Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes

Qing Wang, Youyou Lu, Junru Li, Jiwu Shu

Tsinghua University

Persistent Memory (PM)

Enjoy benefits of both storage and memory !

Storage Features

- ❖ Persistent
- ❖ High-density

Memory Features

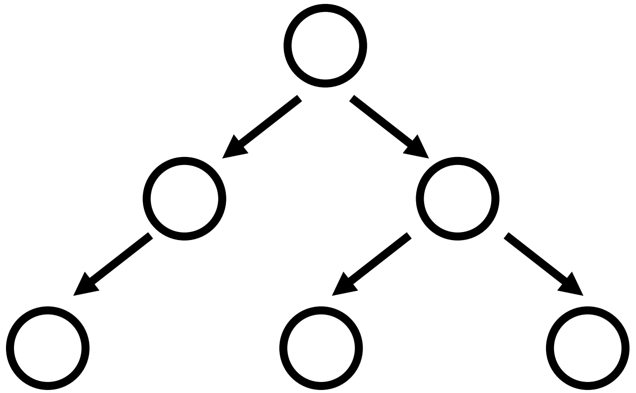
- ❖ Byte-addressable
- ❖ High-performance

The figure of Optane DIMM is from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

PM Indexes

A PM Index is

- ❖ Crash-consistent
- ❖ Without de(serialization) at runtime
- ❖ Support instant recovery



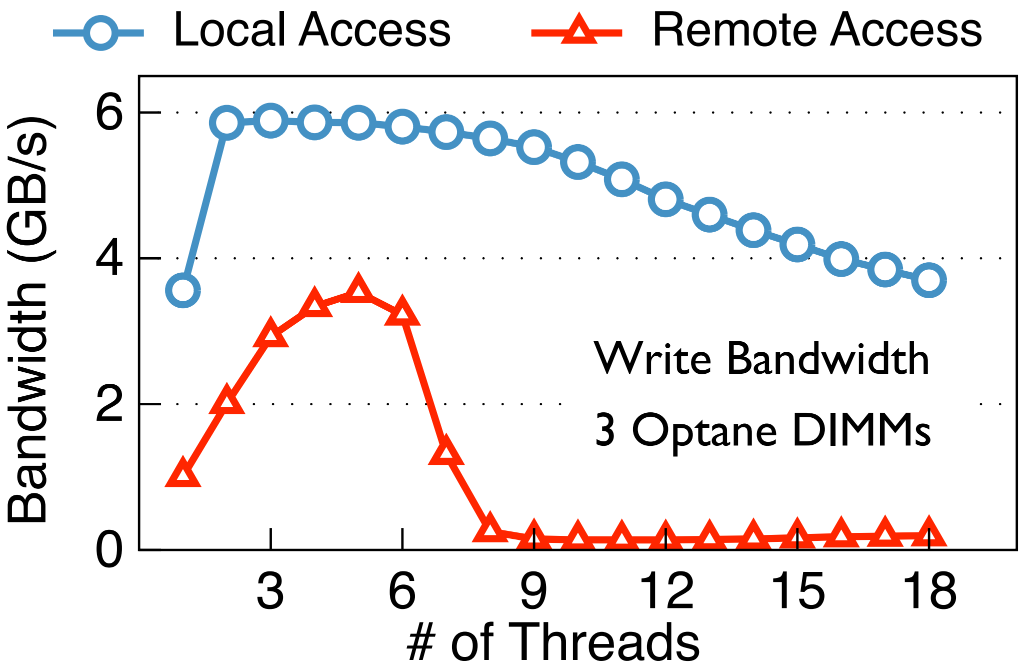
CDDS (FAST'11) WORT (FAST'17) CCEH (FAST'19) Clevel (ATC'20)

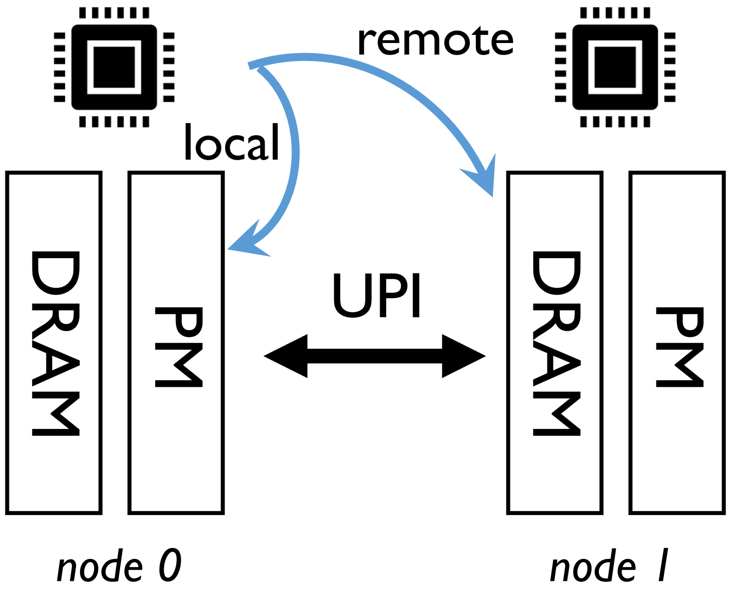
NV-Tree (FAST'15) Level (OSDI'18) Recipe (SOSP'19) TIPS (ATC'21)

Simulation & Emulation | **Optane DIMM**

NUMA impacts on PM Indexes (1)

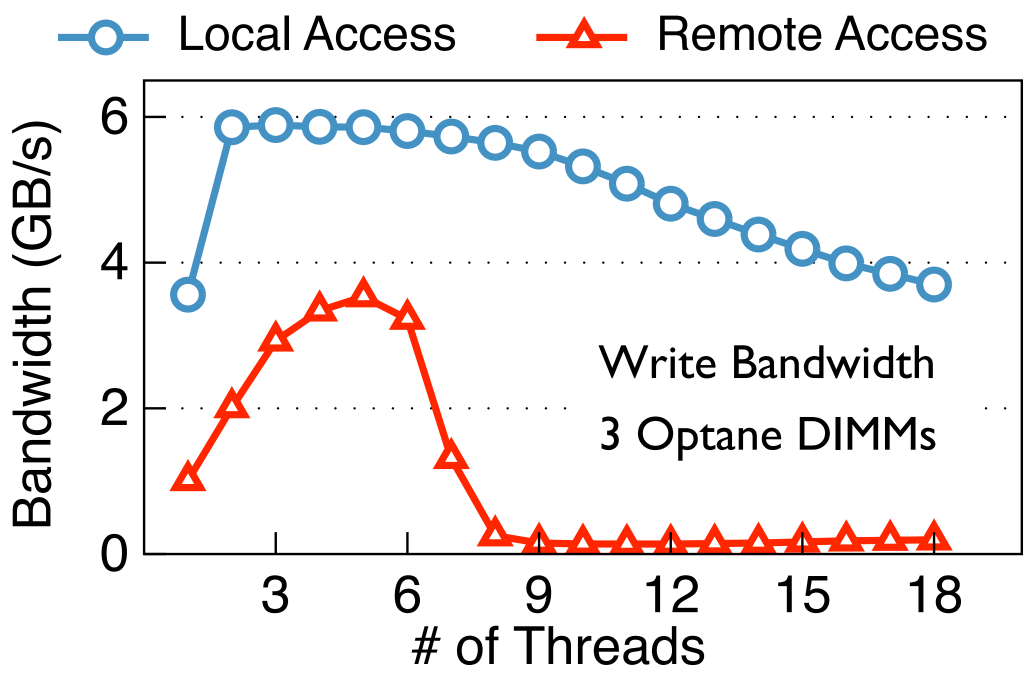
However, NUMA impacts on PM Indexes are under-explored

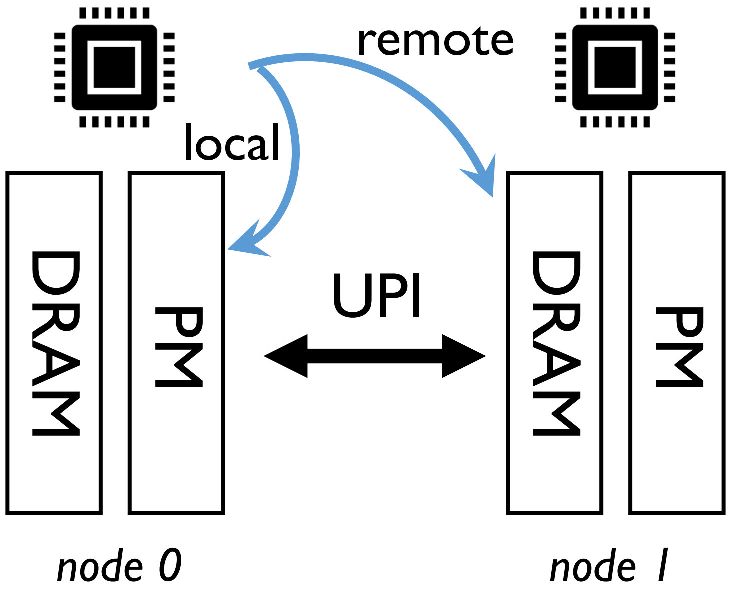




NUMA impacts on PM Indexes (1)

However, NUMA impacts on PM Indexes are under-explored



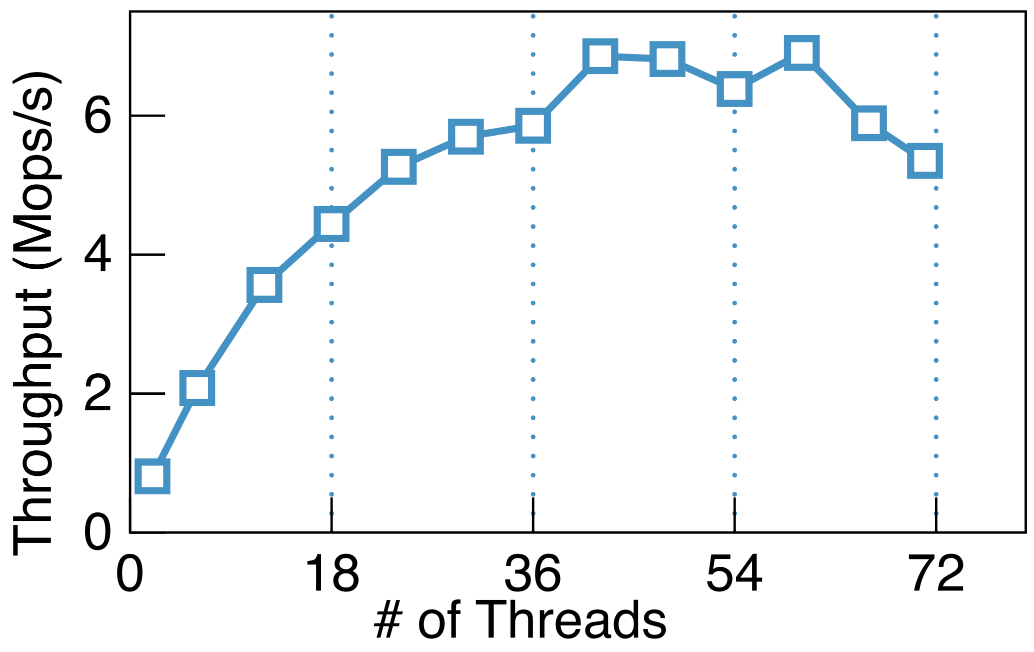


1) Peak remote PM write bandwidth is low
2) Concurrent remote accesses cause bandwidth collapses

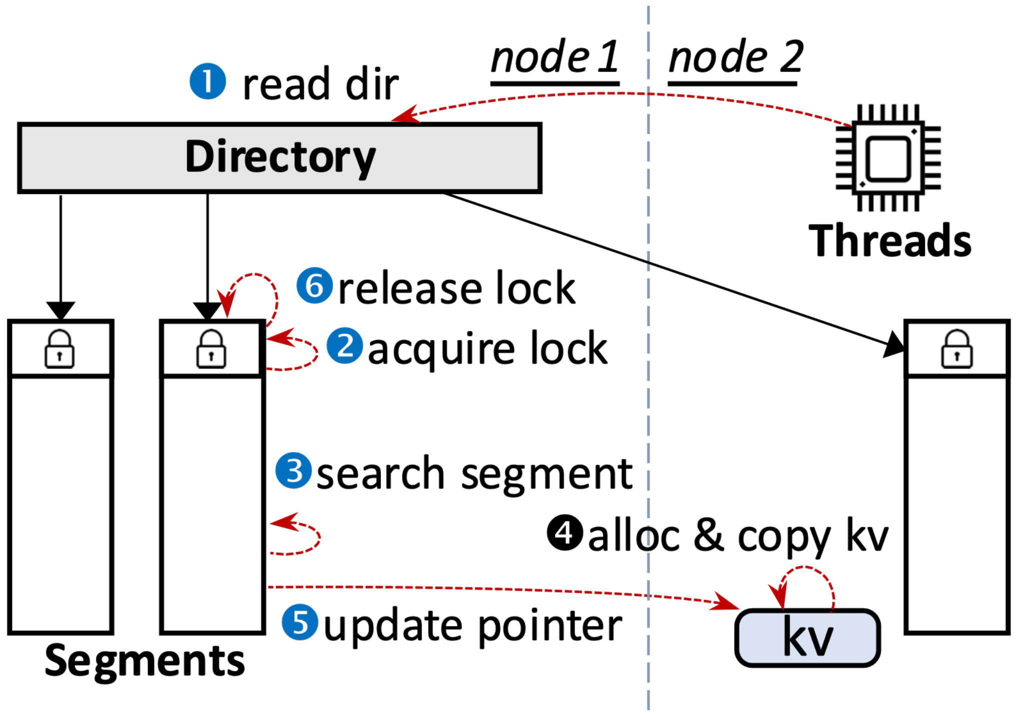
The result is consistent with multiple recent studies (e.g., FAST'19, FAST'21, SIGMOD'21)

NUMA impacts on PM Indexes (2)

Take CCEH (FAST'19, Hash Table) as an example



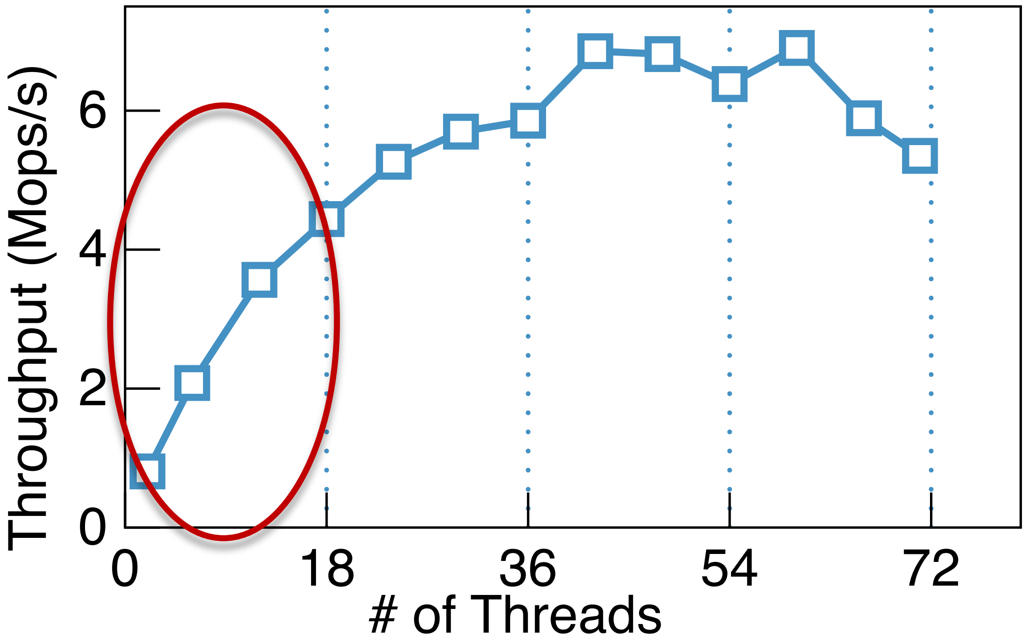
4 nodes, 50% insert/update, Zipfian 0.99



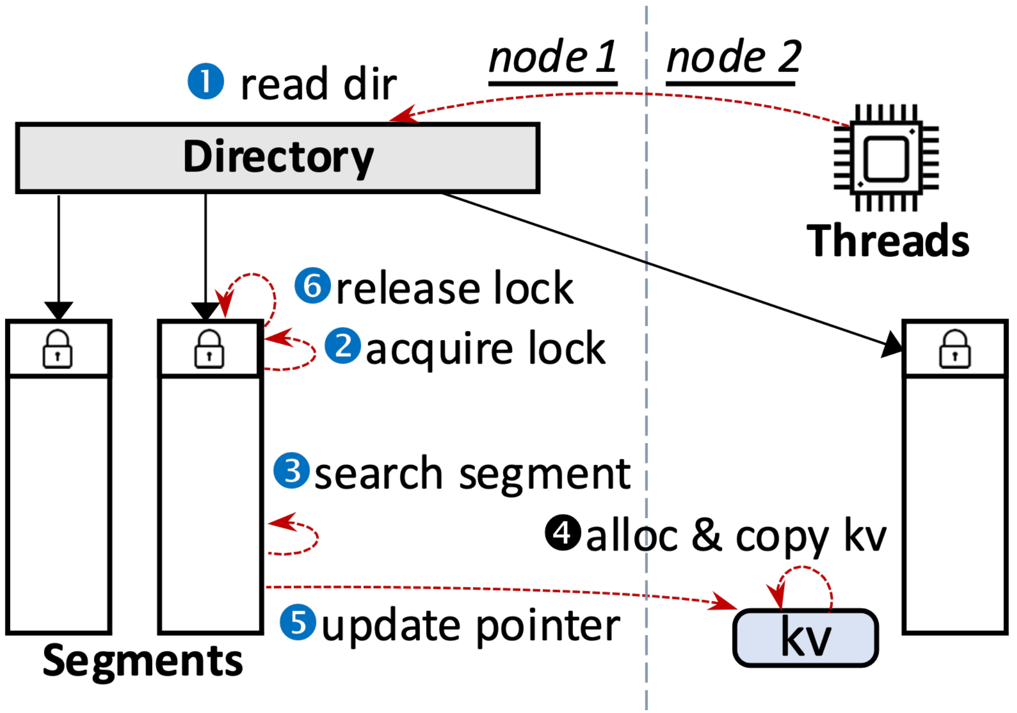
Insert op, up to 5 remote PM accesses

NUMA impacts on PM Indexes (2)

Take CCEH (FAST'19, Hash Table) as an example



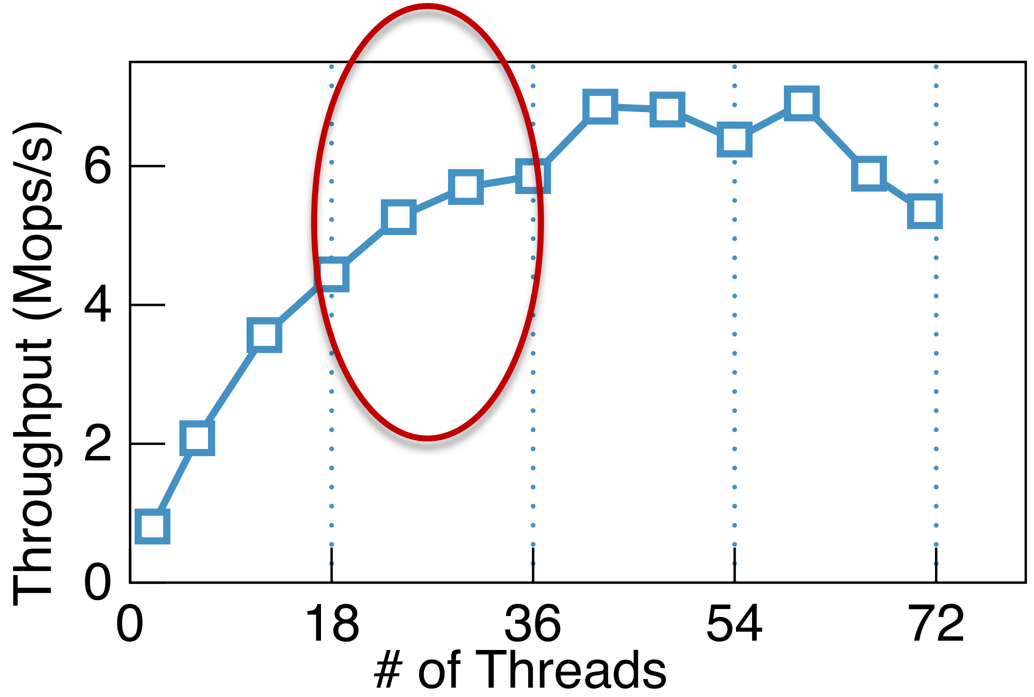
4 nodes, 50% insert/update, Zipfian 0.99



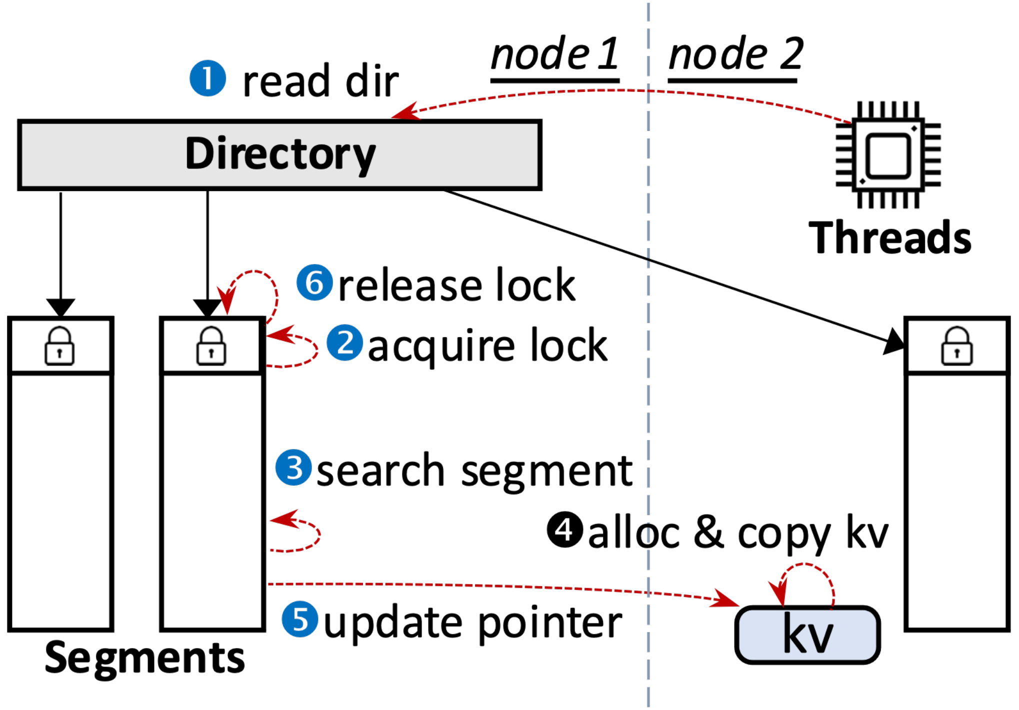
Insert op, up to 5 remote PM accesses

NUMA impacts on PM Indexes (2)

Take CCEH (FAST'19, Hash Table) as an example



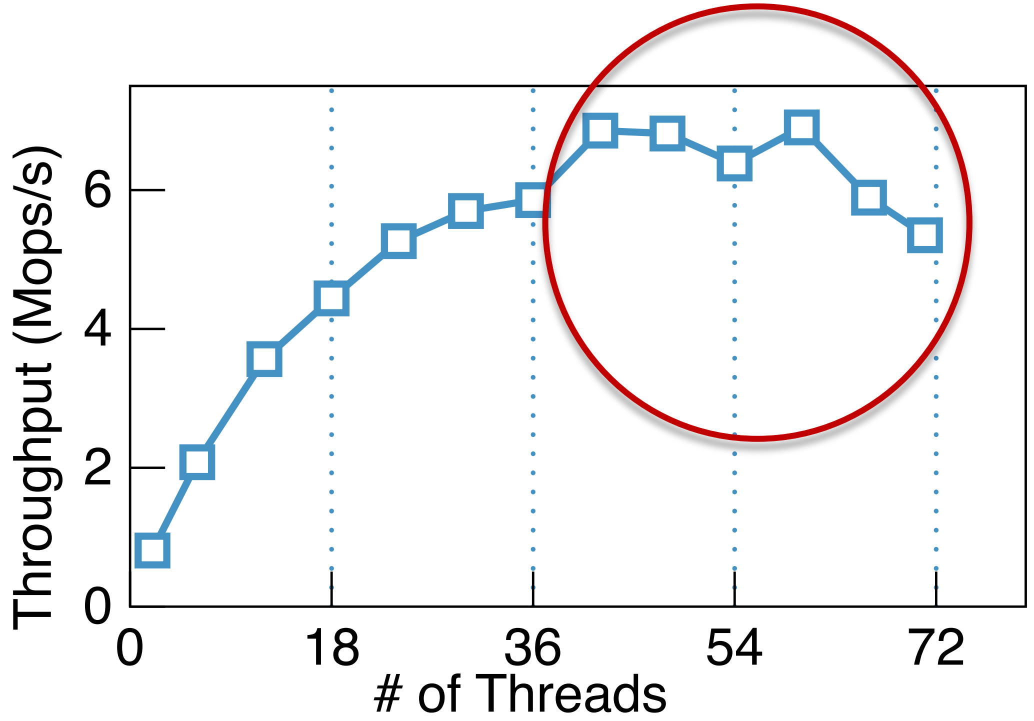
4 nodes, 50% insert/update, Zipfian 0.99



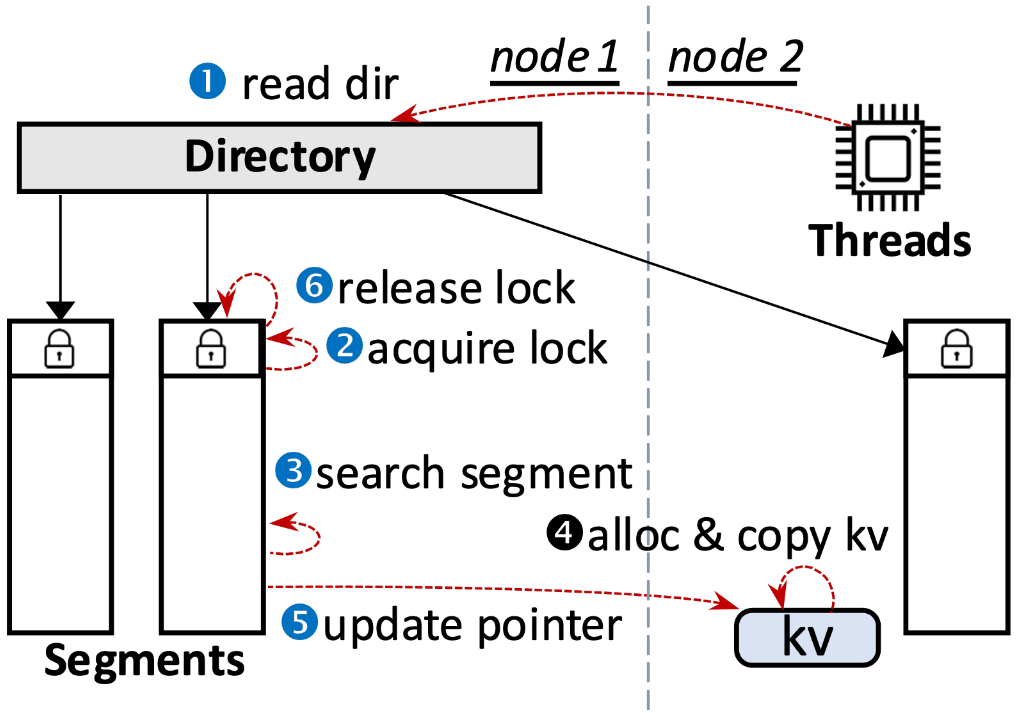
Insert op, up to 5 remote PM accesses

NUMA impacts on PM Indexes (2)

Take CCEH (FAST'19, Hash Table) as an example



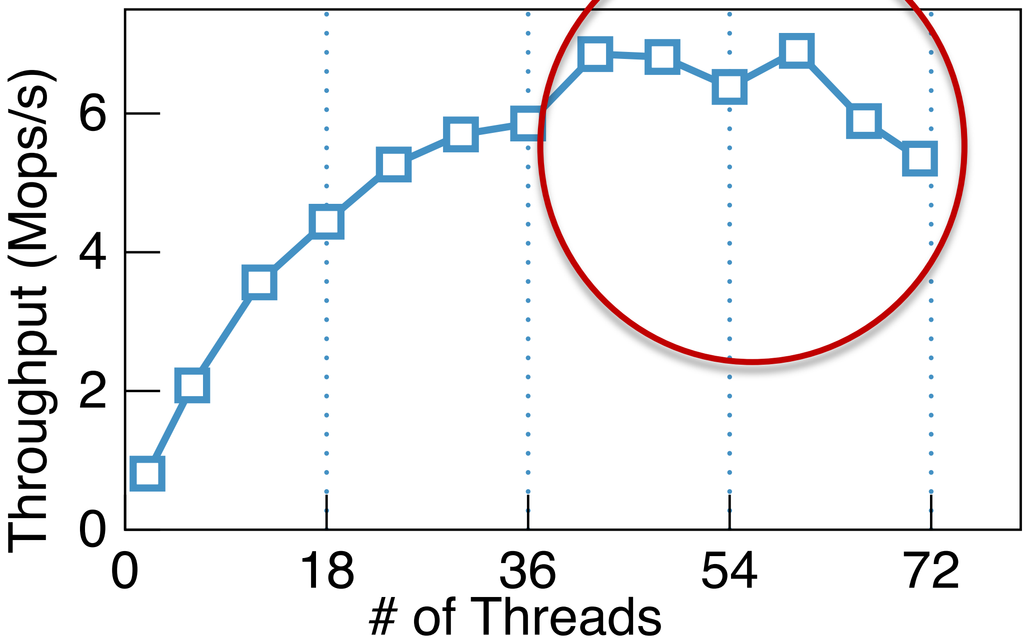
4 nodes, 50% insert/update, Zipfian 0.99



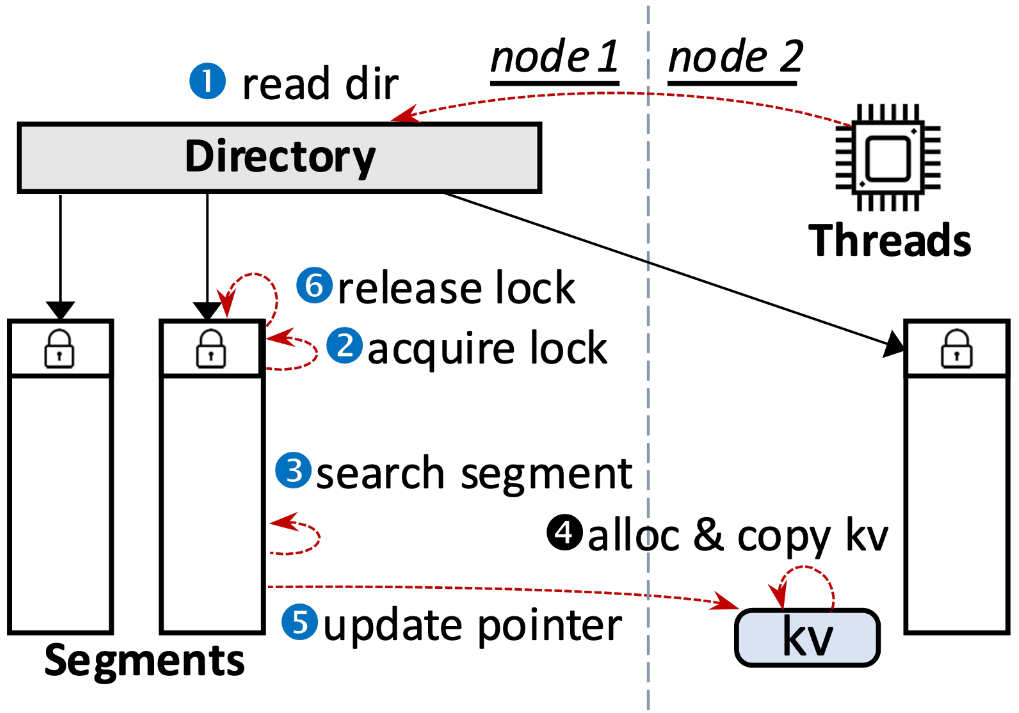
Insert op, up to 5 remote PM accesses

NUMA impacts on PM Indexes (2)

Take CCEH (FAST'19, Hash Table) as an example



4 nodes, 50% insert/update, Zipfian 0.99



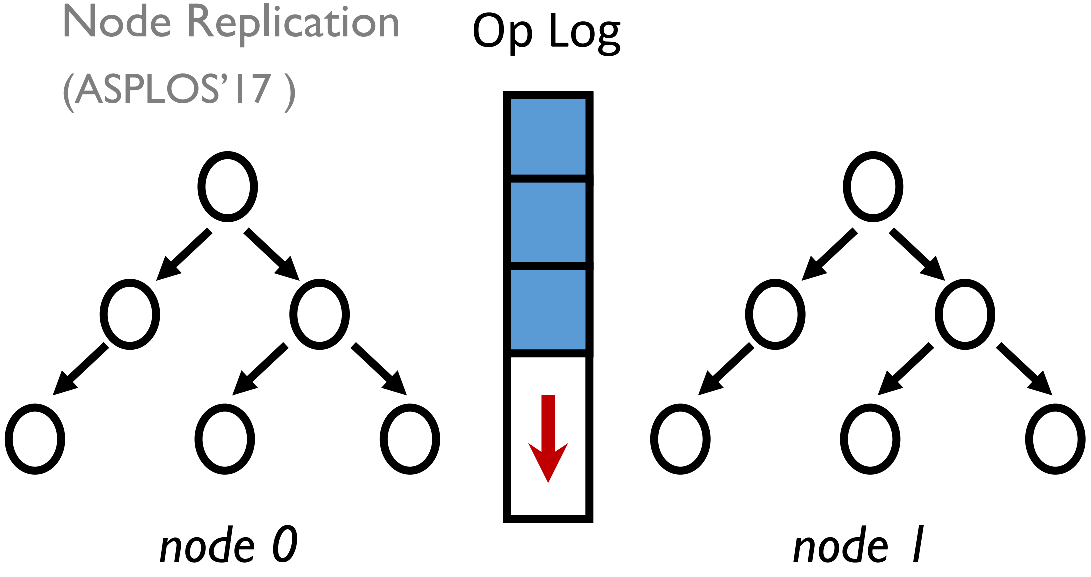
Insert op, up to 5 remote PM accesses

A fast PM index should reduces remote PM accesses, especially for writes

Limitations of Replication-based Approach

NUMA-aware DRAM indexes always use replication

Node Replication (ASPLOS'17)



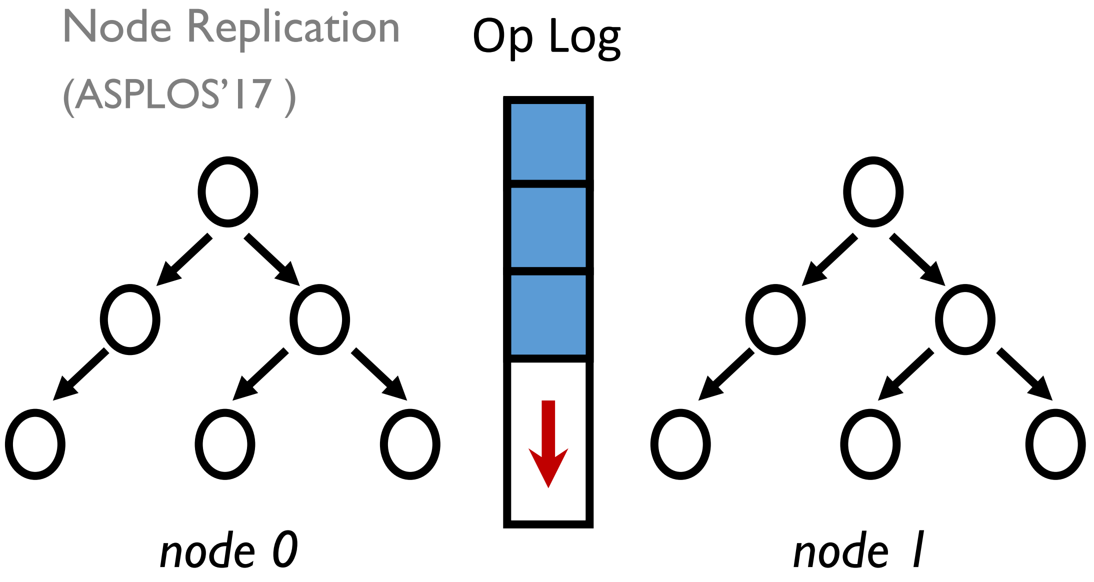
Each NUMA node maintains an index replica

A shared log records index operations

Synchronize replicas via replaying log entries

Limitations of Replication-based Approach

NUMA-aware DRAM indexes always use replication



Each NUMA node maintains an index replica

A shared log records index operations

Synchronize replicas via replaying log entries

Three critical limitations :

- No crash consistency
- Multiple times of PM consumption
- Amplifying local accesses significantly
  - Every update op is executed on every node
  - PM has limited write bandwidth (1/6 DRAM)

Outline

- Background & Motivation
- Nap – a Black-Box Approach to NUMA-Aware PM Indexes
- Results
- Takeaways

Key Idea (1) - Making Hot Accesses NUMA-aware

Modern workloads always feature skewed access patterns

- YCSB (SOCC'10), Twitter cache workloads (OSDI'20) ….
- Top 100K hottest items receive > 50% accesses (typical Zipfian 0.99, 2 billion items)

Key Idea (1) - Making Hot Accesses NUMA-aware

Modern workloads always feature skewed access patterns

- YCSB (SOCC'10), Twitter cache workloads (OSDI'20) ….
- Top 100K hottest items receive > 50% accesses (typical Zipfian 0.99, 2 billion items)

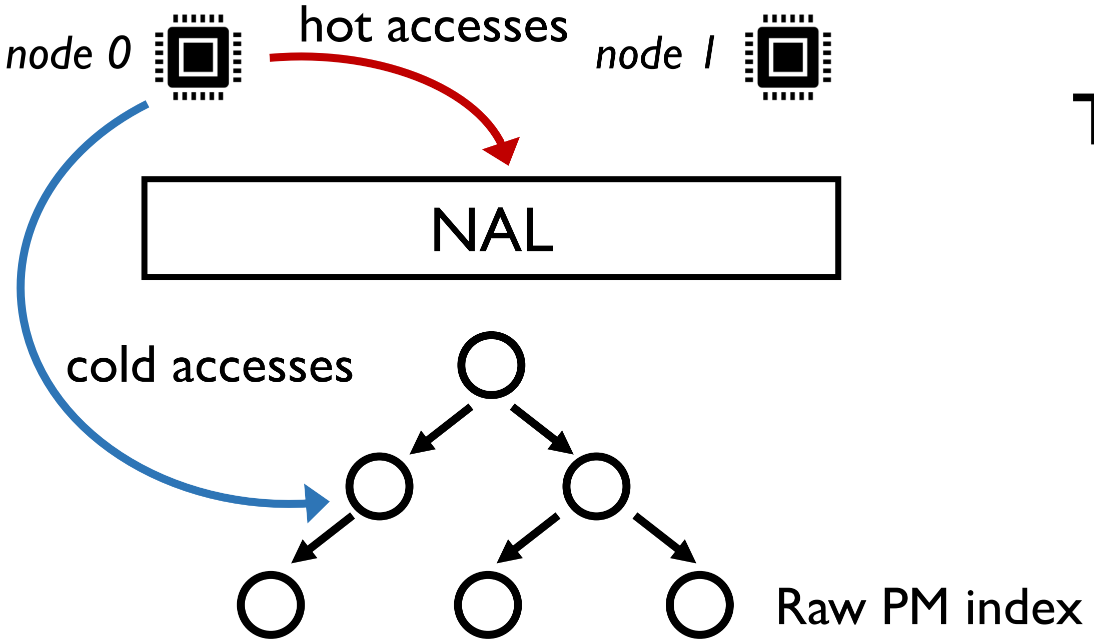
Nap uses a NUMA-aware layer (NAL) to absorb hot accesses

Key Idea (1) - Making Hot Accesses NUMA-aware

Modern workloads always feature skewed access patterns

- YCSB (SOCC'10), Twitter cache workloads (OSDI'20) ….
- Top 100K hottest items receive > 50% accesses (typical Zipfian 0.99, 2 billion items)

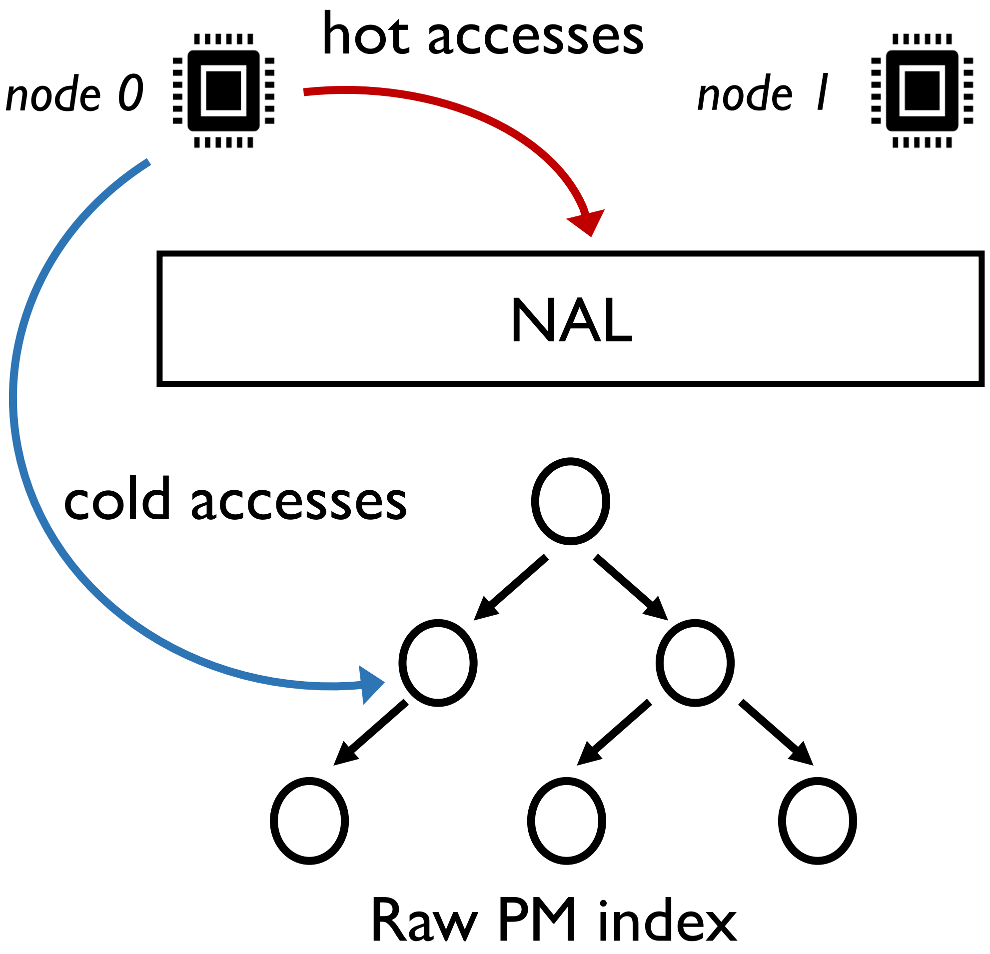
Nap uses a NUMA-aware layer (NAL) to absorb hot accesses



Two components:

- Raw PM index – accommodate a large number of cold items
- NAL – handle accesses to hot items in a NUMA-aware way

Key Idea (1) - Making Hot Accesses NUMA-aware



NAL brings three advantages

Eliminate lots of remote PM accesses

- Hot items receive a significant percentage of accesses

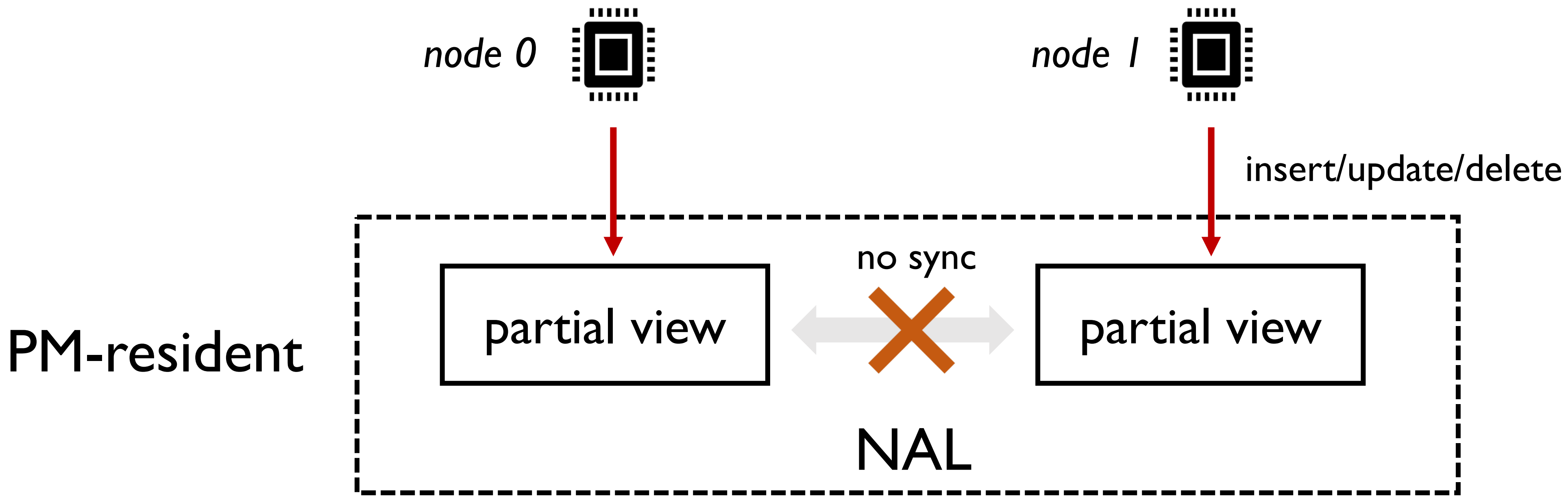
Black-box approach

- Raw PM index can be any concurrent PM index
- Do not require inner-knowledge of raw PM index

Bounded memory usage and recovery time

- Small hot set => Small NAL

Key Idea (2) - Minimizing PM State Synchronization



- NAL maintains per-node partial views in PM
- partial views absorb updates from local threads
- forbid PM state sync between partial views
  - avoid amplification of local PM accesses

Challenges for Practical Design

Making key ideas practical must address several challenges

How to serve lookups to hot items ?

- ❖ the latest value of hot items can be in any partial view (so we call it *partial*)

How to ensure recoverability ?

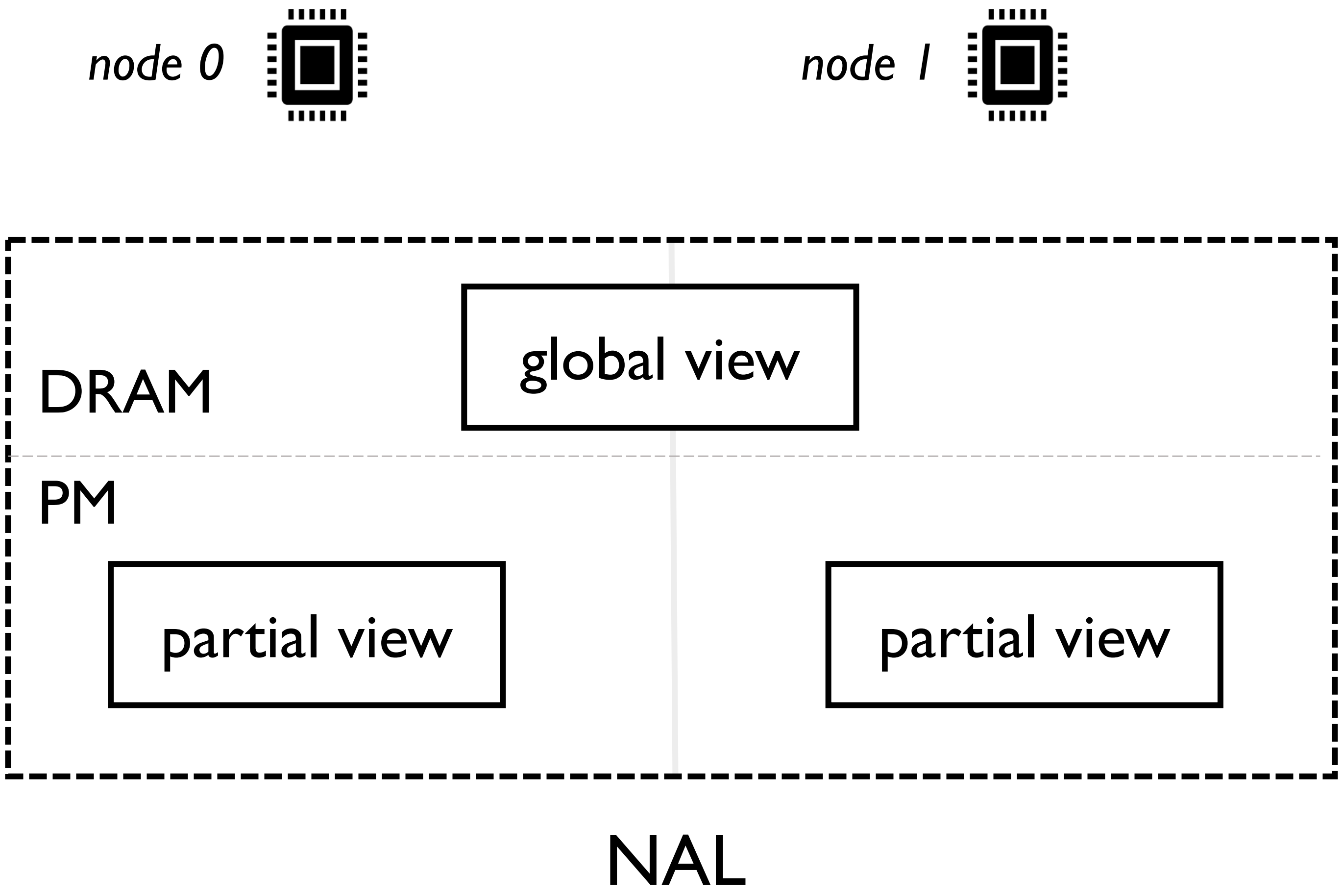
- ❖ upon recovery, restoring the system into a correct state when there are multiple partial views

How to handle hotspots shift ?

- ❖ hotspots shift over time

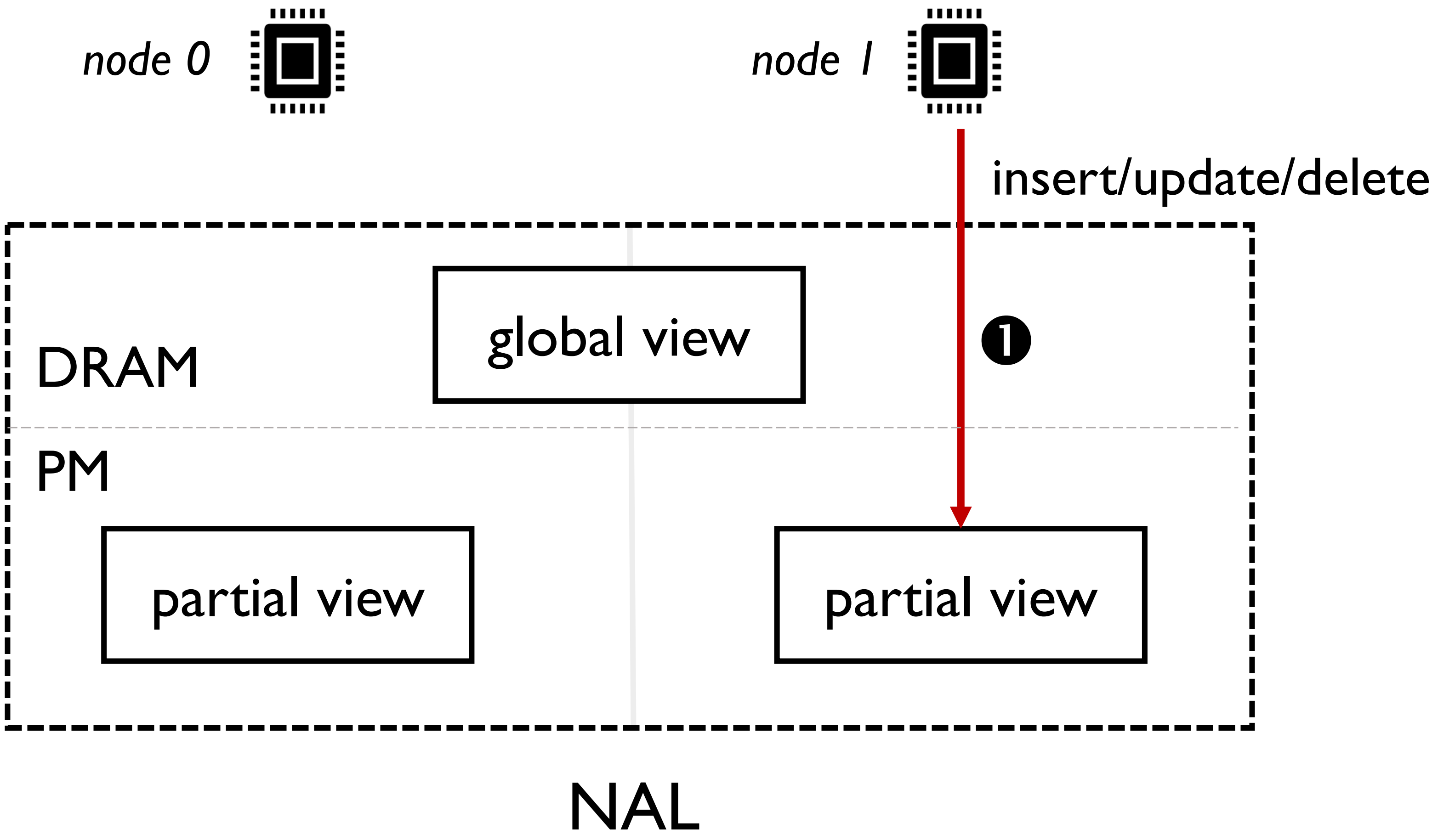
Global View

Nap uses a DRAM-resident global view for lookups to hot items



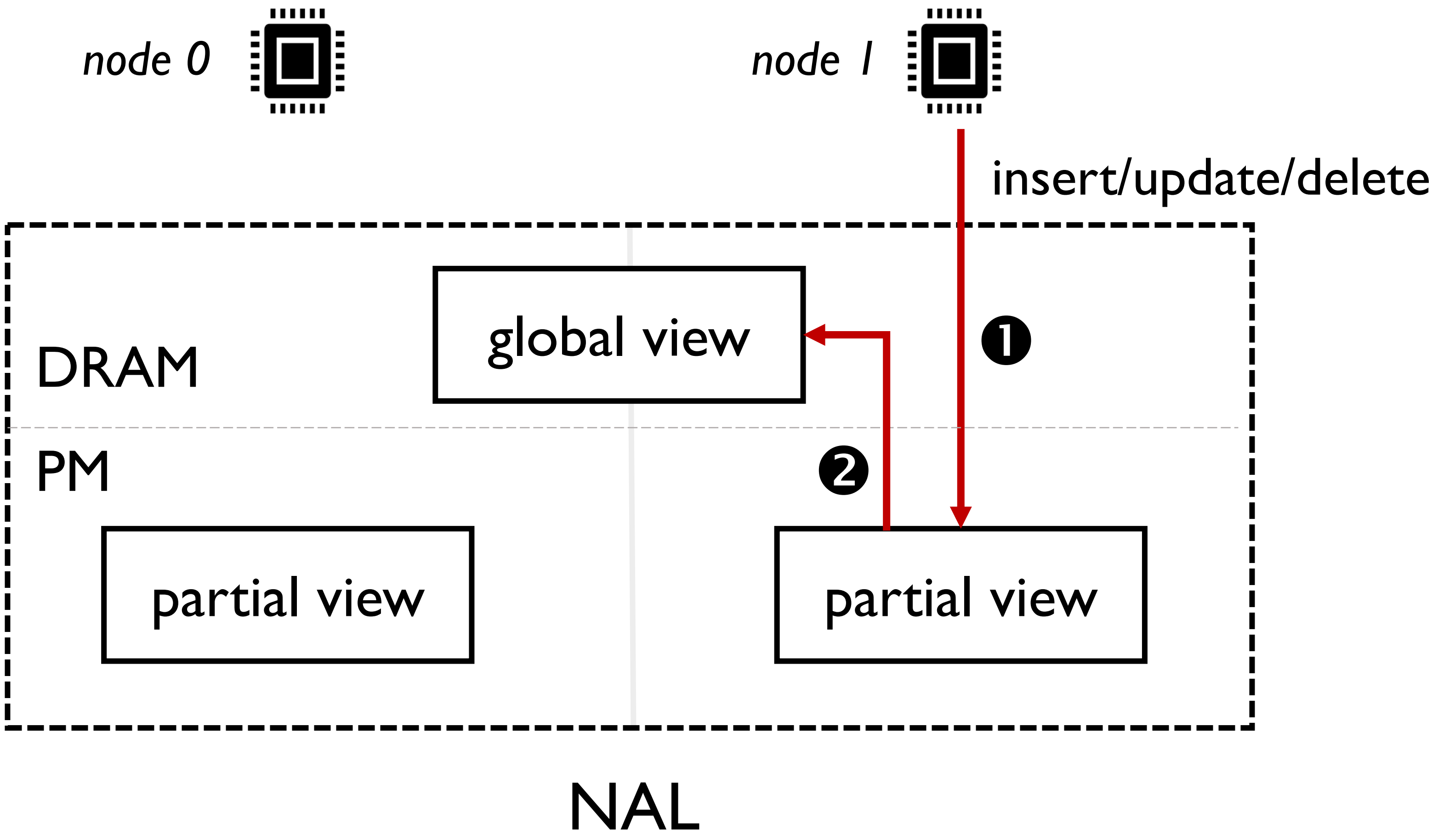
Global View

Nap uses a DRAM-resident global view for lookups to hot items



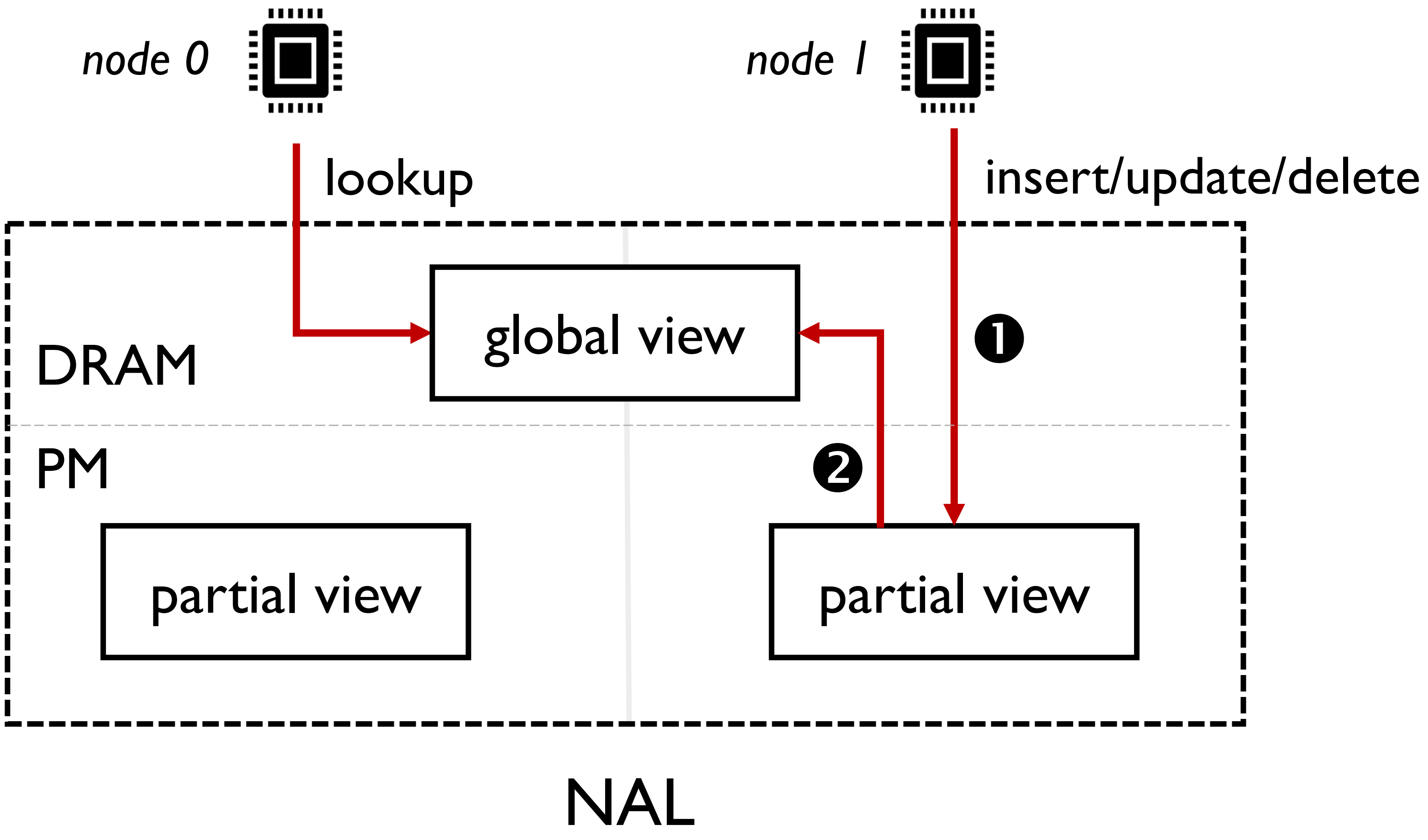
Global View

Nap uses a DRAM-resident global view for lookups to hot items



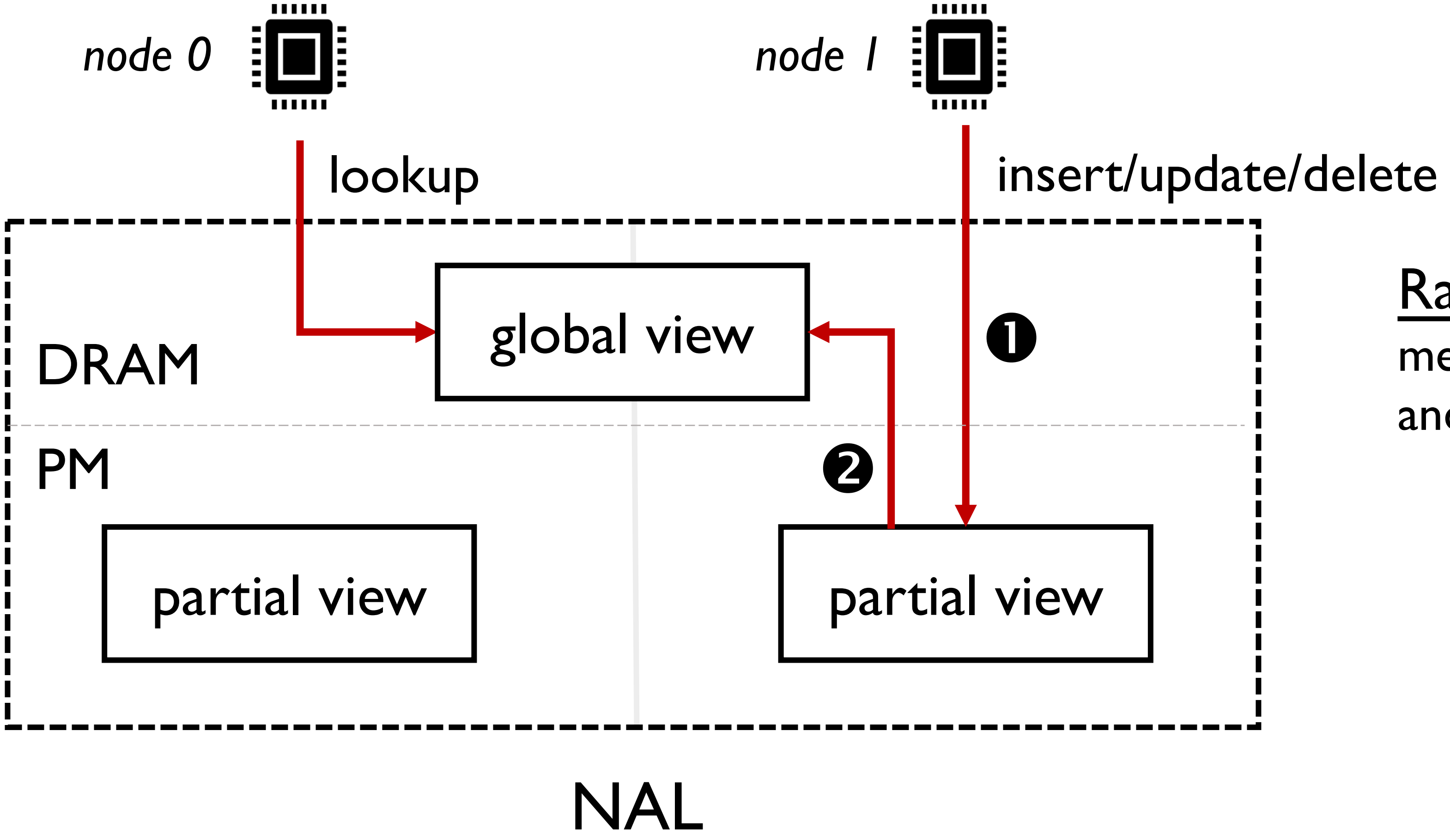
Global View

Nap uses a DRAM-resident global view for lookups to hot items



Global View

Nap uses a DRAM-resident global view for lookups to hot items

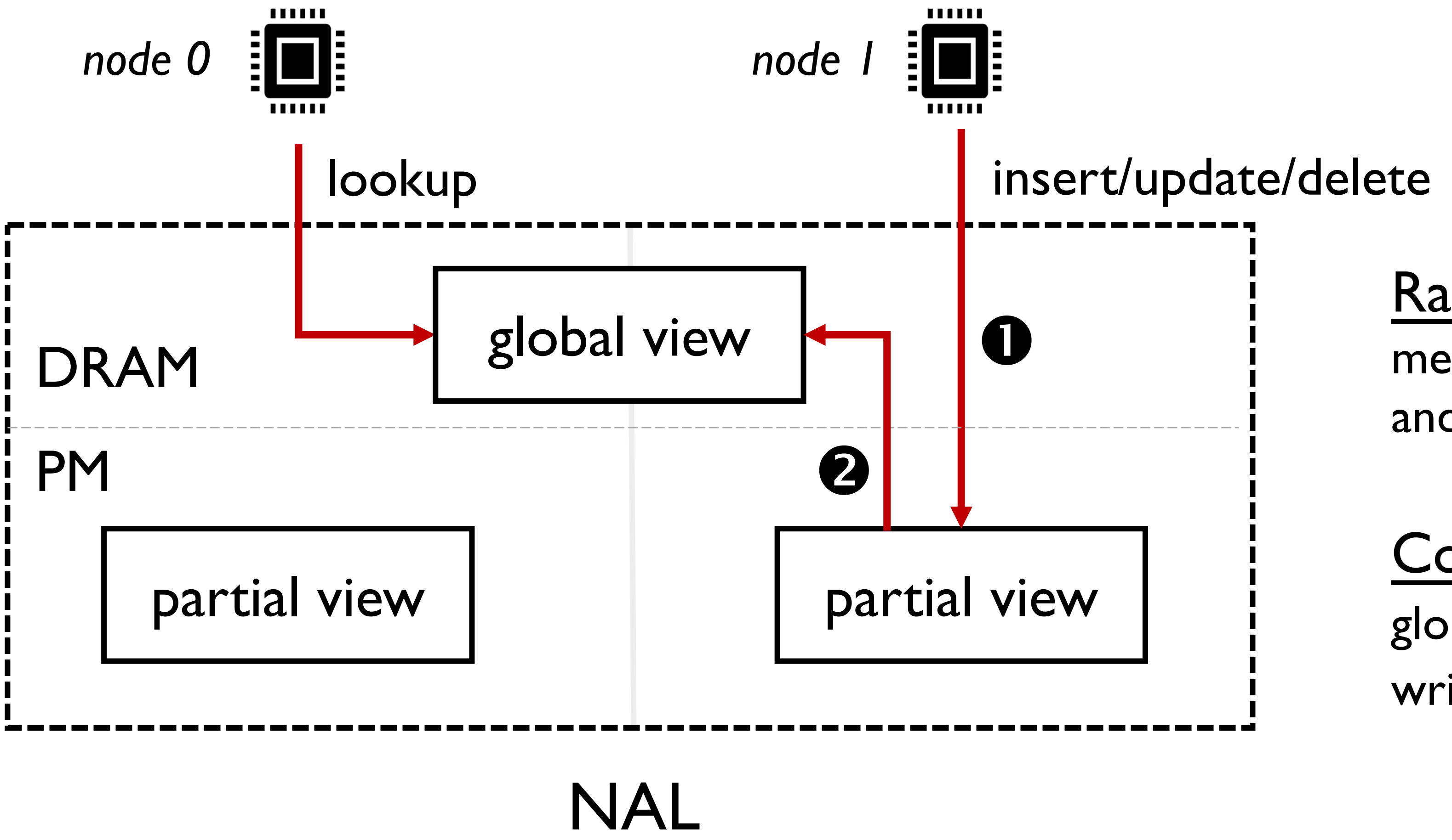


<u>Range queries</u>

merge items from global view and raw PM index

Global View

Nap uses a DRAM-resident global view for lookups to hot items



<u>Range queries</u>

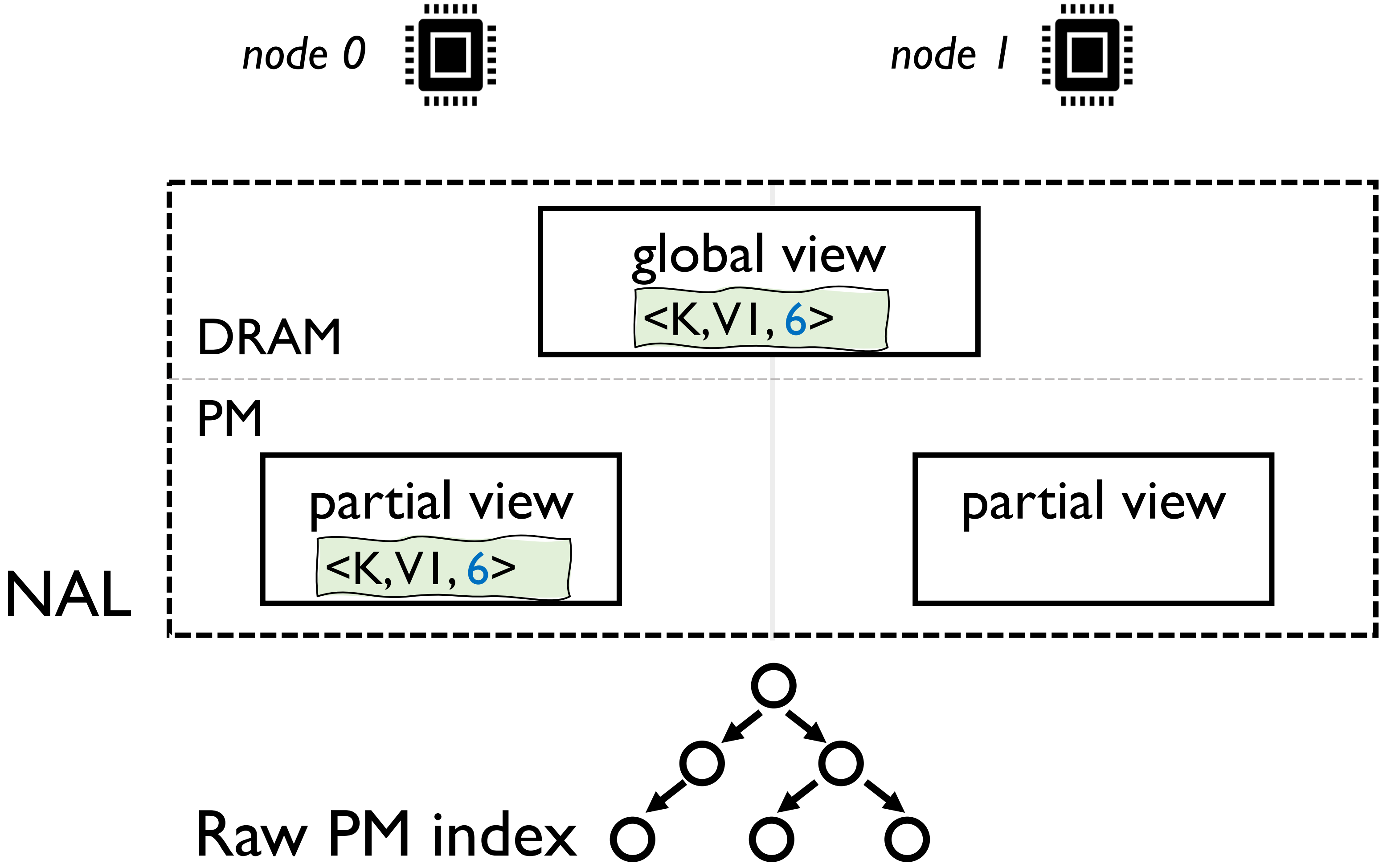
merge items from global view and raw PM index

<u>Concurrency Control</u>

global view maintains readers-writer locks for each hot item

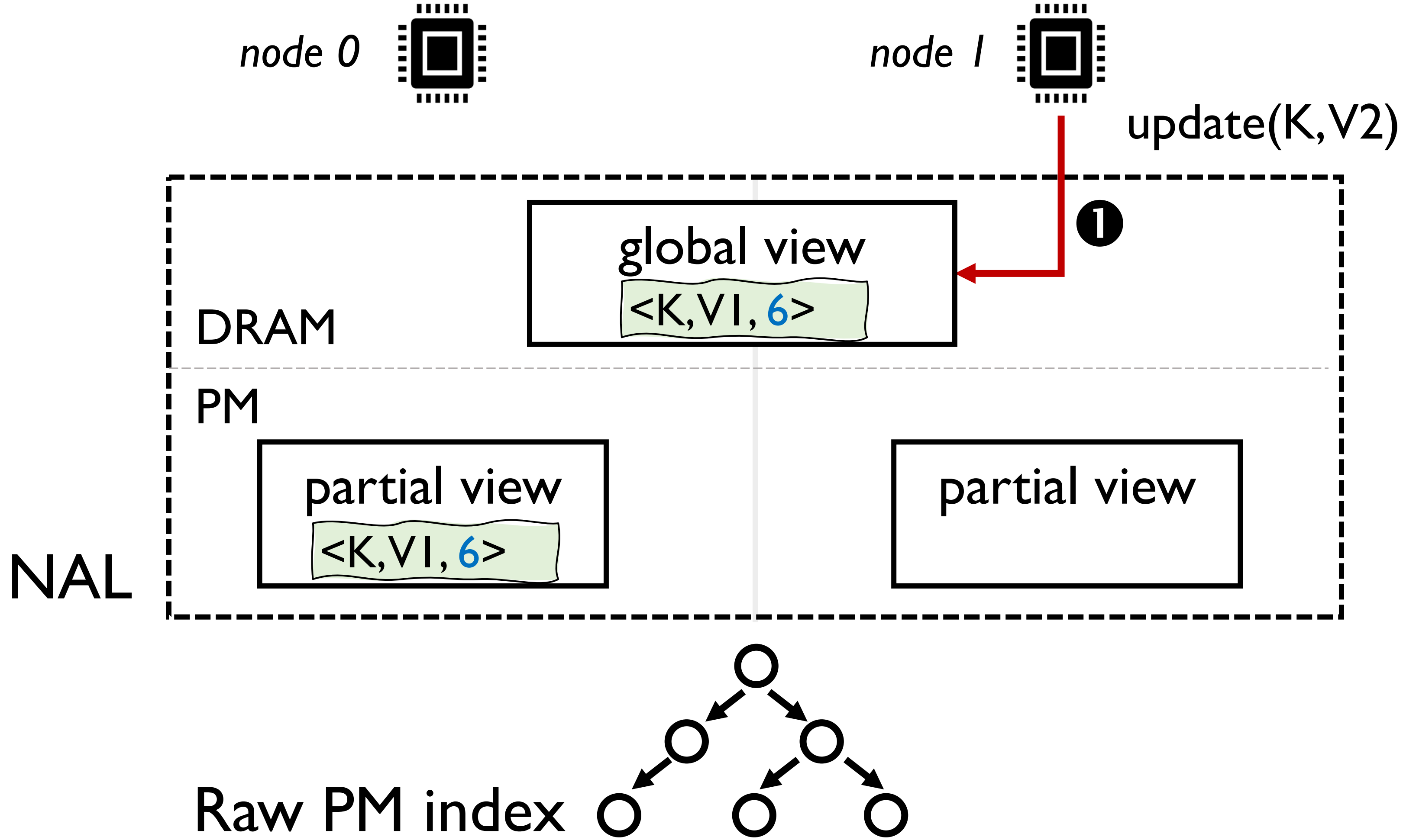
Version-based Updates

Nap uses versions to order updates to different partial views



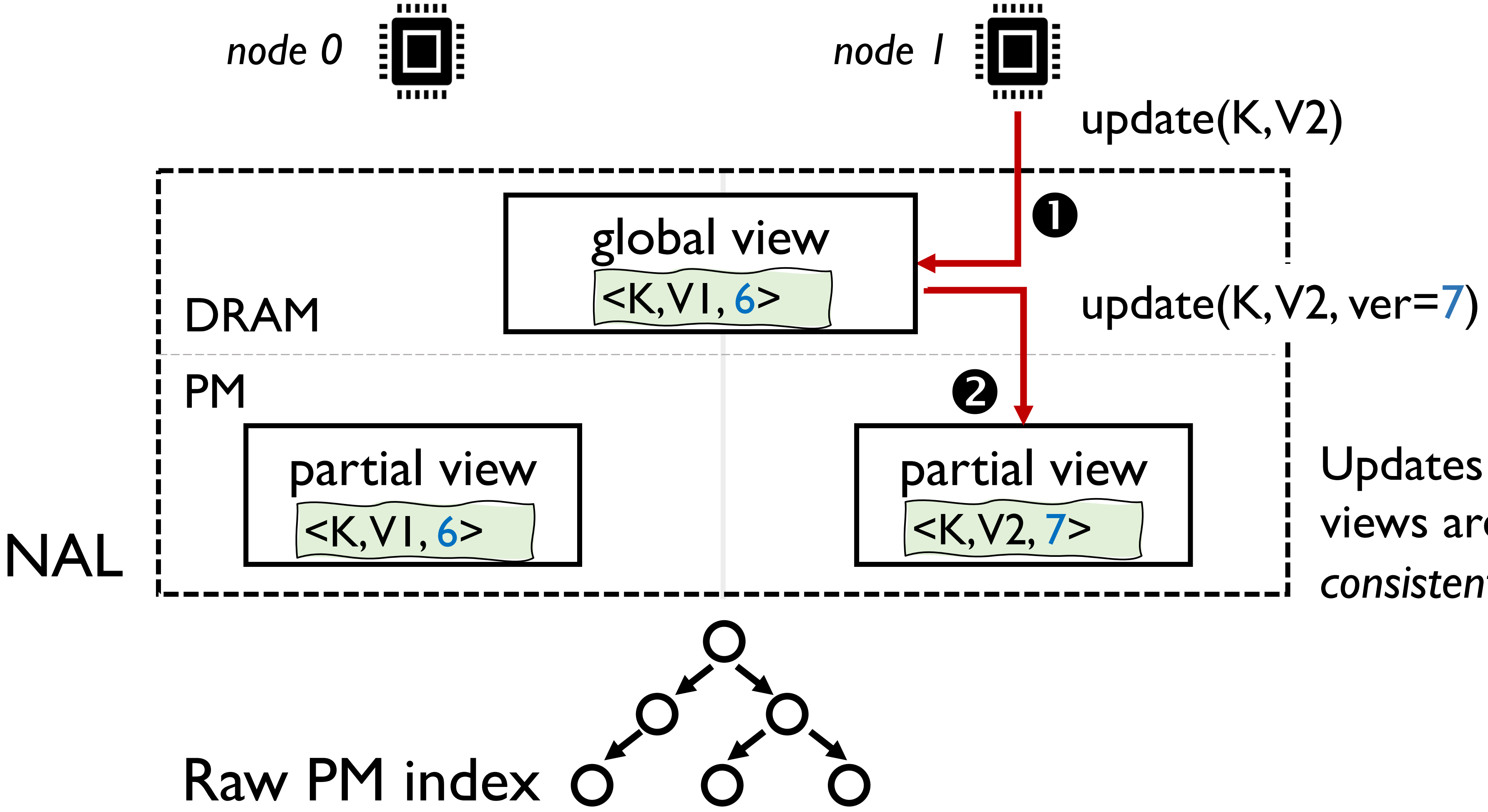
Version-based Updates

Nap uses versions to order updates to different partial views



Version-based Updates

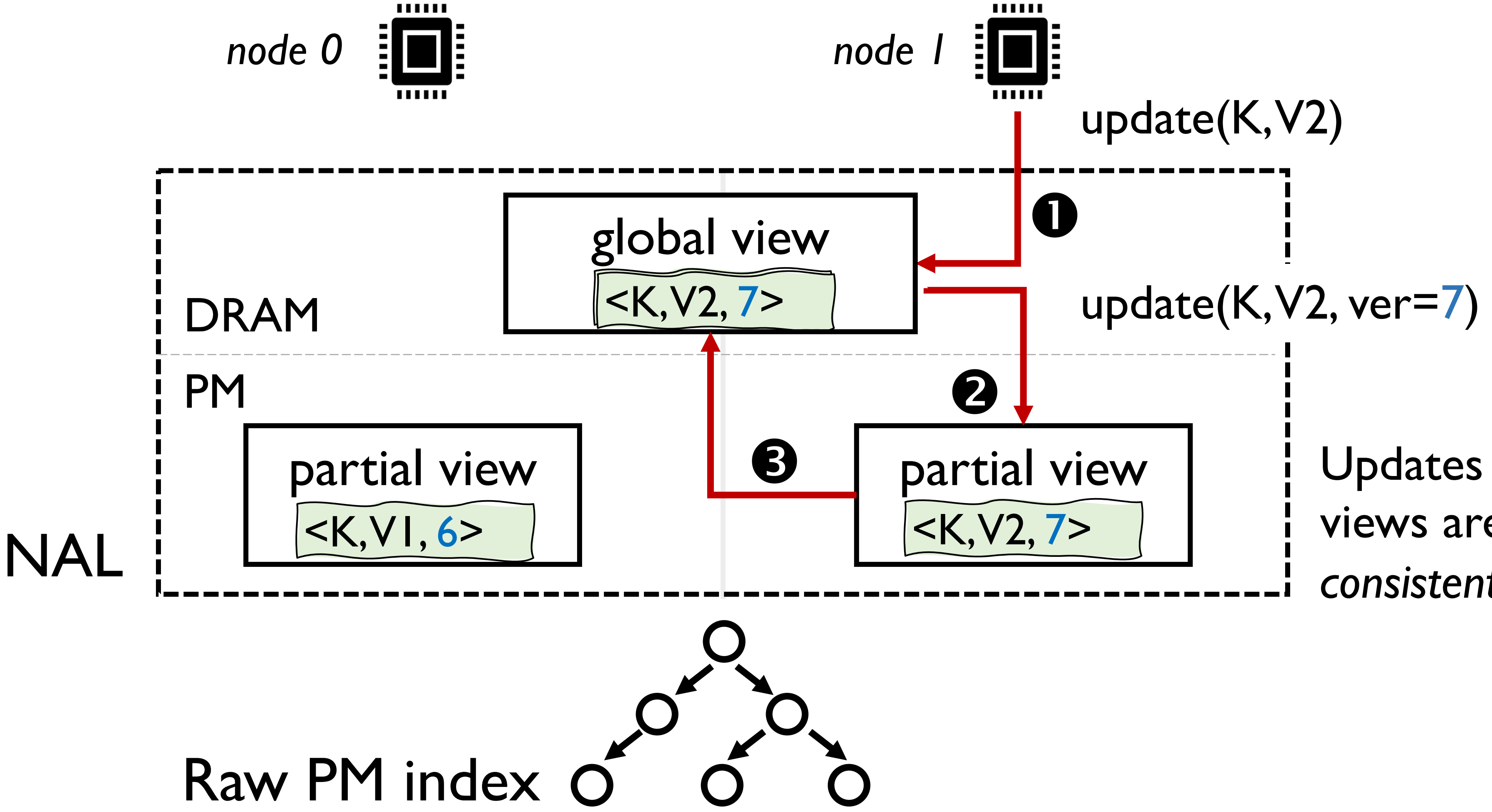
Nap uses versions to order updates to different partial views



Updates to partial views are *crash-consistent*

Version-based Updates

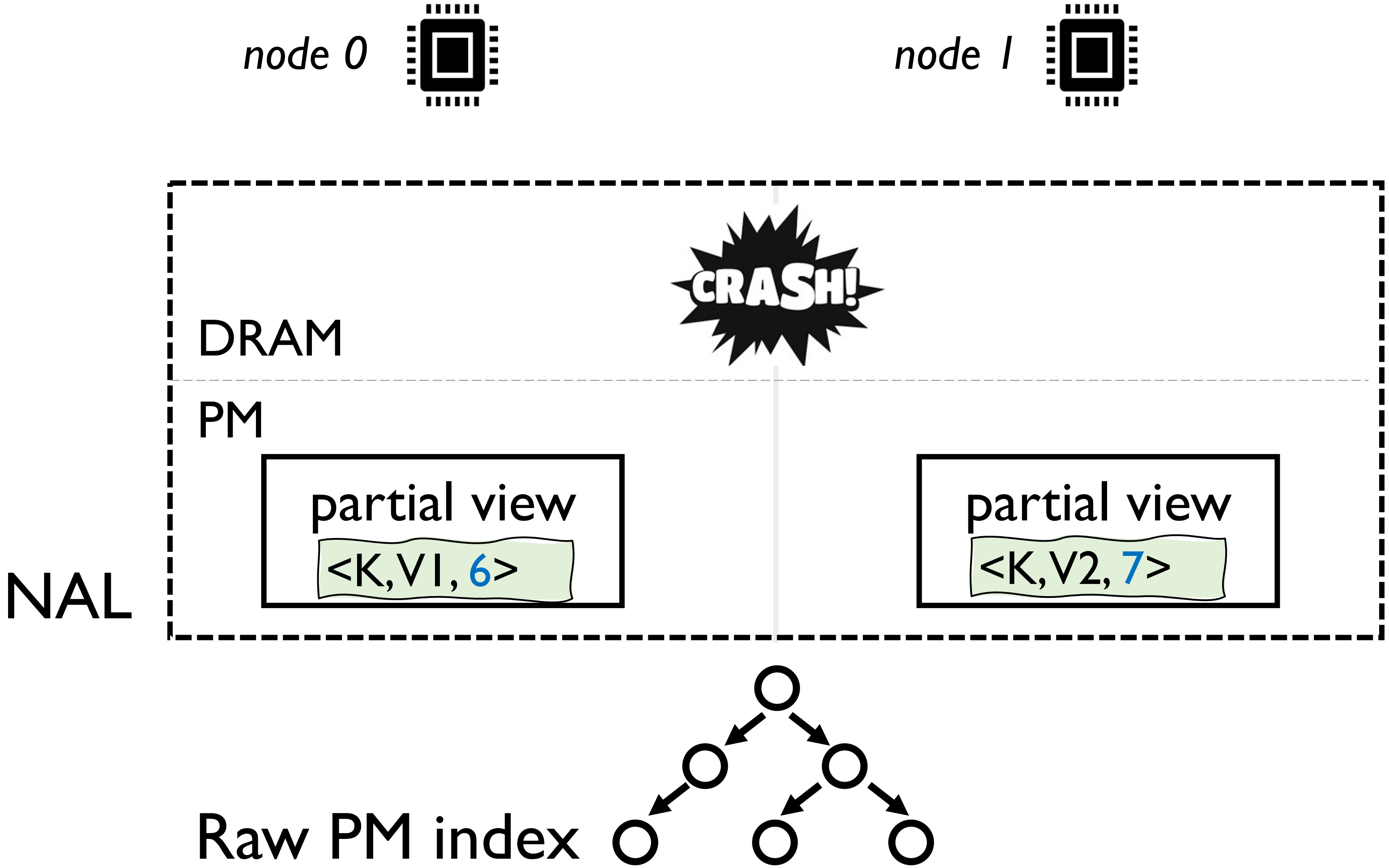
Nap uses versions to order updates to different partial views



Updates to partial views are *crash-consistent*

Version-based Updates

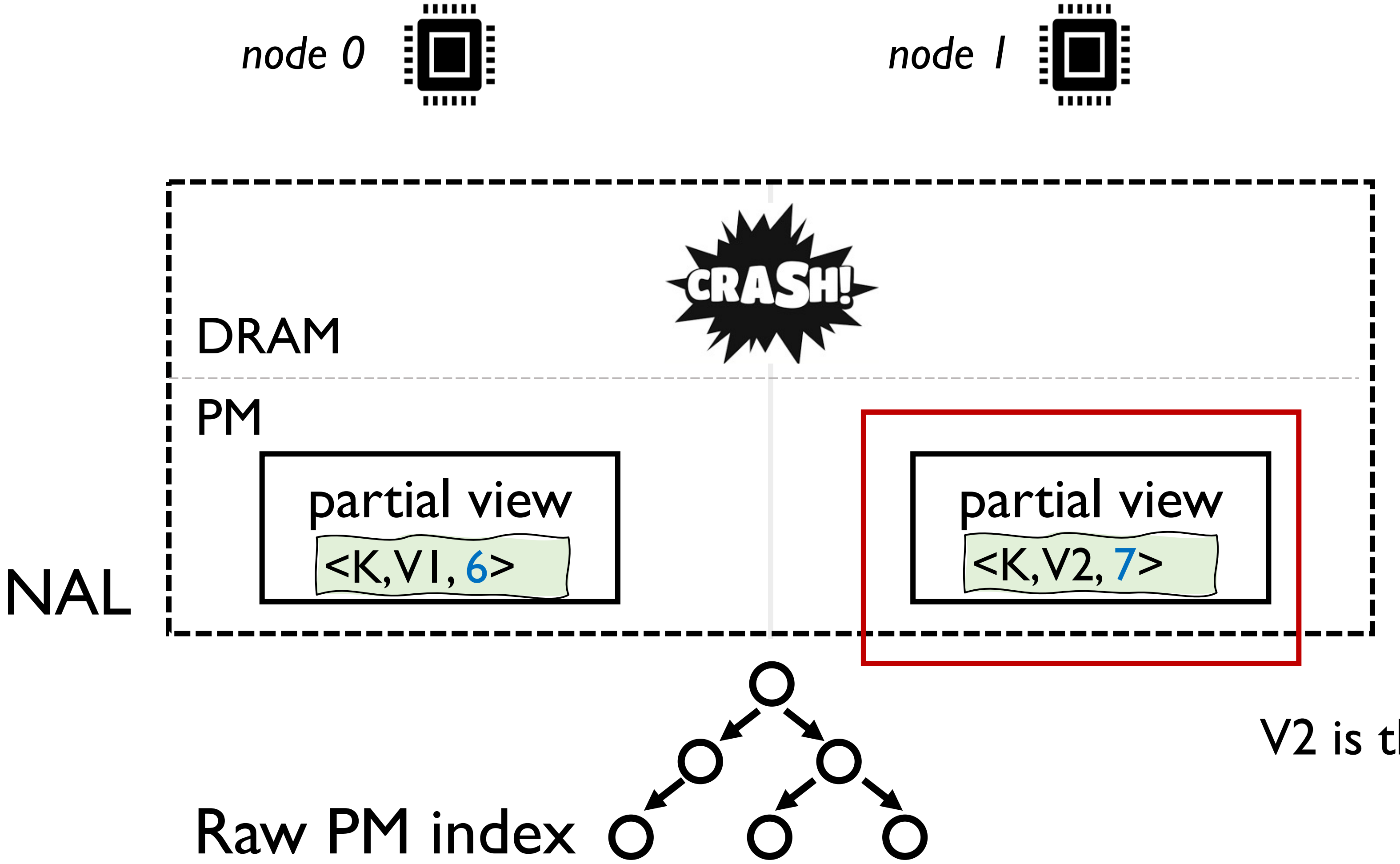
Nap uses versions to order updates to different partial views



Updates to partial views are *crash-consistent*

Version-based Updates

Nap uses versions to order updates to different partial views

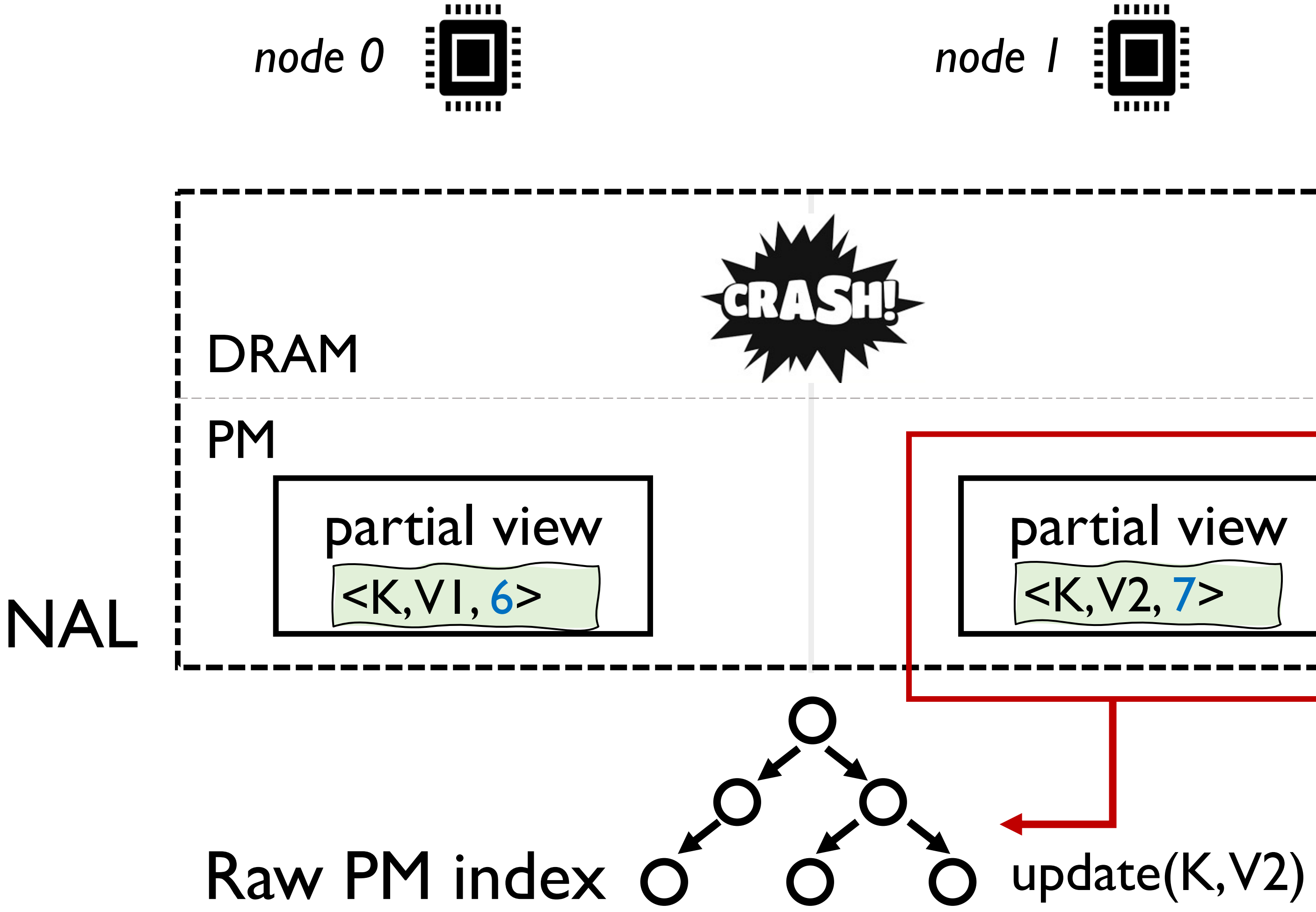


Updates to partial views are *crash-consistent*

V2 is the latest, since 7 > 6

Version-based Updates

Nap uses versions to order updates to different partial views

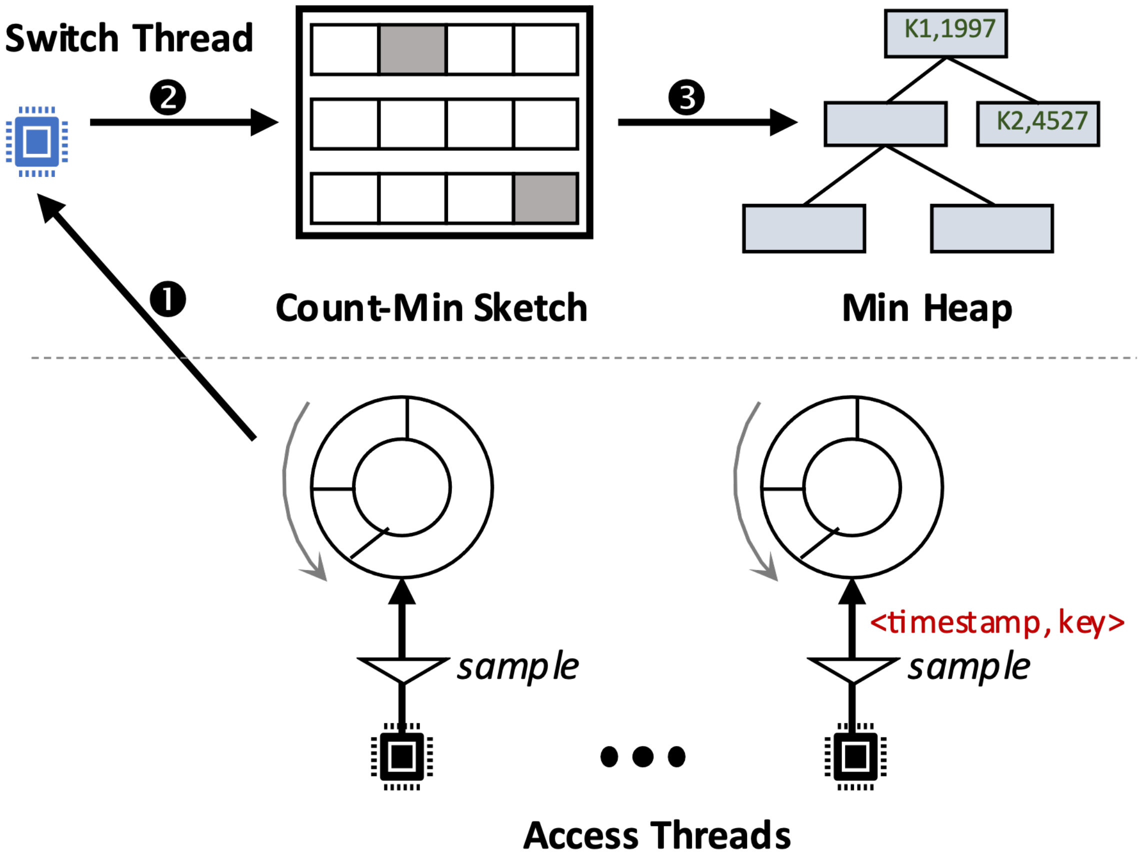


Updates to partial views are *crash-consistent*

V2 is the latest, since 7 > 6

NAL Switch (1) – Detect Hot Set

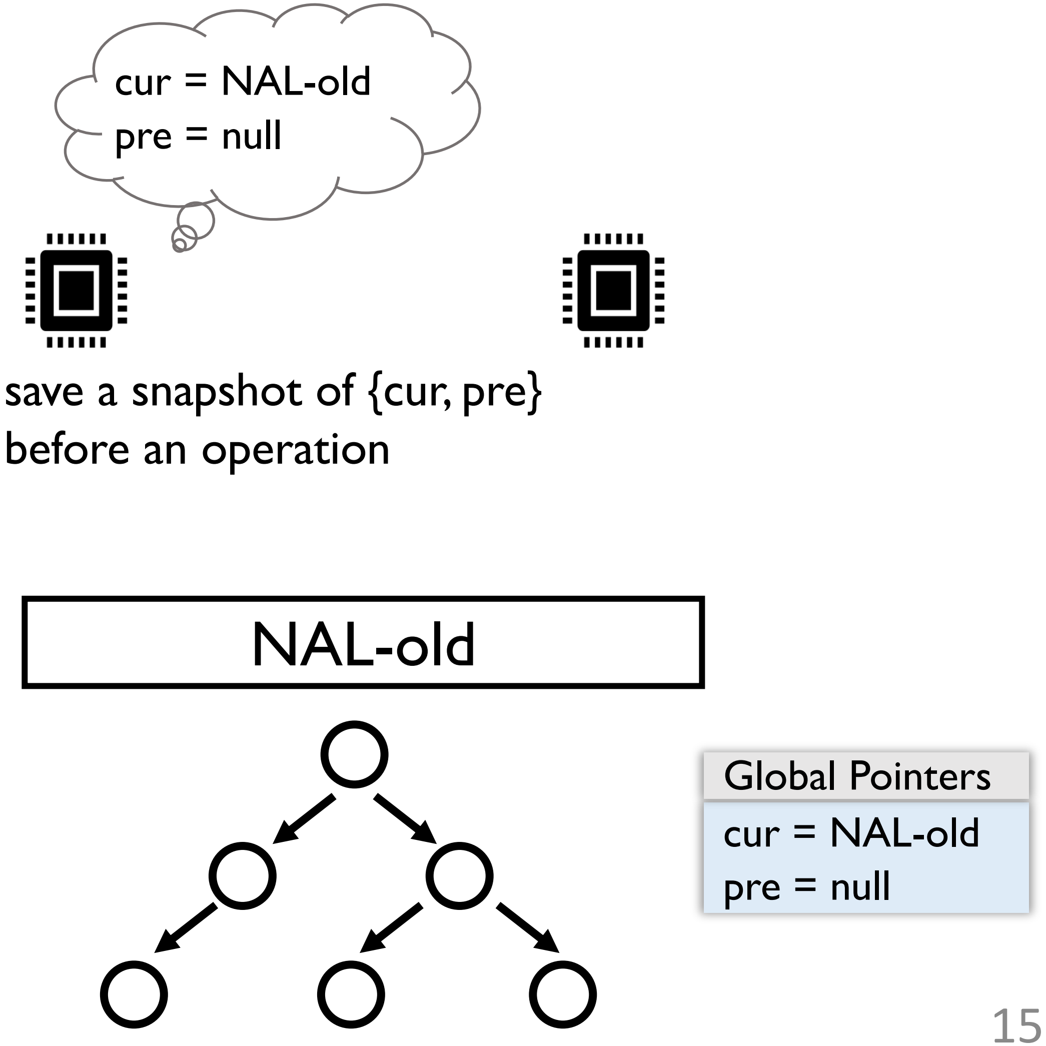
Nap switches to a new NAL when detecting a new hot set



- Access threads publish their access patterns using sampling
- A dedicated *switch thread* produces current hot set
  - a count-min sketch estimates frequency of keys
  - a min heap records the current hot set

NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

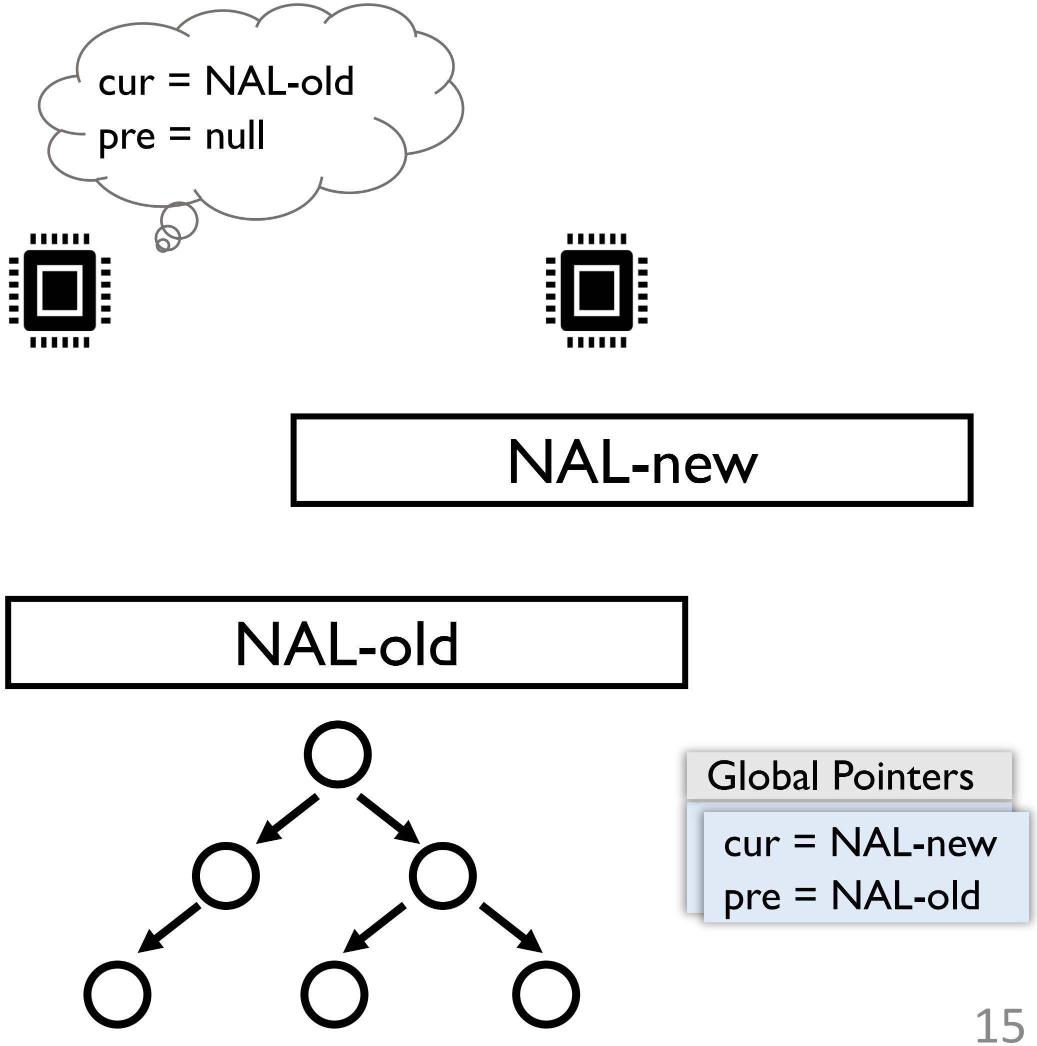


NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked

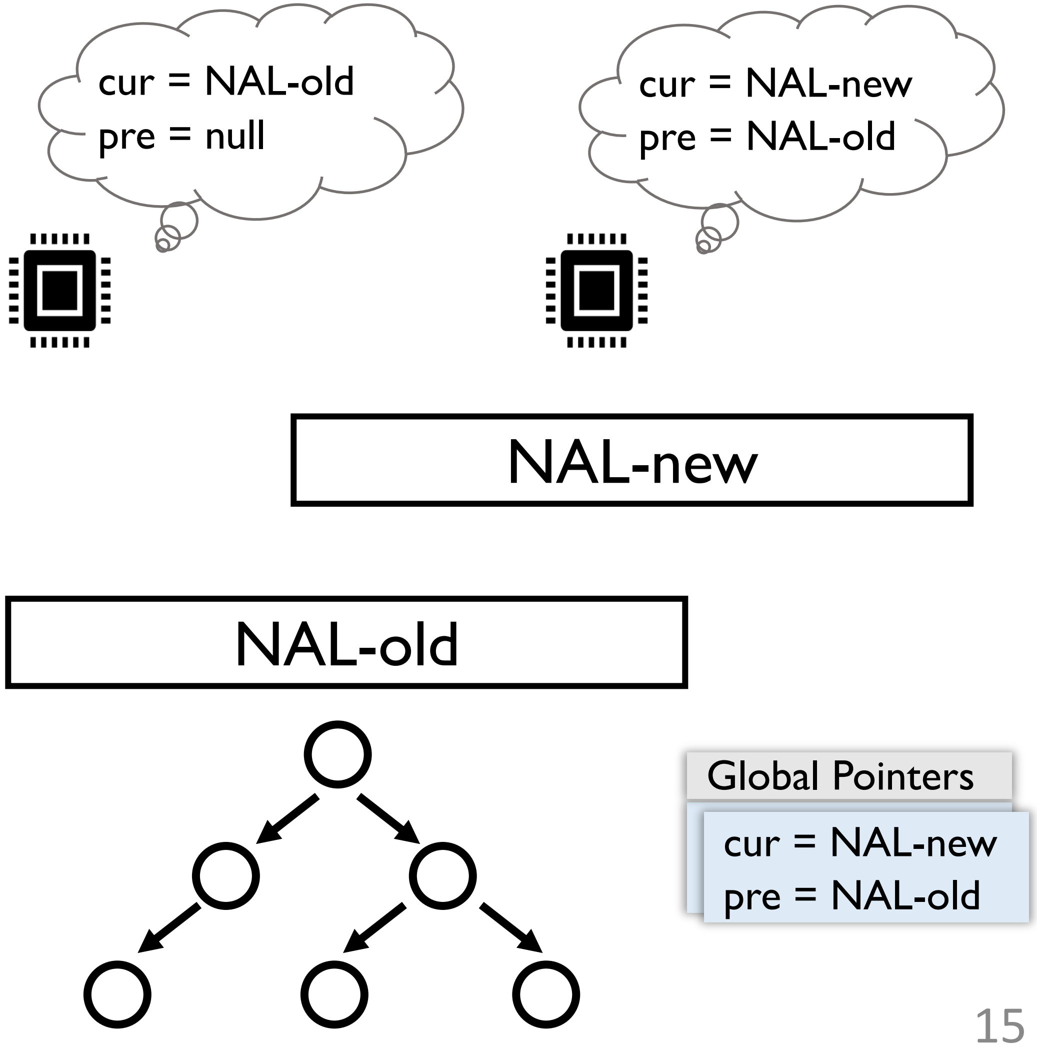


NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked

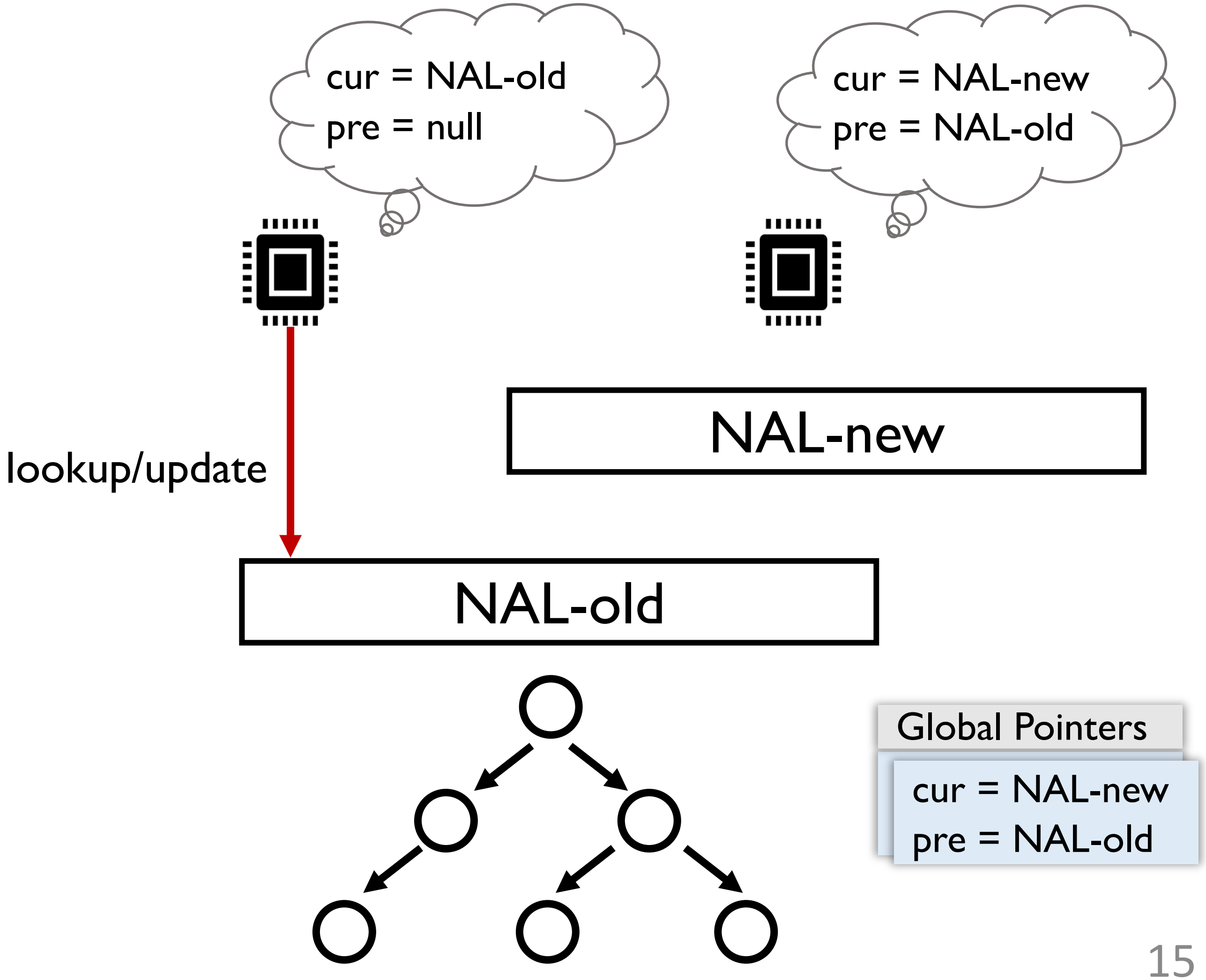


NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked

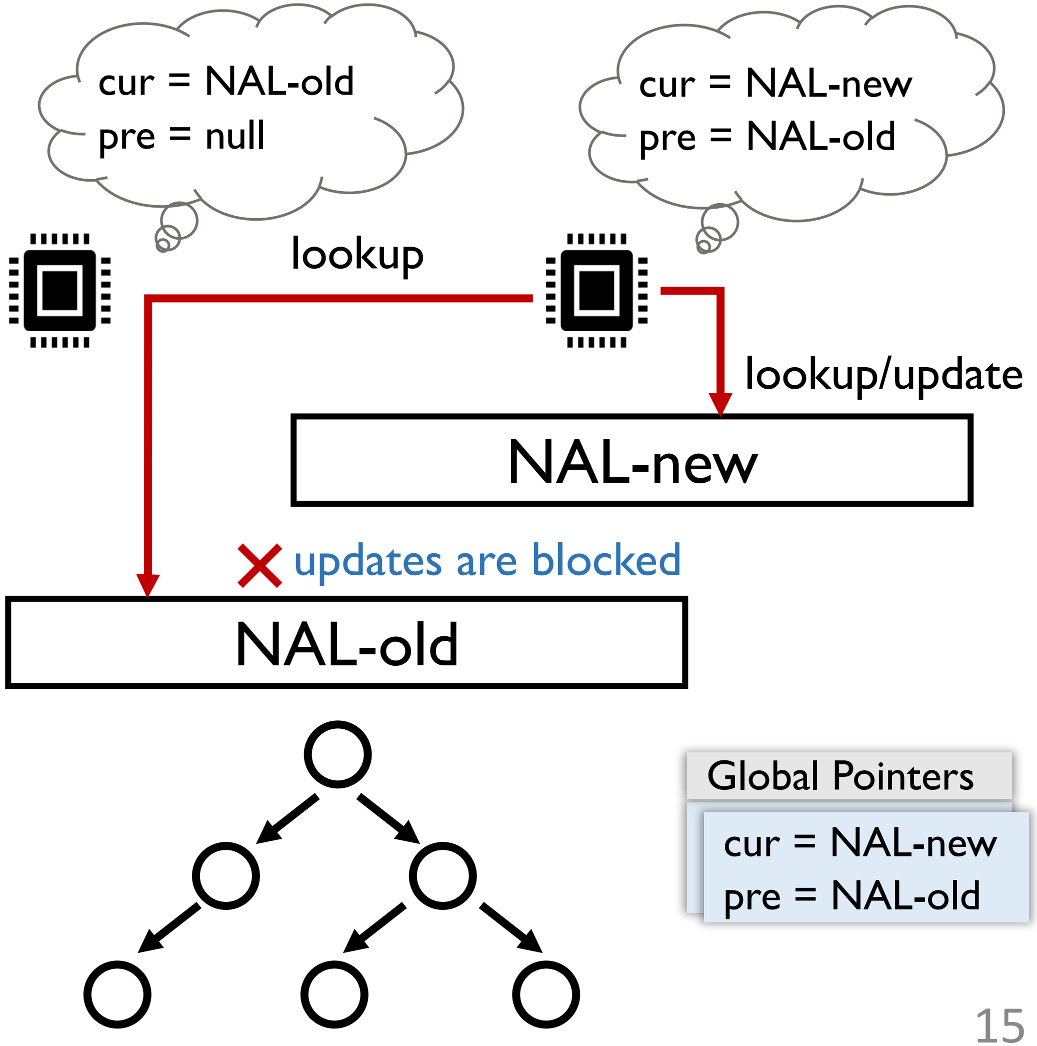


NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked



NAL Switch (2) – Three Phase Switch

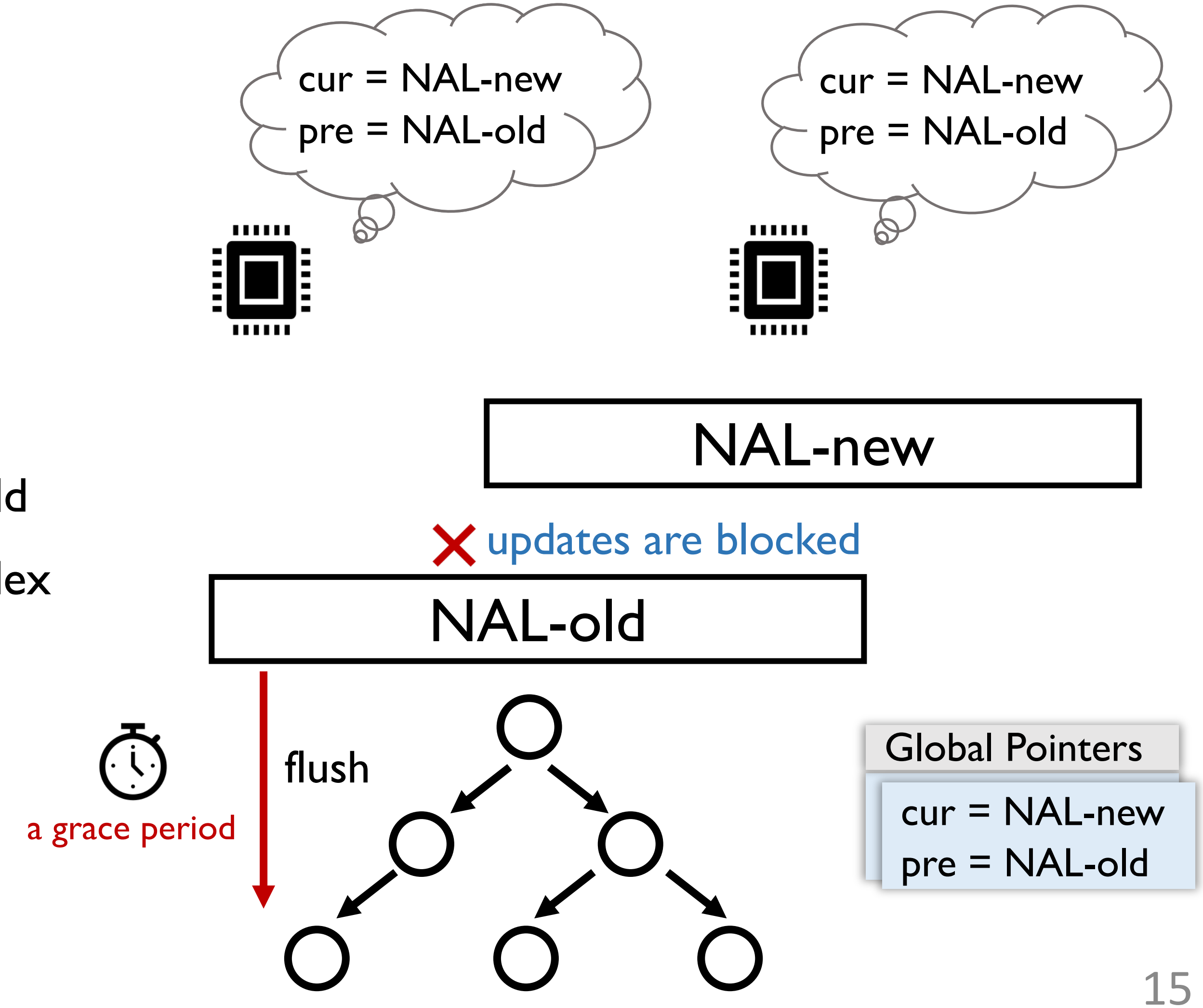
A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked

Phase 2: Flush NAL-old

- wait for a grace period, to ensure no updates on NAL-old
- flush items from global view of NAL-old into raw PM index



NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

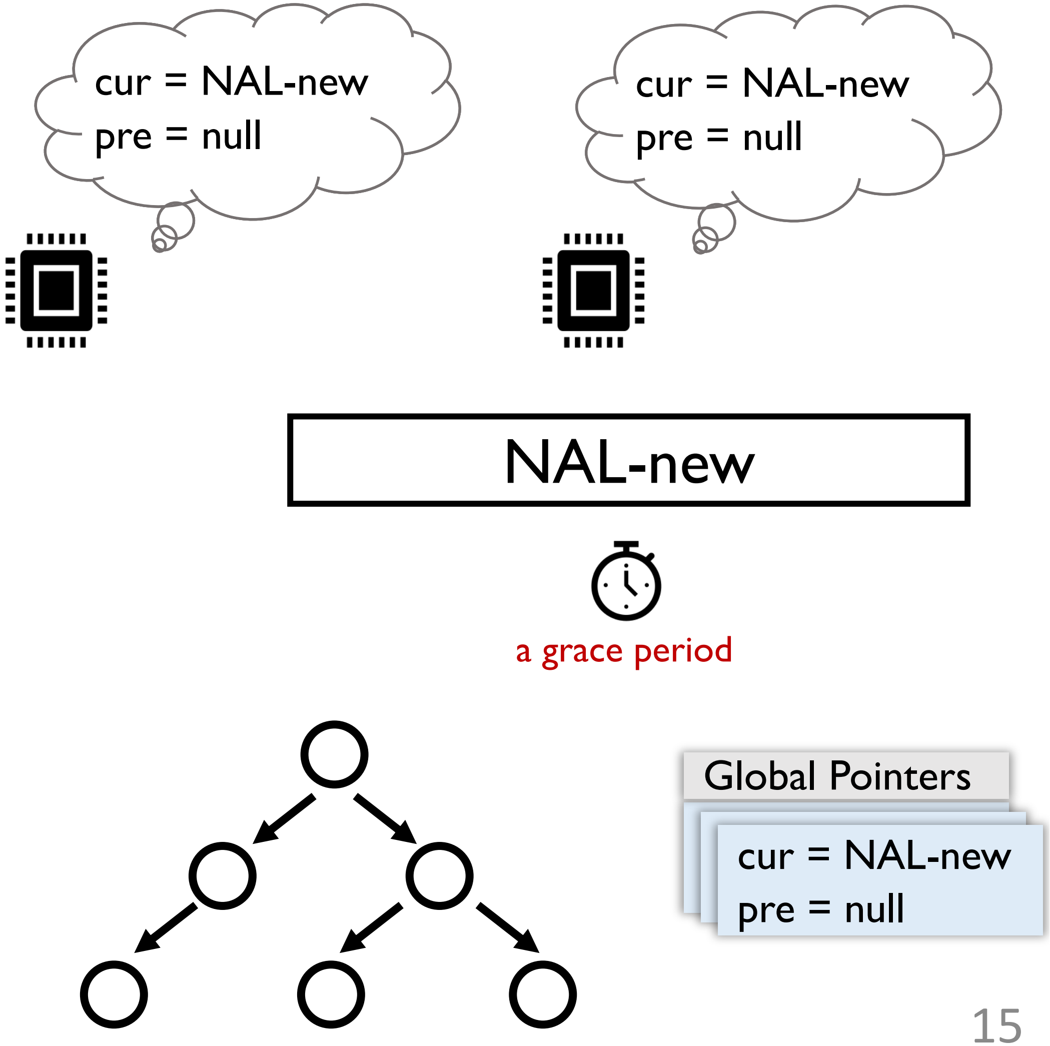
- change global pointers, making NAL-new visible
- some threads see concurrent switch, others don't
- if seeing switch, updates to NAL-old are blocked

Phase 2: Flush NAL-old

- wait for a grace period, to ensure no updates on NAL-old
- flush items from global view of NAL-old into raw PM index

Phase 3: Recycle NAL-old

- set global pointer *pre* to null
- wait for a grace period, to ensure no lookups on NAL-old
- release PM/DRAM space of NAL-old



NAL Switch (2) – Three Phase Switch

A grace-period-based method to minimize blocking

Phase 1: Initialize and install a new NAL

Phase 2: Flush NAL-old

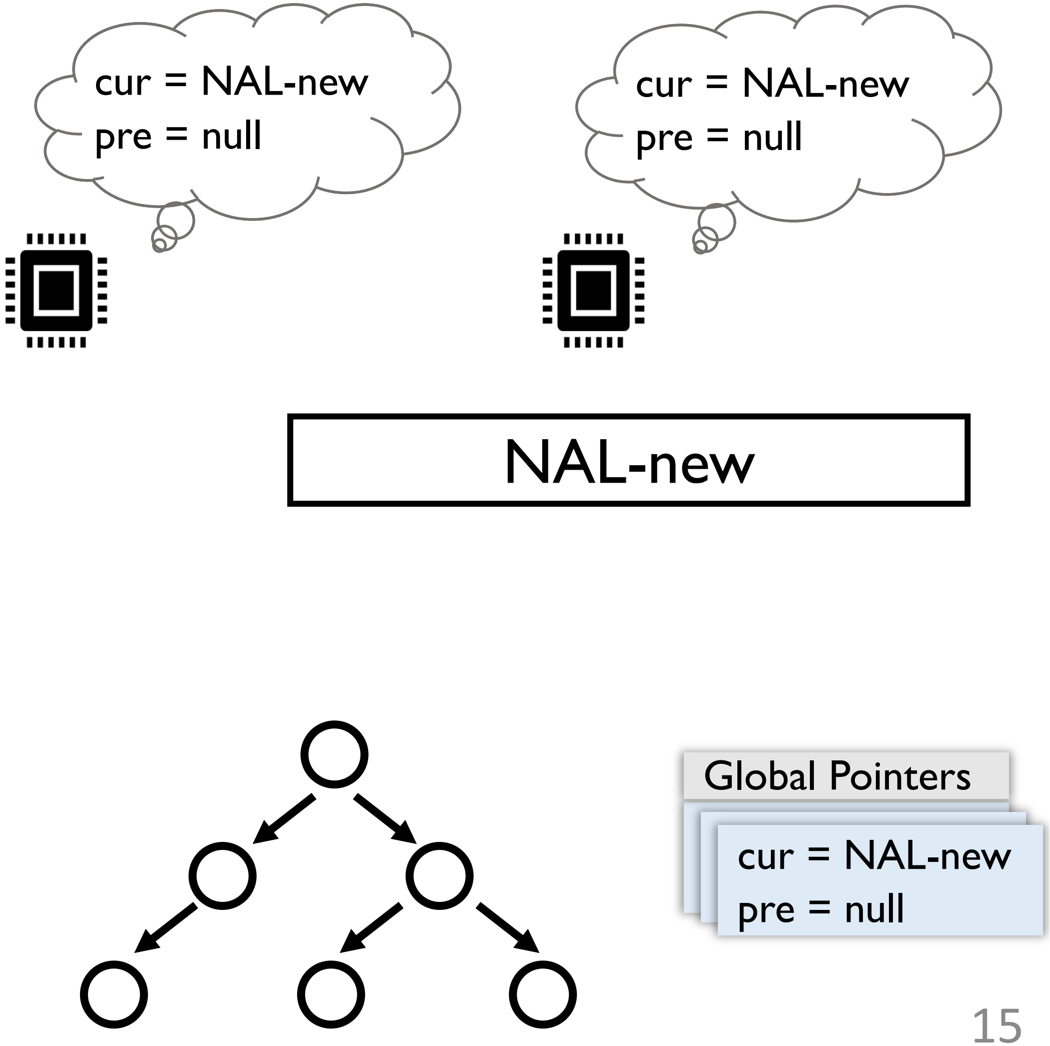
Phase 3: Recycle NAL-old

Only updates to NAL-old during phase 2 are blocked

Such a blocking has only a *small impact* since:

1) Current hot items are handled by NAL-new

2) Flushing NAL-old is data-race-free



Outline

- Background & Motivation
- Nap – a Black-Box Approach to NUMA-Aware PM Indexes
- Results
- Takeaways

Experimental Setup

Hardware Platform

CPU	**4** * Intel Xeon Gold 6240M CPUs, **18** cores per node
PM	**12** * 128GB Optane DIMMs (3 per node)
DRAM	**12** * 32GB DDR4 DIMMs (3 per node)

Experimental Setup

Hardware Platform

CPU	**4** * Intel Xeon Gold 6240M CPUs, **18** cores per node
PM	**12** * 128GB Optane DIMMs (3 per node)
DRAM	**12** * 32GB DDR4 DIMMs (3 per node)

Converted PM indexes

- Hash tables: CCEH (FAST'19), P-CLHT (SOSP'19), Clevel (ATC'20)
- Trees: FAST_FAIR (FAST'18), P-Masstree (SOSP'19)

Experimental Setup

Hardware Platform

CPU	**4** * Intel Xeon Gold 6240M CPUs, **18** cores per node
PM	**12** * 128GB Optane DIMMs (3 per node)
DRAM	**12** * 32GB DDR4 DIMMs (3 per node)

Converted PM indexes

- Hash tables: CCEH (FAST'19), P-CLHT (SOSP'19), Clevel (ATC'20)
- Trees: FAST_FAIR (FAST'18), P-Masstree (SOSP'19)

Nap's Setting

- Hot set size: 100K items

Experimental Setup

Hardware Platform

CPU	**4** * Intel Xeon Gold 6240M CPUs, **18** cores per node
PM	**12** * 128GB Optane DIMMs (3 per node)
DRAM	**12** * 32GB DDR4 DIMMs (3 per node)

Converted PM indexes

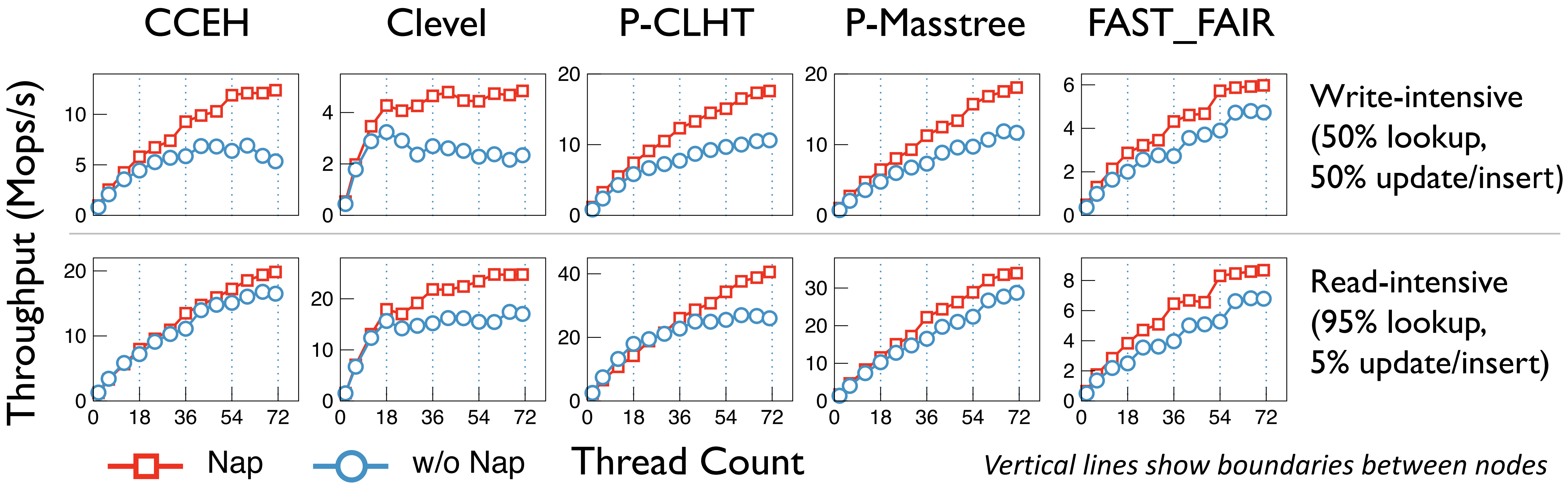
- Hash tables: CCEH (FAST'19), P-CLHT (SOSP'19), Clevel (ATC'20)
- Trees: FAST_FAIR (FAST'18), P-Masstree (SOSP'19)

Nap's Setting

- Hot set size: 100K items

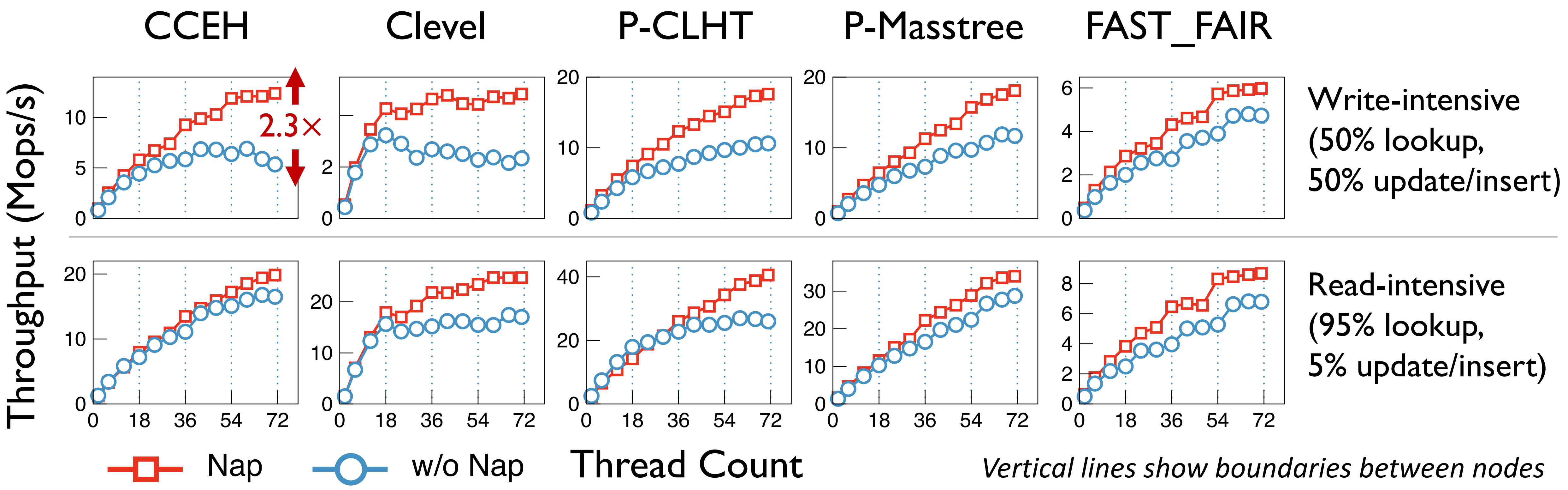
Benchmark: YCSB-like, Zipfian 0.99, 15-byte keys and 8-byte values

Overall Performance



Vertical lines show boundaries between nodes

Overall Performance

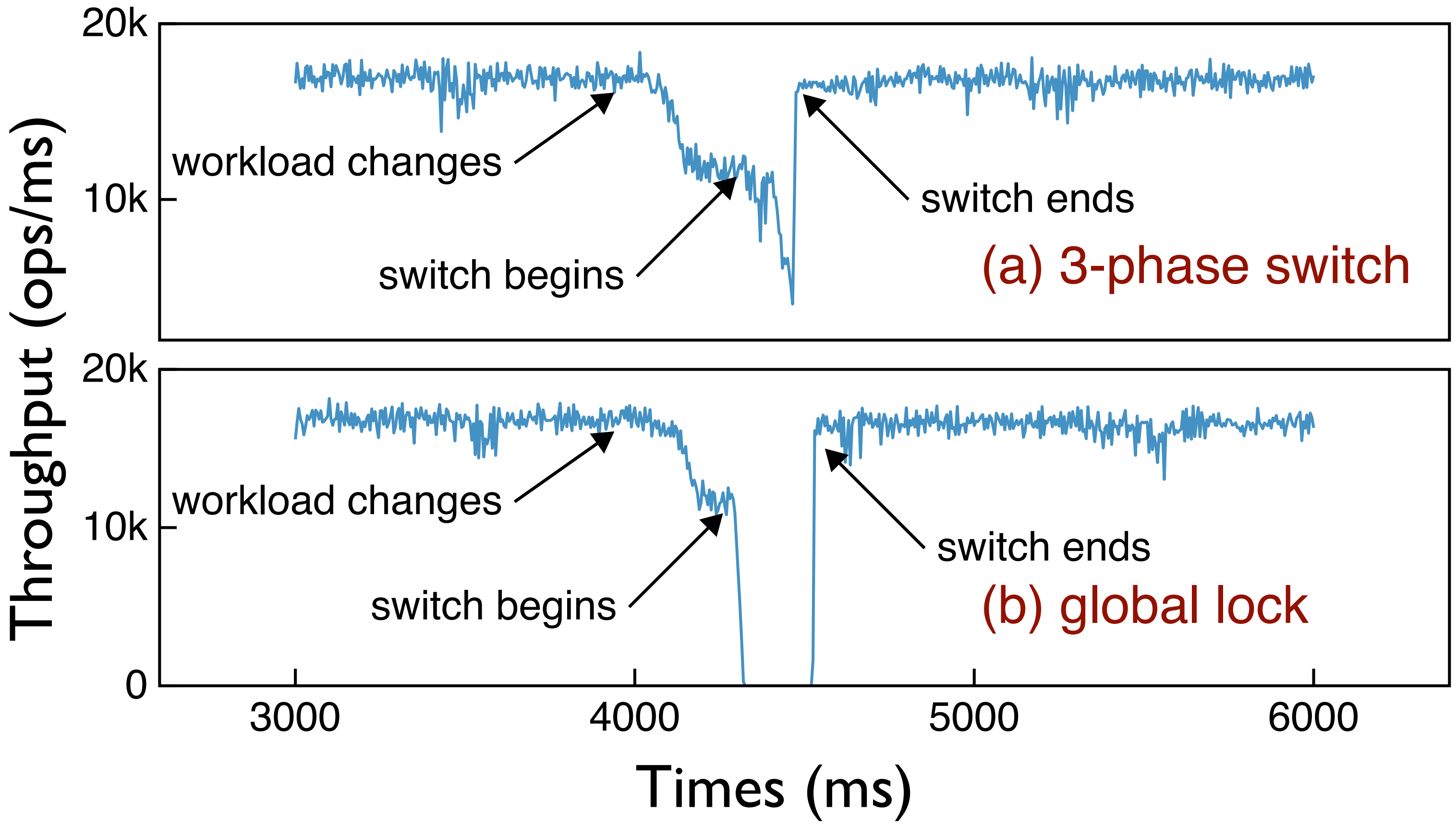


Vertical lines show boundaries between nodes

Nap-converted indexes yield much better scalability under multi-node servers (up to 2.3X)

- NAL absorbs 45% - 54% operations (hot accesses)
- In NAL, partial views eliminate remote PM writes and global view eliminates remote PM reads

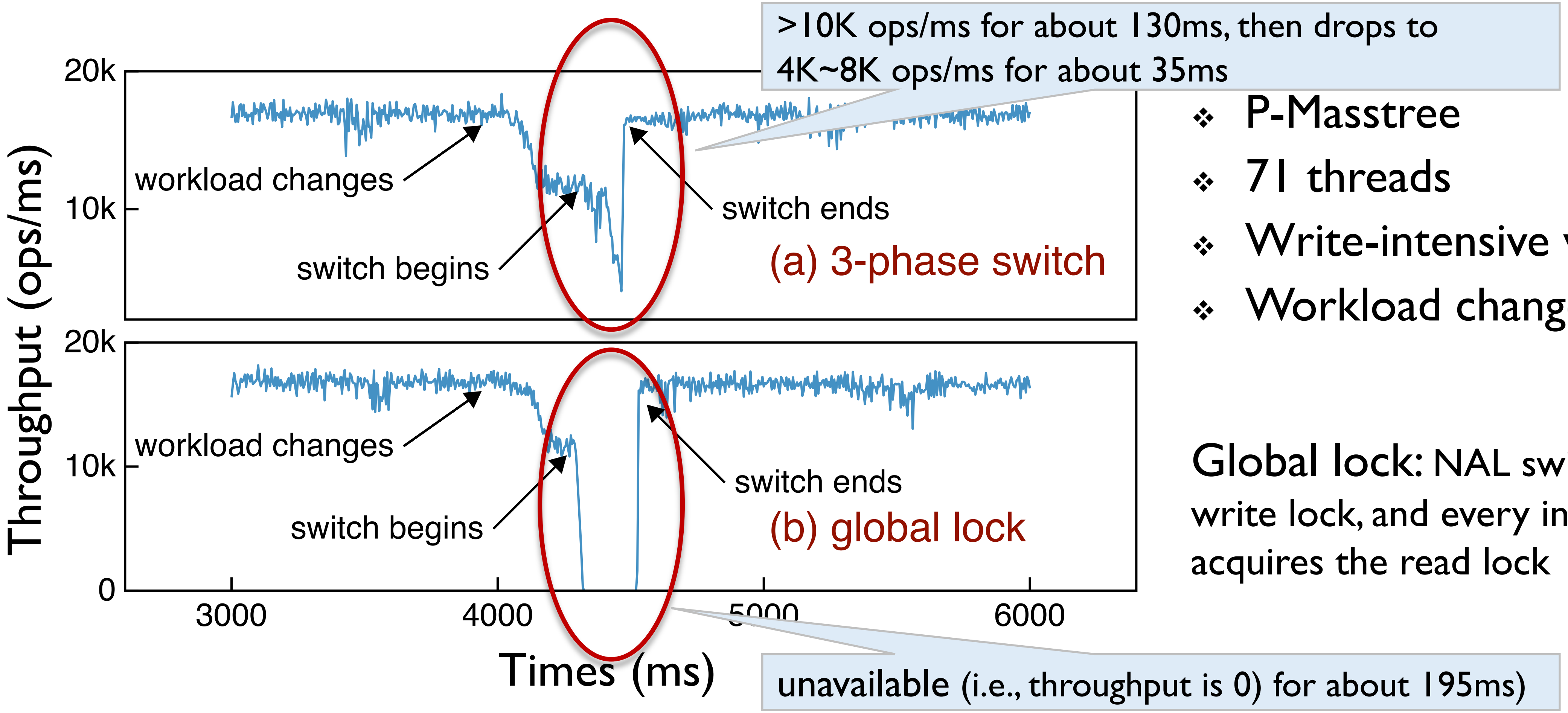
Dynamic Workloads



- P-Masstree
- 71 threads
- Write-intensive workloads
- Workload changes at time 4s

Global lock: NAL switch acquires the write lock, and every index operation acquires the read lock

Dynamic Workloads



- P-Masstree
- 71 threads
- Write-intensive workloads
- Workload changes at time 4s

Global lock: NAL switch acquires the write lock, and every index operation acquires the read lock

Nap is robust enough to react to dynamic workloads quickly without sacrificing availability

Overheads of Nap

- Throughput degradation under scan-intensive workloads
  - 3% for FAST_FAIR, 14% for P-Masstree
- PM and DRAM consumption
  - Less than 70MB when NAL maintains 100K hot items
- Recovery time
  - 300ms ~ 1000ms

Outline

- Background & Motivation
- Nap – a Black-Box Approach to NUMA-Aware PM Indexes
- Results
- Takeaways

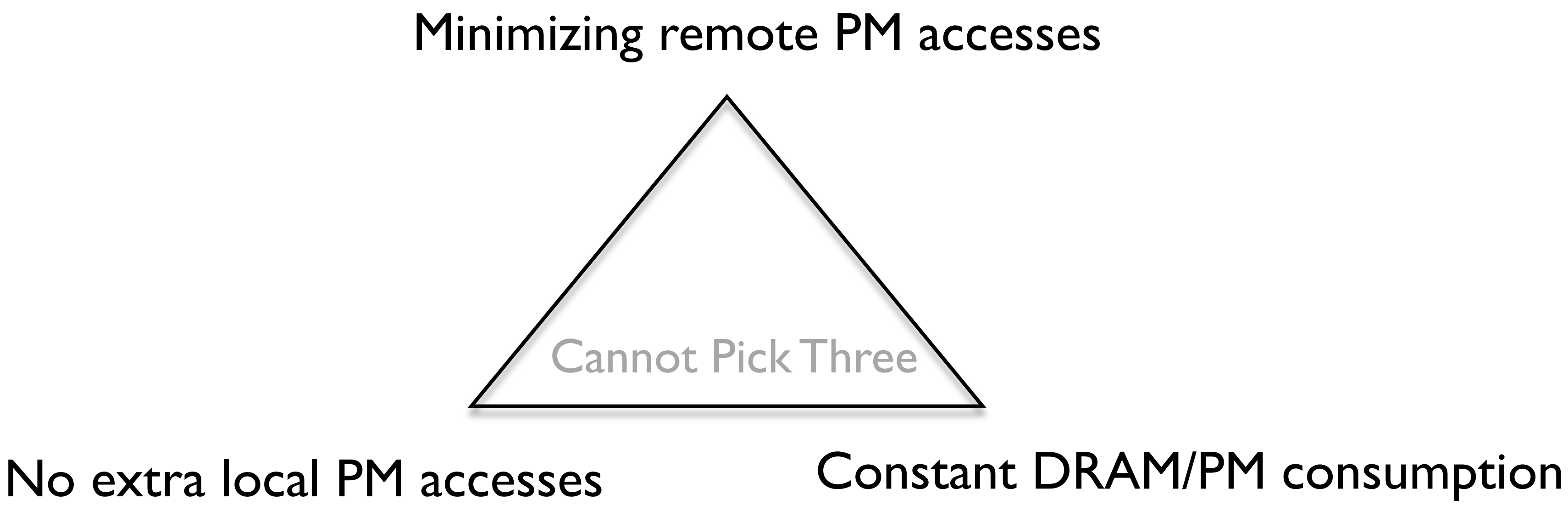
Takeaway 1

A fast NUMA-aware PM index must reduce remote PM accesses

without consuming extra local PM bandwidth

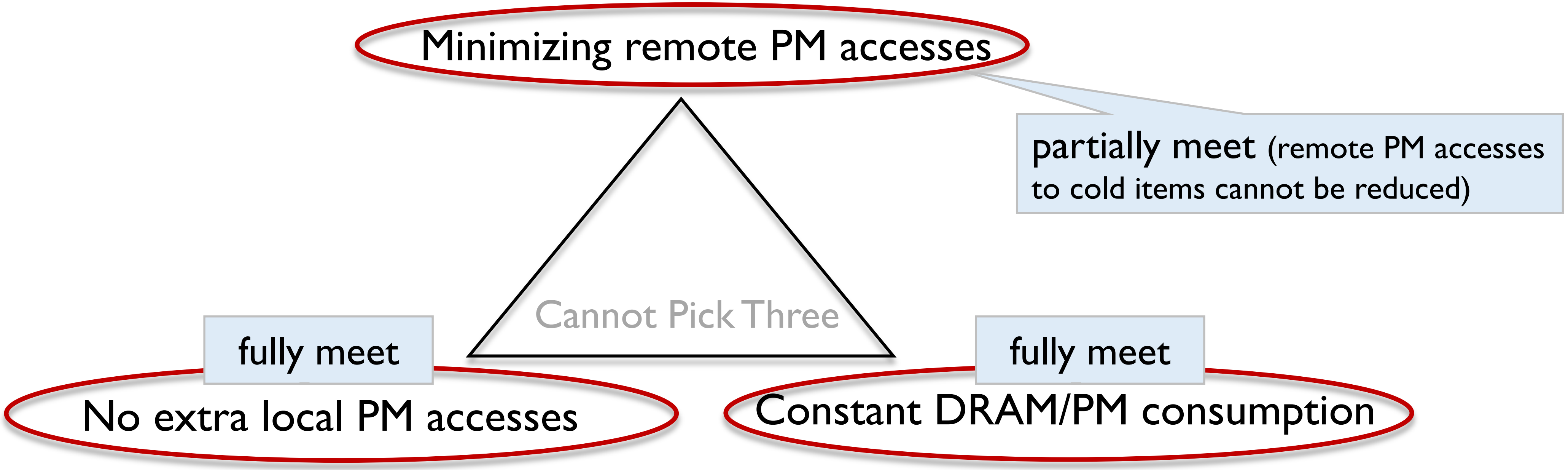
Takeaway 2

A PM index is impossible to be optimal in three aspects at the same time



Takeaway 2

A PM index is impossible to be optimal in three aspects at the same time



Nap achieves a sweet spot by leveraging the characteristics of common skewed workloads with two key ideas:

1) making hot accesses NUMA-aware
2) minimizing PM state synchronization between NUMA nodes

Thanks & QA

Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes

Contact Information: q-wang18@mails.tsinghua.edu.cn