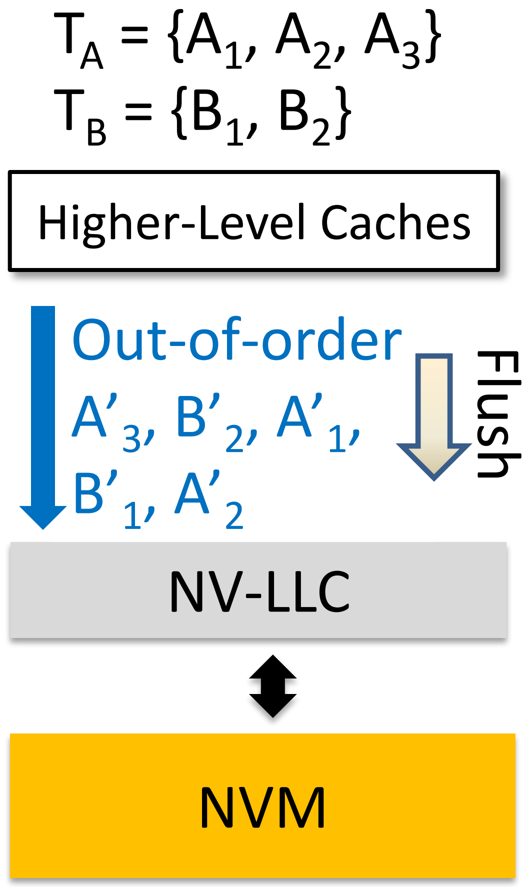
- How does a write-back cache work?
  - A processor writes a value
  - Old values remain in lower levels
  - Until the new value gets evicted
- Insight: multiversioned system by nature
  - Allow in-place updates to directly overwrite original data
  - No need for logging or copy-on-write



A Multiversioned Persistent Memory Hierarchy

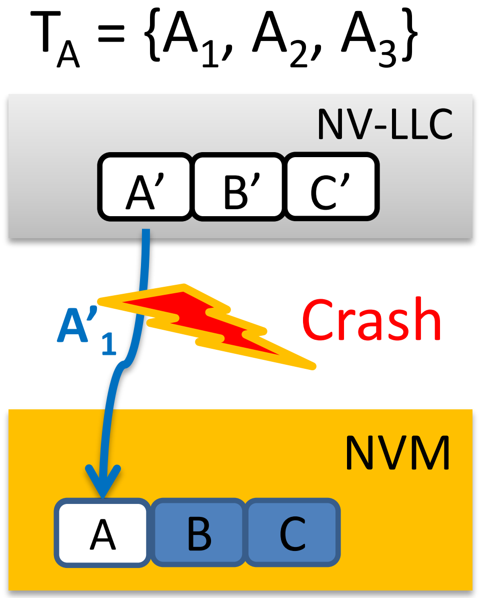
Preserving Write Ordering: Out-of-order Writes + In-order Commits

- Out-of-order writes to NV-LLC
  - NV-LLC remembers the committing state of each cache line
- In-order commits of transactions
  - Example: T_A before T_B
  - T_B will not commit until A_2' arrives in NV-LLC
- Committing a transaction
  - Flush higher-level caches (very fast)
  - Change cache line states in NV-LLC



A Hardware Memory Barrier

- Why
  - Prevents early eviction of uncommitted transactions
  - Avoids violating atomicity



- How
  - Extend replacement policy with transaction-commit info to keep uncommitted transactions in NV-LLC
  - Handle NV-LLC overflows using OS supported CoW

Outline

- Ordering
- Transactions
- **Write endurance**
  - **Start-gap wear leveling [Qureshi, MICRO09]**
  - **Dynamically replicated memory [Ipek, ASPLOS10]**

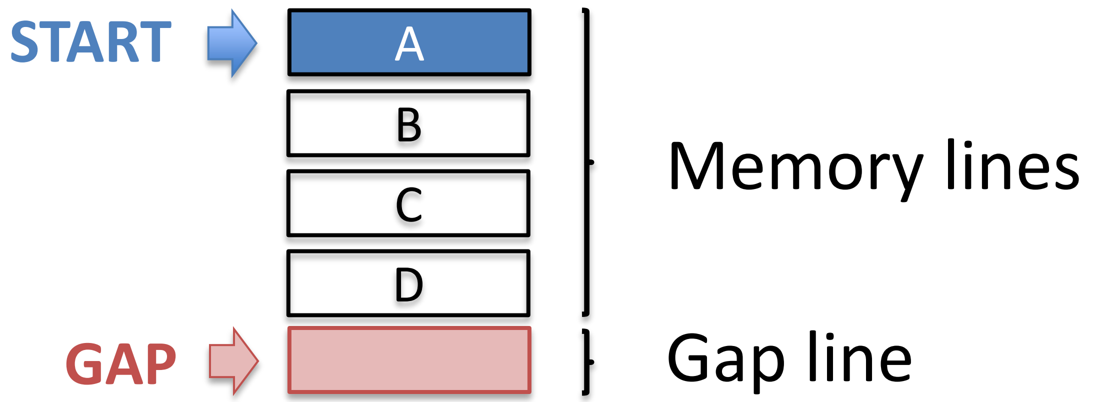
  Note: Mechanisms target NVM-based main memory

Start-Gap Wear Leveling

- Table-based wear leveling is too costly for NVM
  - Storage overheads and indirection latency
- Instead, use algebraic mapping between logical and physical addresses
  - Periodically remap a line to its neighbor

Start-Gap Wear Leveling

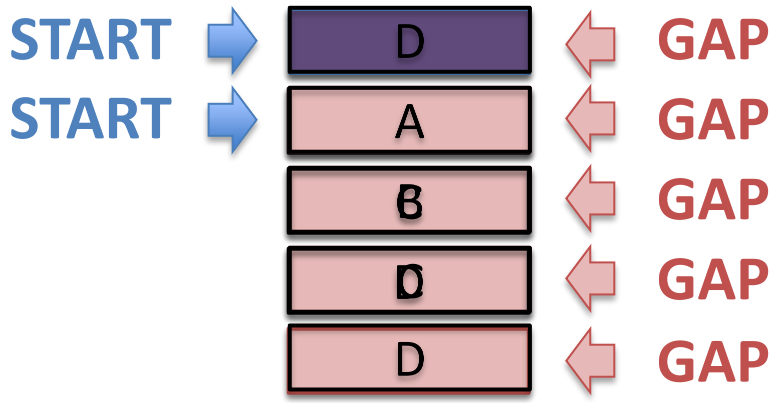
- Table-based wear leveling is too costly for NVM
  - Storage overheads and indirection latency
- Instead, use algebraic mapping between logical and physical addresses
  - Periodically remap a line to its neighbor



Start-Gap Wear Leveling

- Table-based wear leveling is too costly for NVM
  - Storage overheads and indirection latency
- Instead, use algebraic mapping between logical and physical addresses
  - Periodically remap a line to its neighbor

Move *start* every one gap rotation

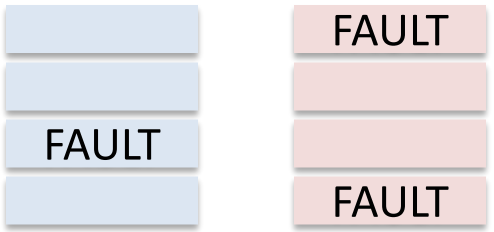


Wear-leveling: Move *gap* every 100 memory writes

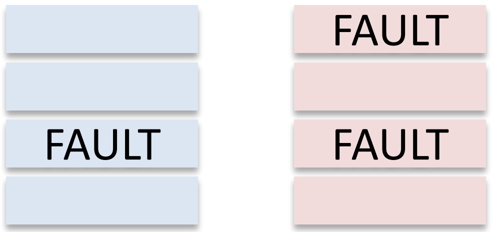
NVMAddr = (Start+Addr); if (PhysAddr >= Gap) NVMAddr++

Dynamically Replicated Memory

- Reuse faulty pages with non-overlapping faults

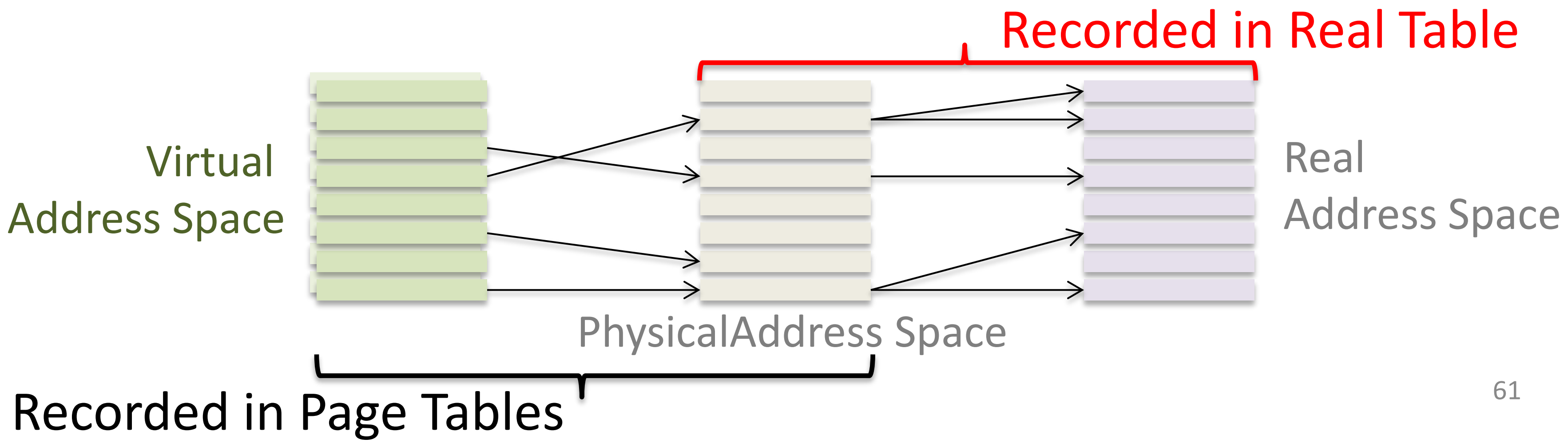


Compatible pages



Incompatible pages

- Record pairings in a new level of indirection



Summary

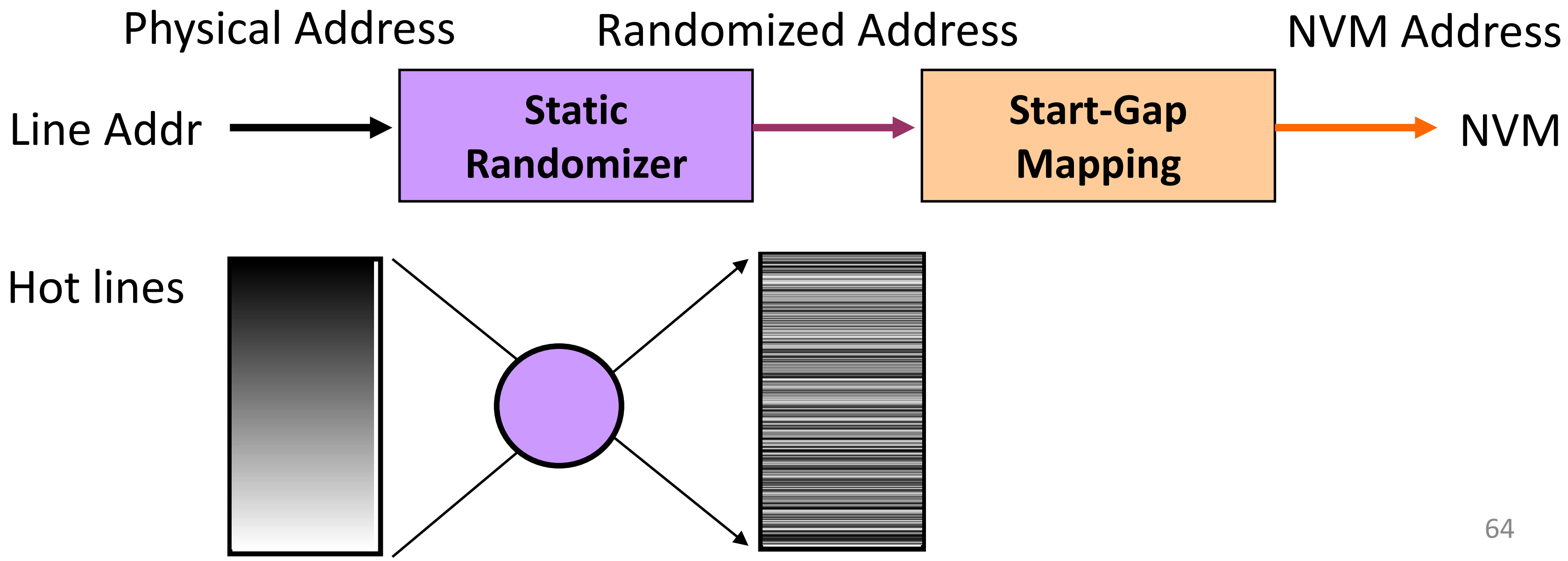
- Ordering support
  - Reduces unnecessary ordering constraints
  - Exposes persist concurrency
- Transaction support
  - Removes versioning software overheads
- Endurance support further increases lifetime

Questions?

Backup Slides

Randomized Start Gap

- Start gap may move spatially-close hot lines to other hot lines
- Randomize address space to spread hot regions uniformly


Physical Address
Randomized Address
NVM Address
Line Addr
Static Randomizer
Start-Gap Mapping
NVM
Hot lines