Rethinking Memory Reclamation for Concurrent Data Structures

Ajay Singh

Collaborators: Trevor Brown (University of Waterloo), Michael Spear (Lehigh University) and Ali Mashtizadeh (University of Waterloo)

* Supported by the Hellenic Foundation for Research and Innovation (HFRI) under the "Second Call for HFRI Research Projects to support Faculty Members and Researchers" (project number: 3684).

Modern Data Structures

- Follow optimistic synchronization or non blocking paradigms.
- Permit higher parallelism.
- Widely adopted in open-source software (e.g., Meta's Folly, Linux Kernel).

Key Property: Unsynchronized Reads

- Threads can read from a shared memory location while it is being concurrently modified.

“

Unsynchronized Reads/Traversals in concurrent data structures **enable high scalability** but **Complicate Memory Management!**

”

Example: LockFree Stack [RK Treiber, IBM, 1986]

```
class Node{int data; Node* next;};
Node* Top = nullptr;
```

```
void push (int data) {
    Node* node = new Node(data);
    while (true) {
        Node* t = Top;
        node->next = Top;
        if (CAS(&Top, t, node))
          break;
    }
}
```

```
int pop () {
    while (true) {
        Node* t = Top;
        if (t == nullptr) return EMPTY;

        Node* next = t->next;
        if (CAS(&Top, t, next)) {
            int res = t->data
            delete t; // ???
            return res;
        }
    }
}
```

Pop(): LockFree Stack [RK Treiber, IBM, 1986]

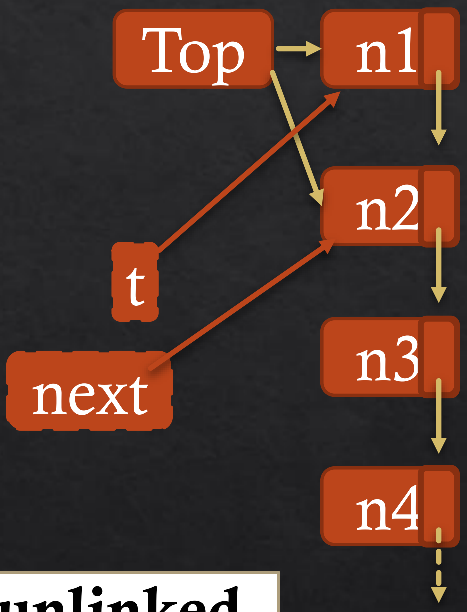
```
int pop () {
    while (true) {
        Node* t = Top;
        if (t == nullptr) return EMPTY;

        Node* next = t->next;
        if (CAS(&Top, t, next)) {
            int res = t->data
            delete t; // ???
            return res;
        }
    }
}
```

Unsynchronized read

Unsynchronized read

Works as long as the unlinked node is not reclaimed

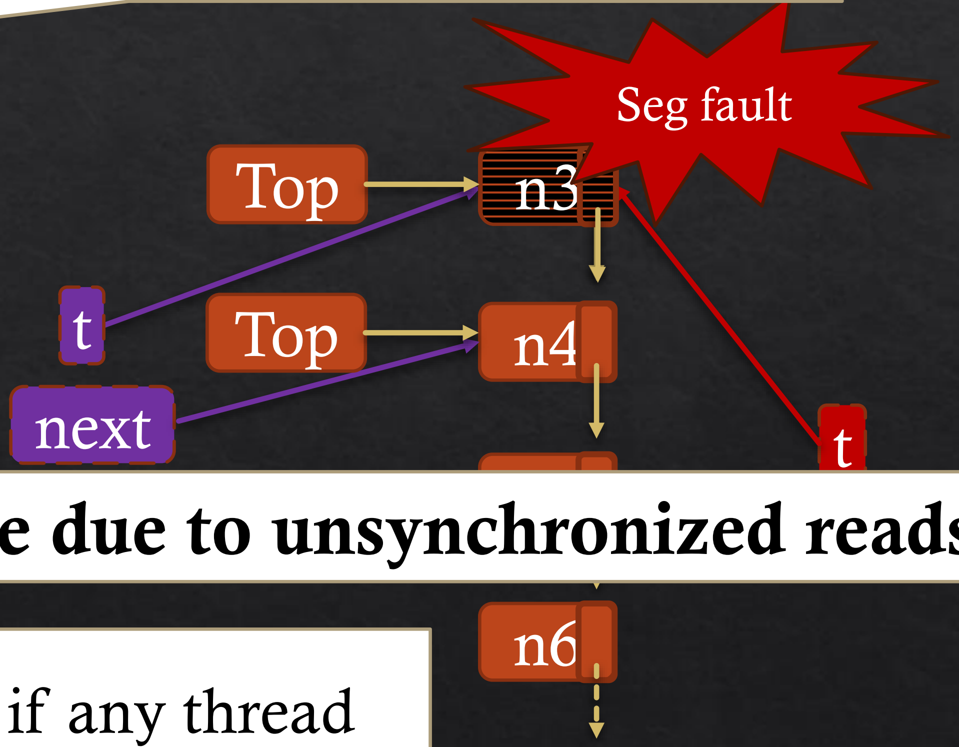


Problem: Read-Reclaim Race

```
int pop () {
    while (true) {
        Node* t = Top;
        if (t == nullptr) return EMPTY;

        Node* next = t->next;
        if (CAS(&Top, t, next)) {
            int res = t->data
            delete t; // ???
            return res;
        }
    }
}
```

<u>Reader</u>: does not know if any thread could concurrently free the node it is accessing.



read-reclaim race due to unsynchronized reads!

<u>Reclaimer</u>: does not know if any thread can access the node it is trying to free.

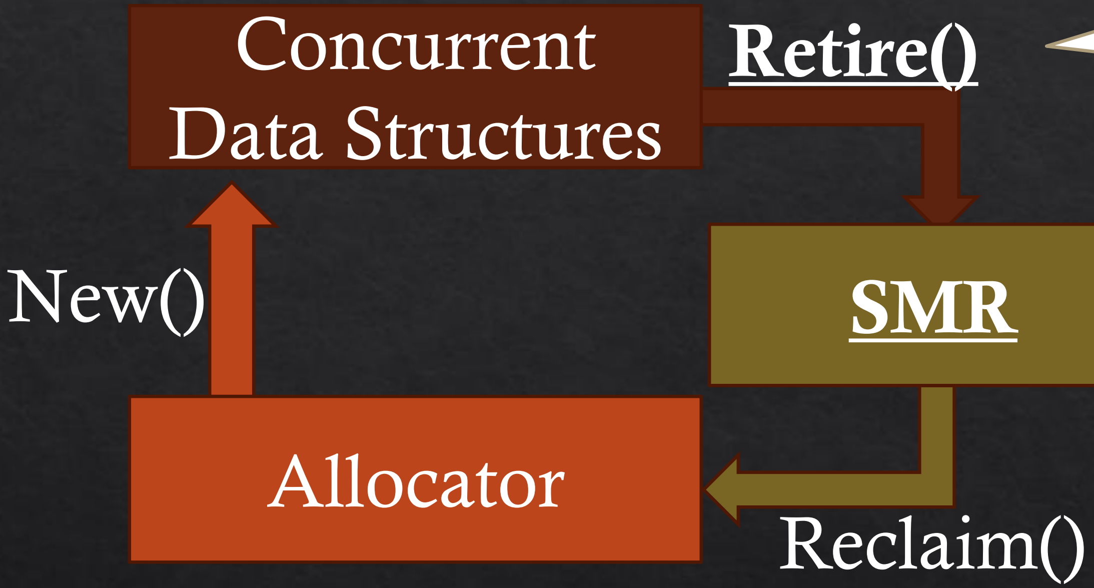
Solution: Safe Memory Reclamation (SMR)

Synchronize readers and **reclaimers** to **decide a safe time to reclaim** unlinked nodes thereby resolving errors due to read-reclaim races.

Readers learn whether a node is safe to access i.e. will not be concurrently reclaimed.

Reclaimers learn whether the node they have unlinked is safe to be reclaimed, i.e., no thread holds a reference to the node.

Key Aspect: Unlink→Retire→Reclaim



retired nodes: unlinked but not yet reclaimed (garbage).
retire set (retSet): per thread set to temporarily store retired nodes.

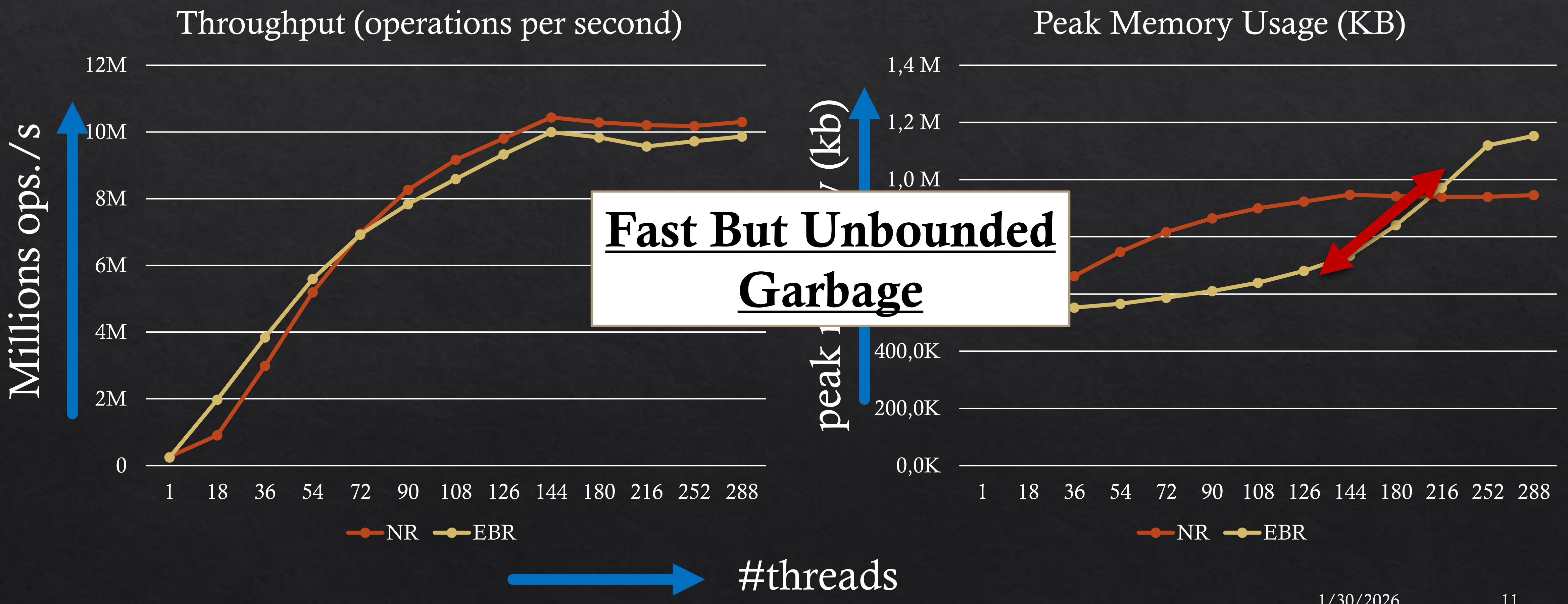
Popular SMR Algorithms

- Epoch Based Reclamation
- Hazard Pointers

(I) EBR: Epoch Based Reclamation

- Assumption:
  - Threads are **Quiescent** (do not access shared nodes) between two consecutive data structures operations.
  - Entry Point: Operations start from an entry point in data structure, e.g., head in lists or root in trees.
- All threads announce when they are quiescent.
- Reclaimers wait for all threads to go quiescent at least once to reclaim retired nodes.

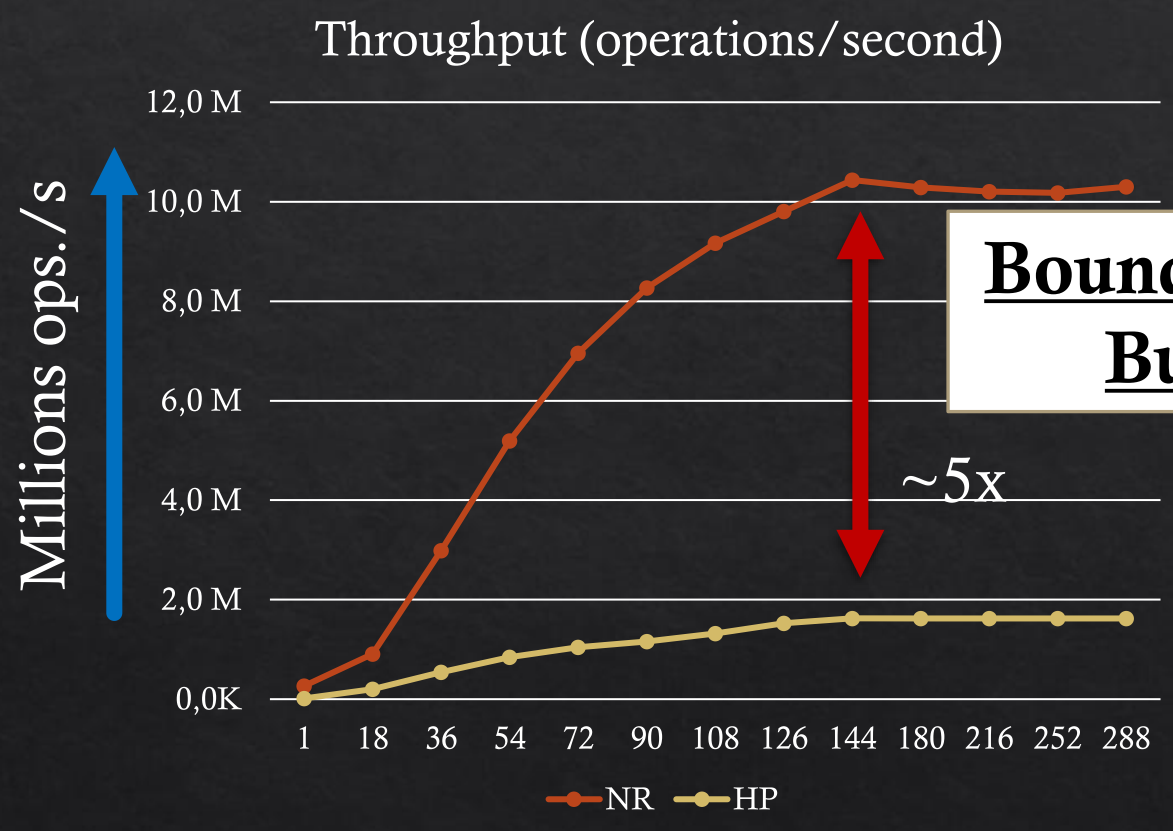
Harris Michael List. 2K size. 100% updates

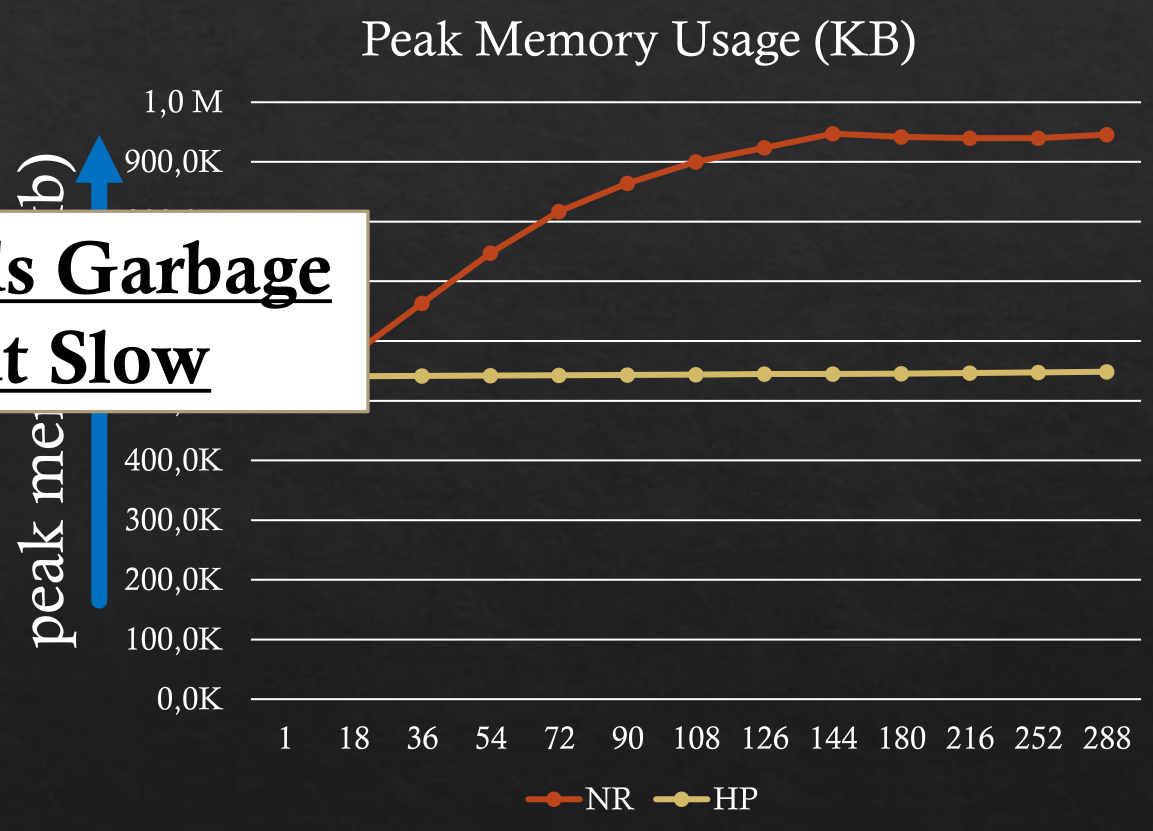


(2) HP : Hazard Pointers

- Readers:
  - Reserve pointers to nodes before accessing.
  - Unreserve pointer after they finish accessing.
- Reclaimers:
  - Scan all published reservations.
  - Reclaim all retired nodes that are not reserved.

Harris Michael List. 2K size. 100% updates





Bounds Garbage But Slow

#threads

- Epoch Based Reclamation: **Fast** but **does not bound garbage**

- Hazard Pointers Reclamation: **Bounds garbage** but **not fast**

- **Ease of Use**?

(II) HP : Complicated to Use

```
int pop () {
    while (true) {
        Node* t = Top;
        if (t == nullptr) {
            unprotect();
            return EMPTY;
        }
        protect(t);
        if (t != Top) {
            unprotect();
            continue;
        }
        Node* next = t->next;
        if (CAS(&Top, t, next)) {
            int res = t->data
            retire t;
            unprotect();
            return res;
}}}
```

mfence

announce

validate

- Identifying hazardous accesses!
- Correctly reserving:
    - Write pointer at SWMR location.
    - Memory Fence
    - Validate reserved pointer.
- Unreserve when operation exits.
- What if validation fails?

EBR : Easy to Use

```
int pop () {
    startOp();
    while (true) {
        Node* t = Top;
        if (t == nullptr) {
           endOp();
           return EMPTY;
        }
        Node* next = t->next;
        if (CAS(&Top, t, next)) {
            int res = t->data
            retire t;
            endOp();
            return res;
}}}
```

- Announce quiescence:
  - startOp():
  - endOp():

Algorithms with one trade-off or Another

- Pointer Reservation: HP, PTB, HP++ — Per-read overhead
- Epoch Reservation: IBR, HE — Change in memory layout of nodes & Weaker Bound on Garbage.
- Epoch Based: EBR, RCU, DEBRA — Unbounded garbage
- Optimistic Access: OA, AOA, FA, VBR — Custom allocator & change in memory layout.
- Hybrid: TS, FS, Cadence — Compiler or architecture dependent

Three Problems Three Solutions

- **Problem 1:** Difficult to achieve several desirable properties simultaneously. → Neutralization based reclamation
- **Problem2:** high uneven overhead in Hazard Pointers. → Publish on Ping
- **Problem3:** Deferred reclamation paradigm has drawbacks. → Conditional Access

Problem 1

Difficult to achieve several desirable properties simultaneously.

Desirable Properties in SMRs

- Performance
- Bounded Garbage
- Wide Applicability
- Usability

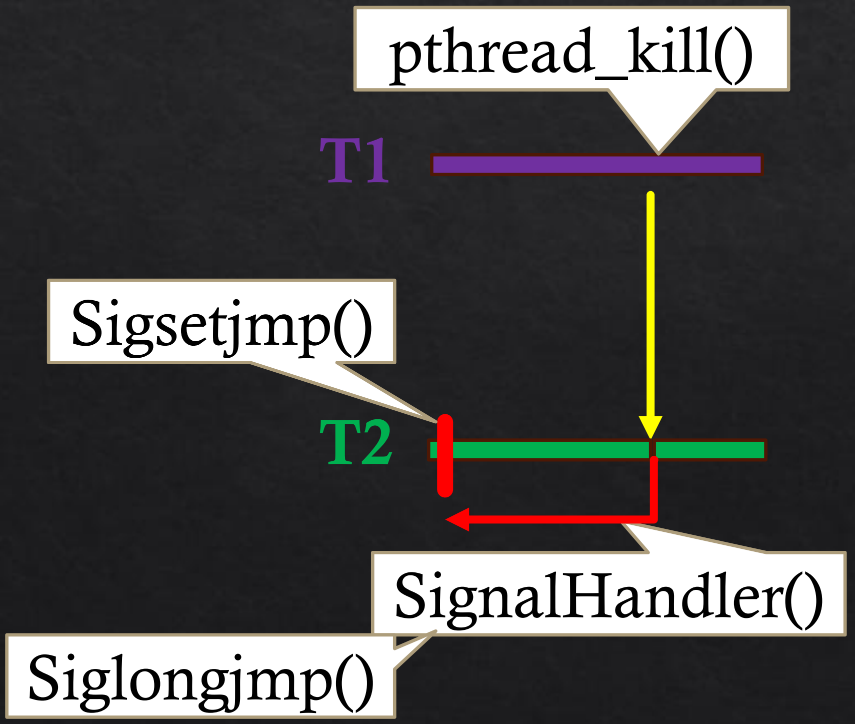
Problem1: No algorithm satisfies all key desirable properties.

Neutralization Based Reclamation (NBR)

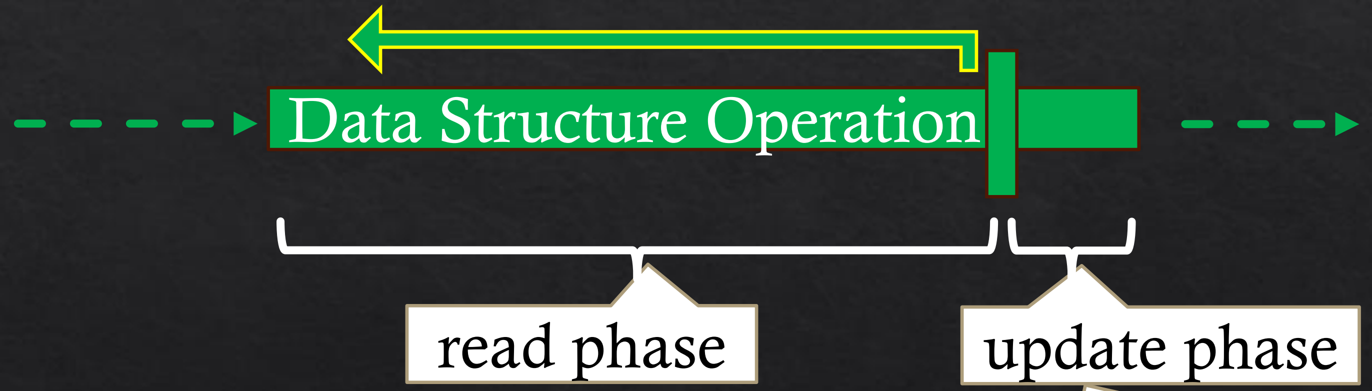
- Threads follow a **neutralization process**.
- A reclaimer, before reclaiming nodes, neutralizes all readers.
- A neutralized reader either :
  - Discards its references, if it hasn't done any updates yet, or
  - Must have reserved the references, if it has executed updates.
- After neutralizing all readers, a reclaimer:
  - Scans all reservations, if any.
  - Reclaims all node **not** reserved by any reader.

Key Components for Neutralization

(I) Posix Signals



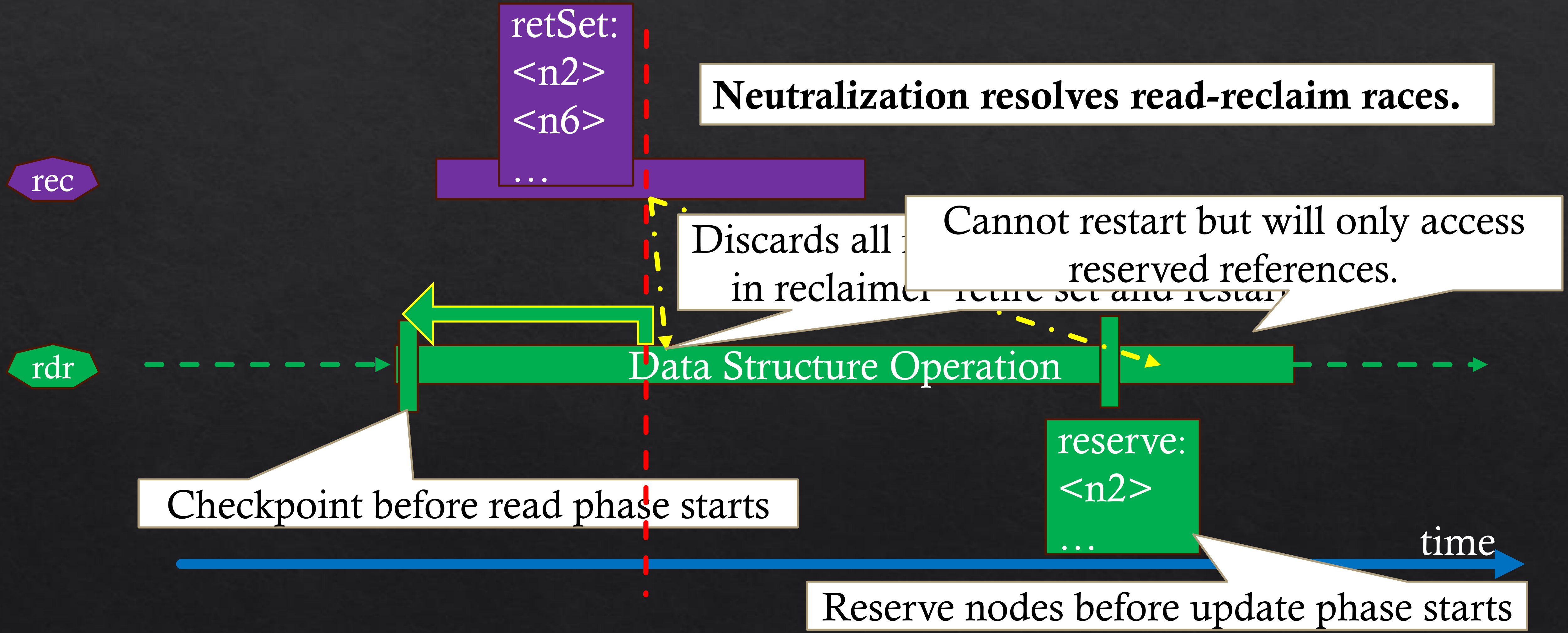
(II) Access-Aware Data Structure



Threads can restart from an entry point in read phase discarding their current references.

Set of nodes needed to execute updates are known beforehand.

Neutralization Process



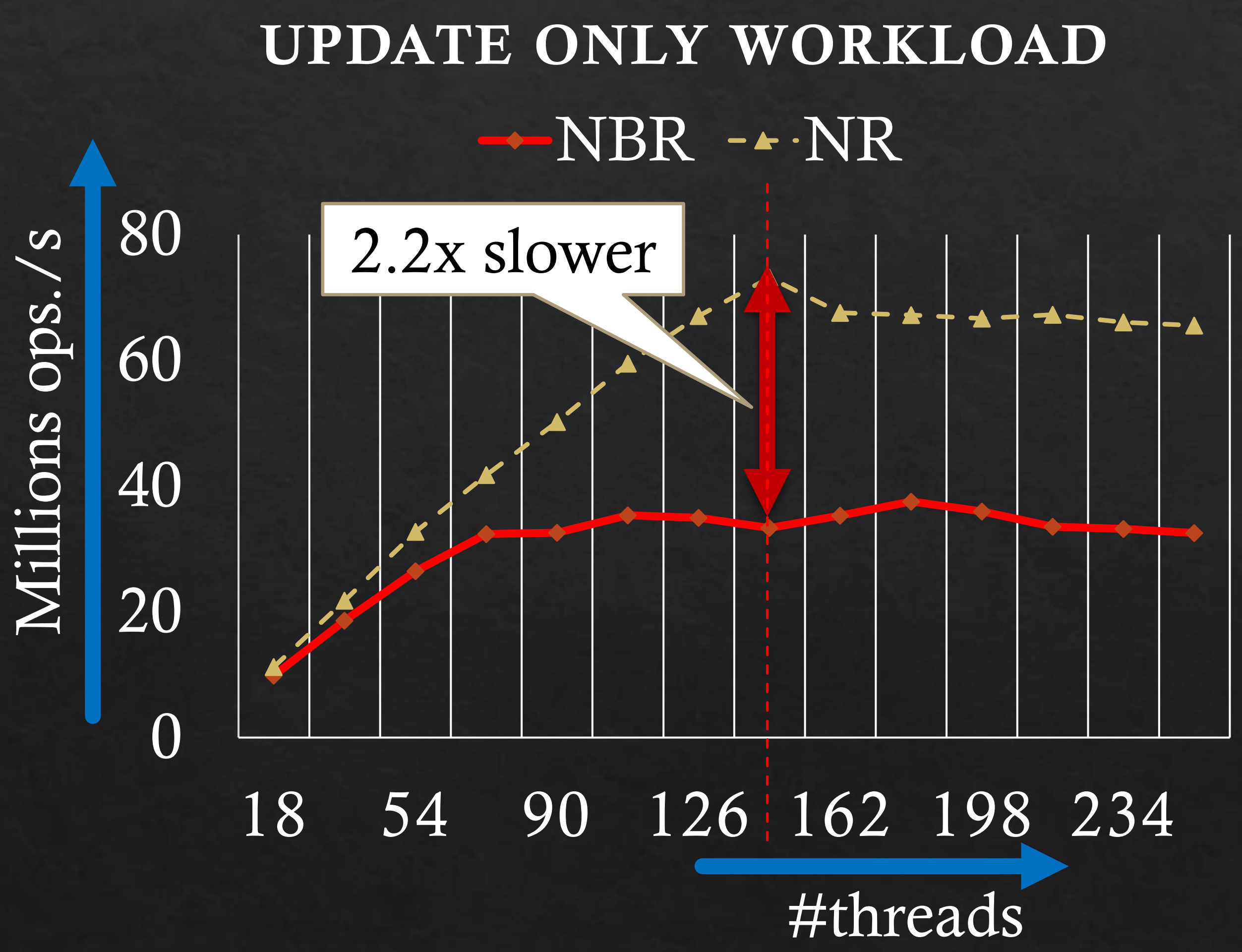
NBR Usability

- NBR Interface:
  - CHECKPOINT()
  - BeginReadPhase()
  - EndReadPhase(…)
  - Retire(…)
- Identify read phase
- Identify write phase and all nodes needed beforehand.

```
void operation(int key) {
    while (true) {
[redacted]
        Node* pred = head;
        Node* curr = pred->next;
        while (curr->key < k) {
            pred = curr;
            curr = curr->next;
        }
[redacted]

        LOCK(&pred->lock); LOCK(&curr->lock);
        if (validate(pred, curr)) {
            // do update
        }
        UNLOCK(&curr->lock); UNLOCK(&pred->lock);
}
```

Search Tree Throughput with NBR



<u>Overhead: Sending too many neutralizing signals</u>

O(N^2) signals for N threads to reclaim exactly once.

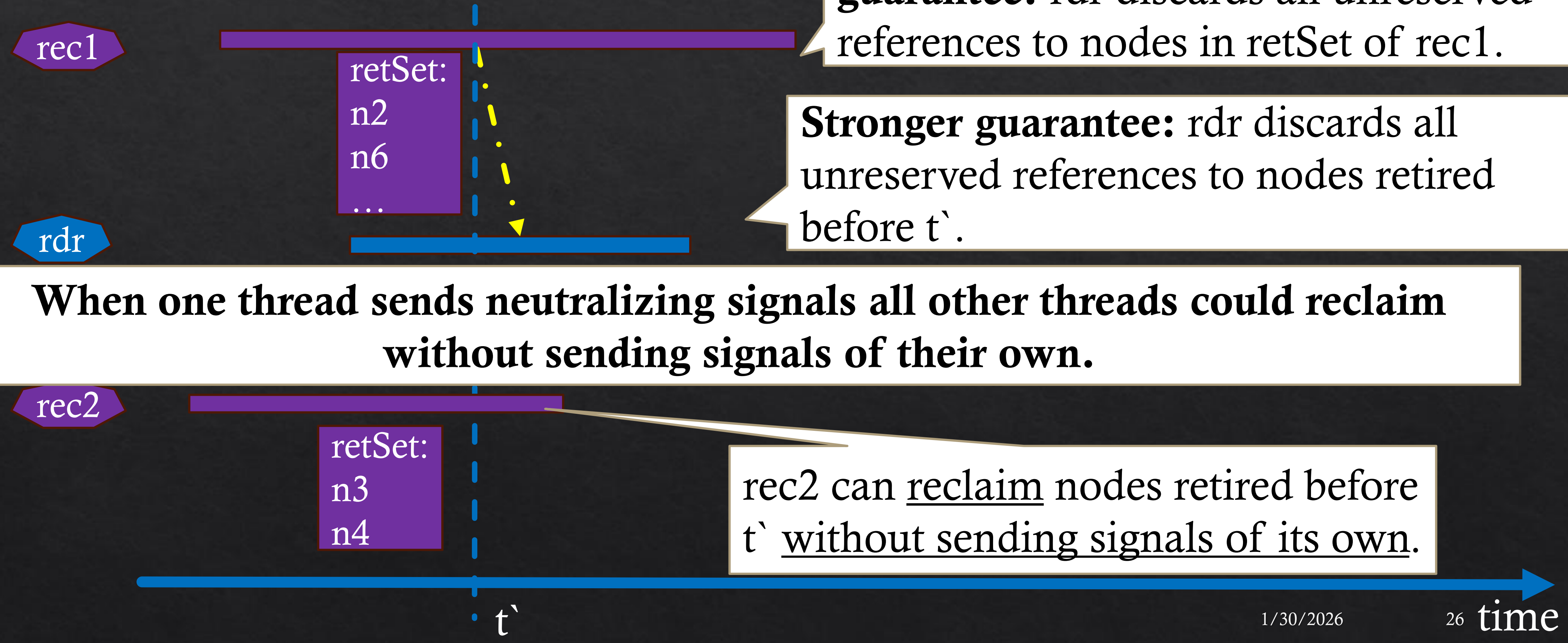
Can we reduce the number of signals and thus eliminate signaling overhead **?**

Observation in NBR

guarantee: rdr discards all unreserved references to nodes in retSet of rec1.

Stronger guarantee: rdr discards all unreserved references to nodes retired before t`.

When one thread sends neutralizing signals all other threads could reclaim without sending signals of their own.

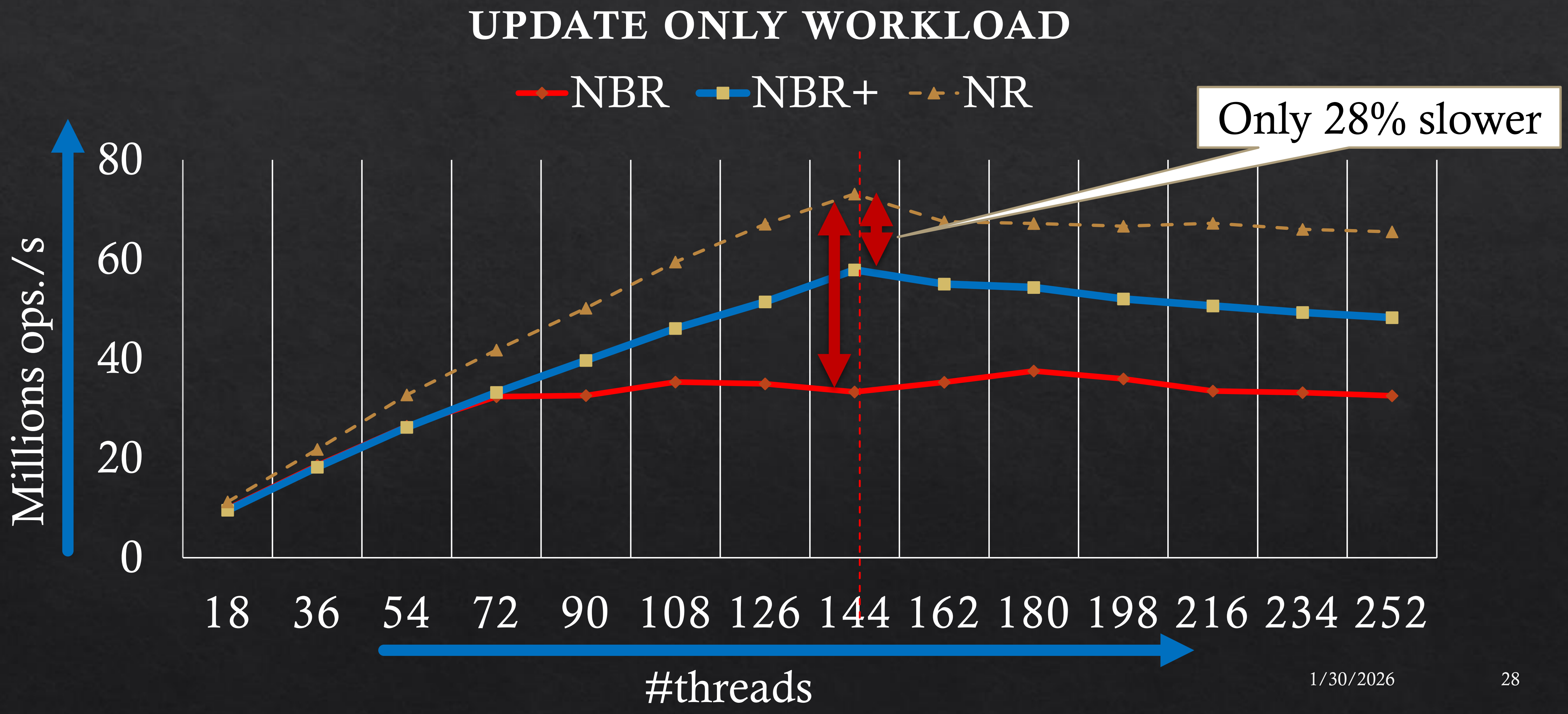


rec2 can reclaim nodes retired before t` without sending signals of its own.

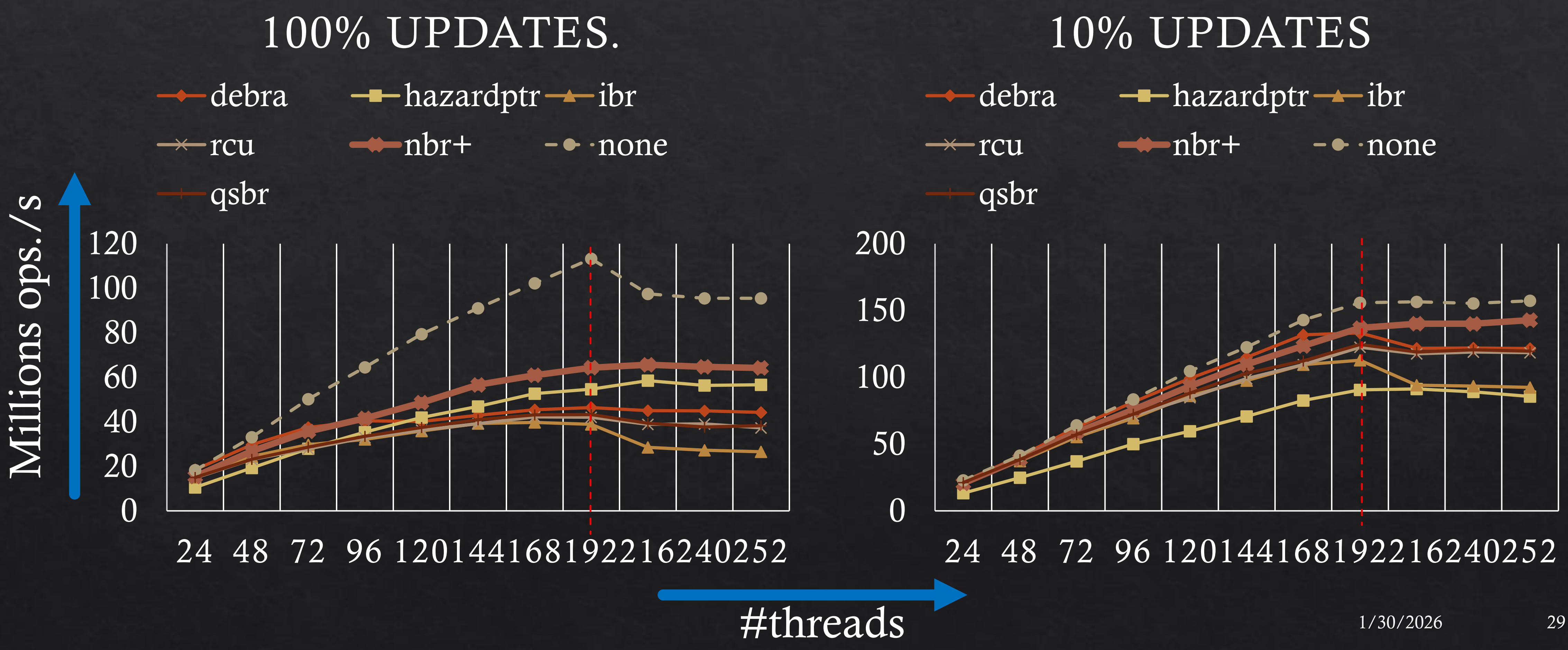
NBR+

- Low and High Watermark: Thresholds for a thread's retSet size to trigger reclamation.
- At High Watermark:
  - Send neutralization signals.
  - Announces start and finish times of the neutralization process.
- At Low Watermark:
  - Monitor if any thread has started and finished neutralization.
  - Reclaim unreserved nodes up to the Low Watermark without sending signals.

Search Tree Throughput with NBR+



External Binary Search Tree



- NBR solves SMR without compromising on desirable properties.
- Leverages POSIX signals.

Problem 2

High uneven overhead in Hazard Pointers

Reservations in Hazard Pointers

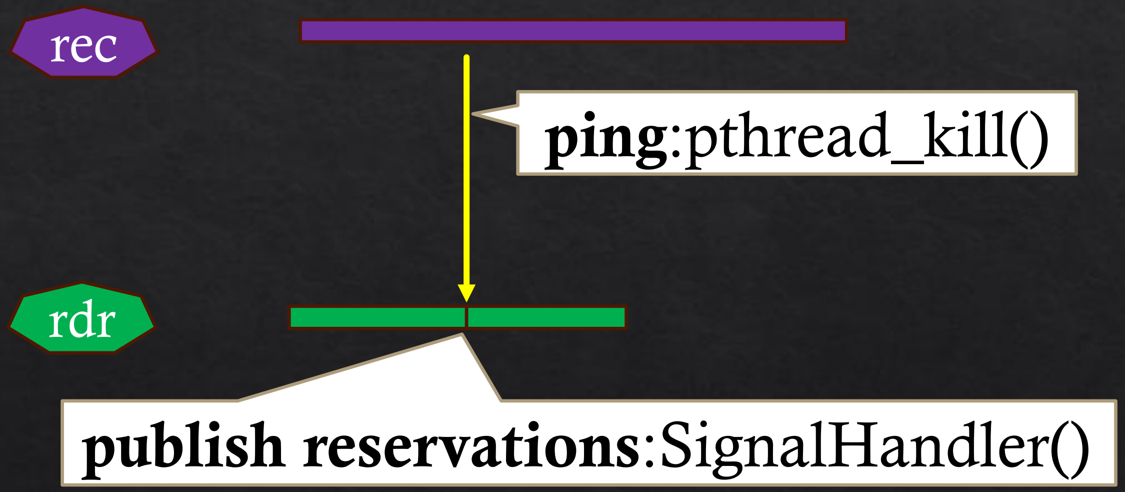
- **Readers**: before accessing, reserve nodes and publish reservations.
- Publishing reservations incurs high overhead for reader due to memory fences.

Problem 2: Uneven overhead
Although threads reclaim infrequently, readers publish reservations frequently (eagerly), incurring high overhead.

Publish Reservations Reactively

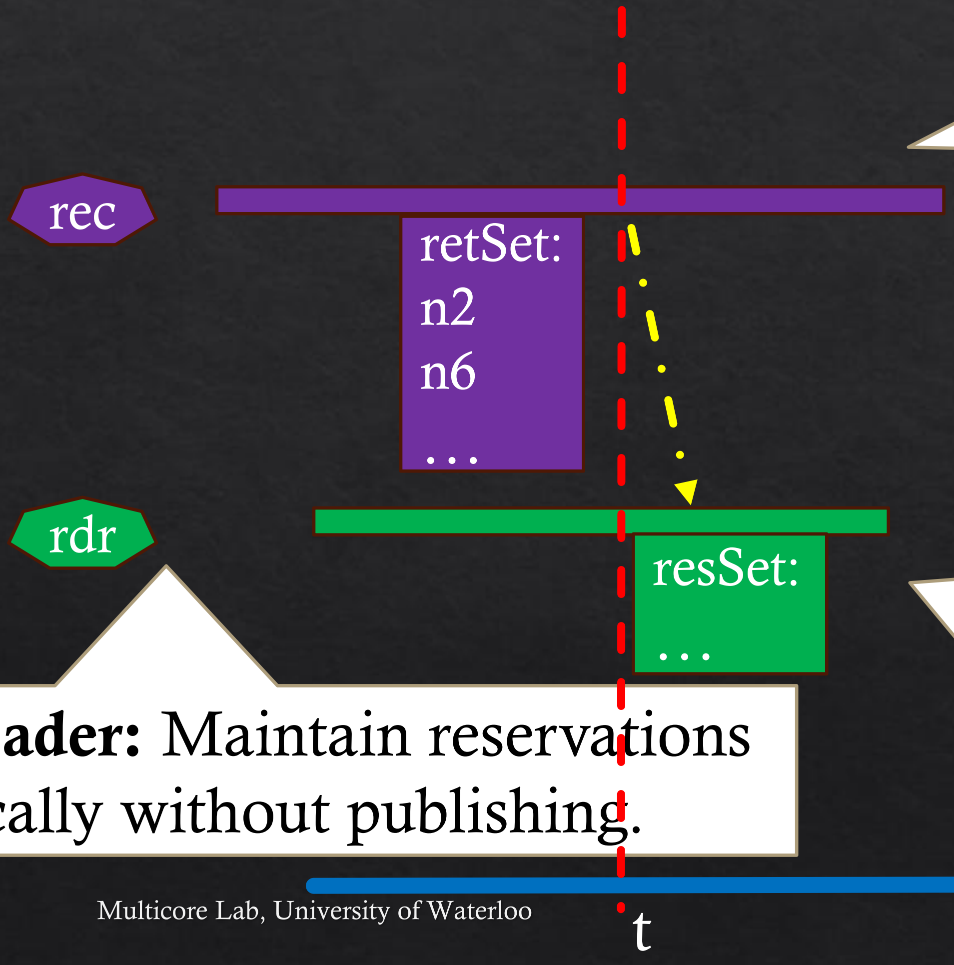
Let readers maintain reservations locally and publish on demand to reclaimers.

Leverage posix signal based inter process communication.



Publish on Ping (POP)

1. Ping the reader to publish reservations, if any.
2. Wait for reservation to be published.
3. Scan and reclaim unreserved.

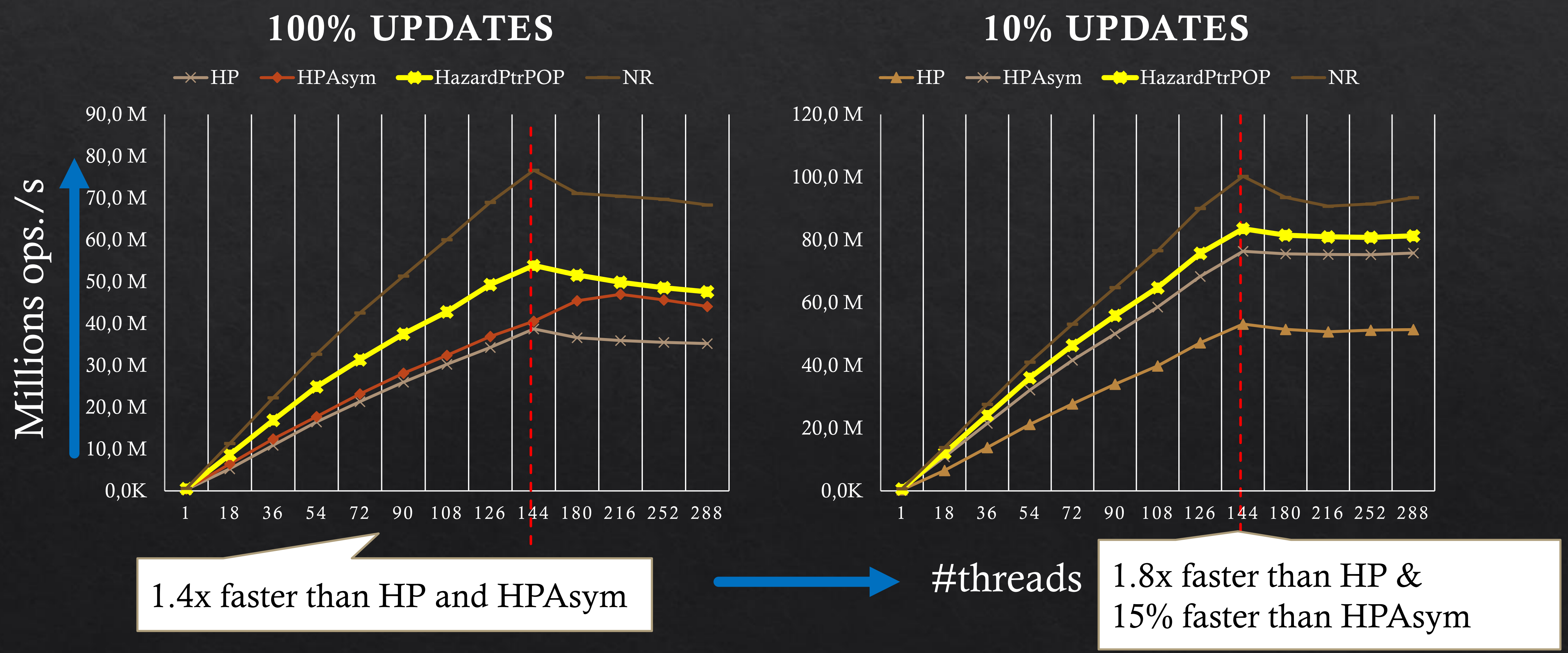


Reader: Maintain reservations locally without publishing.

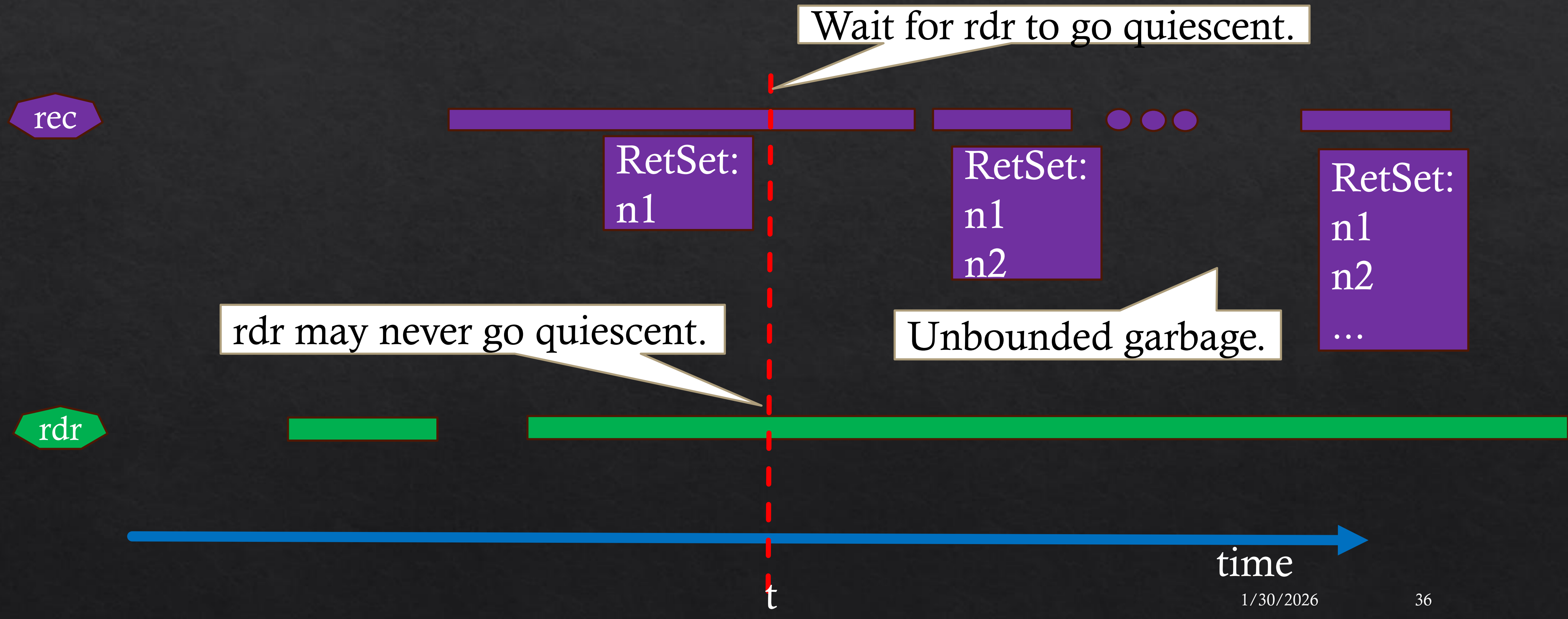
In signal handler:

1. Publish reservations.
2. Indicate to reclaimer that reservation is published.

HazardPtrPOP: Search Tree Throughput

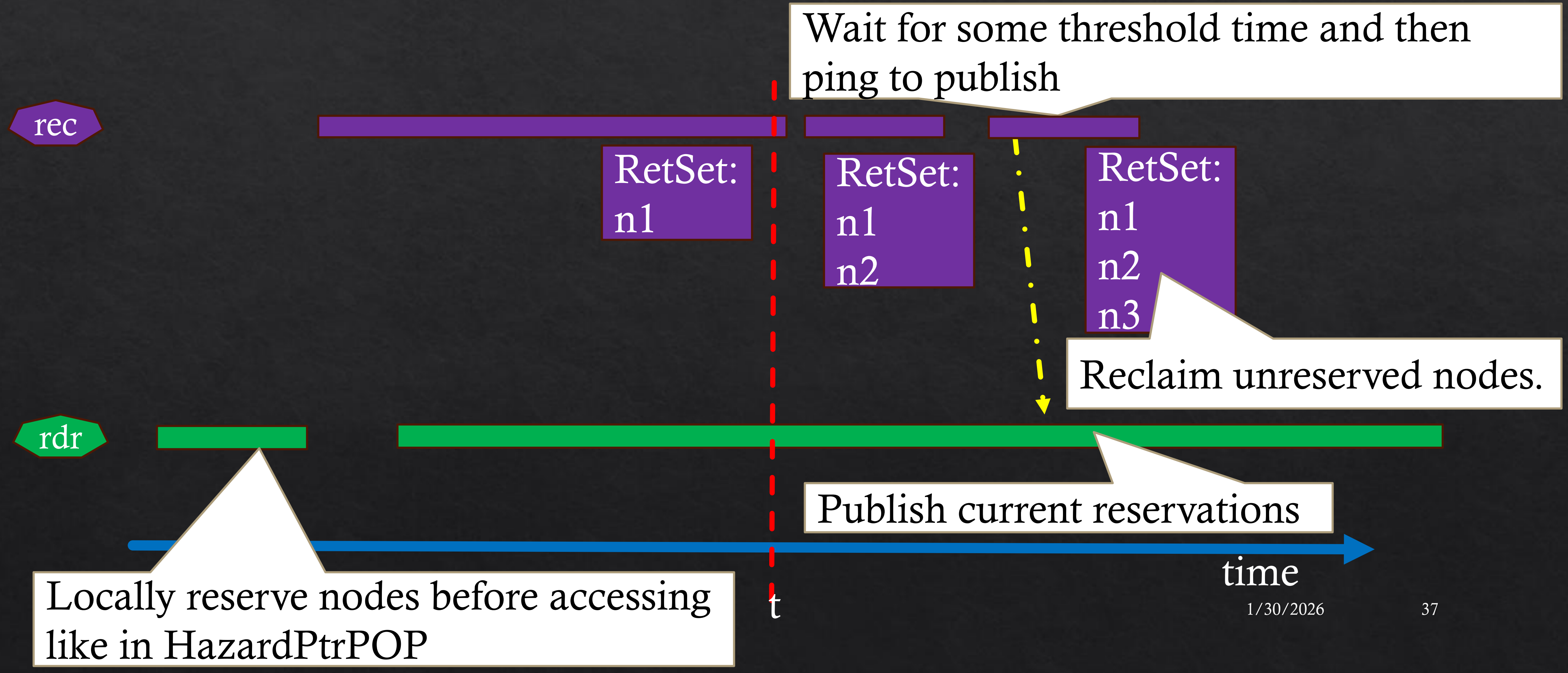


Problem: EBR

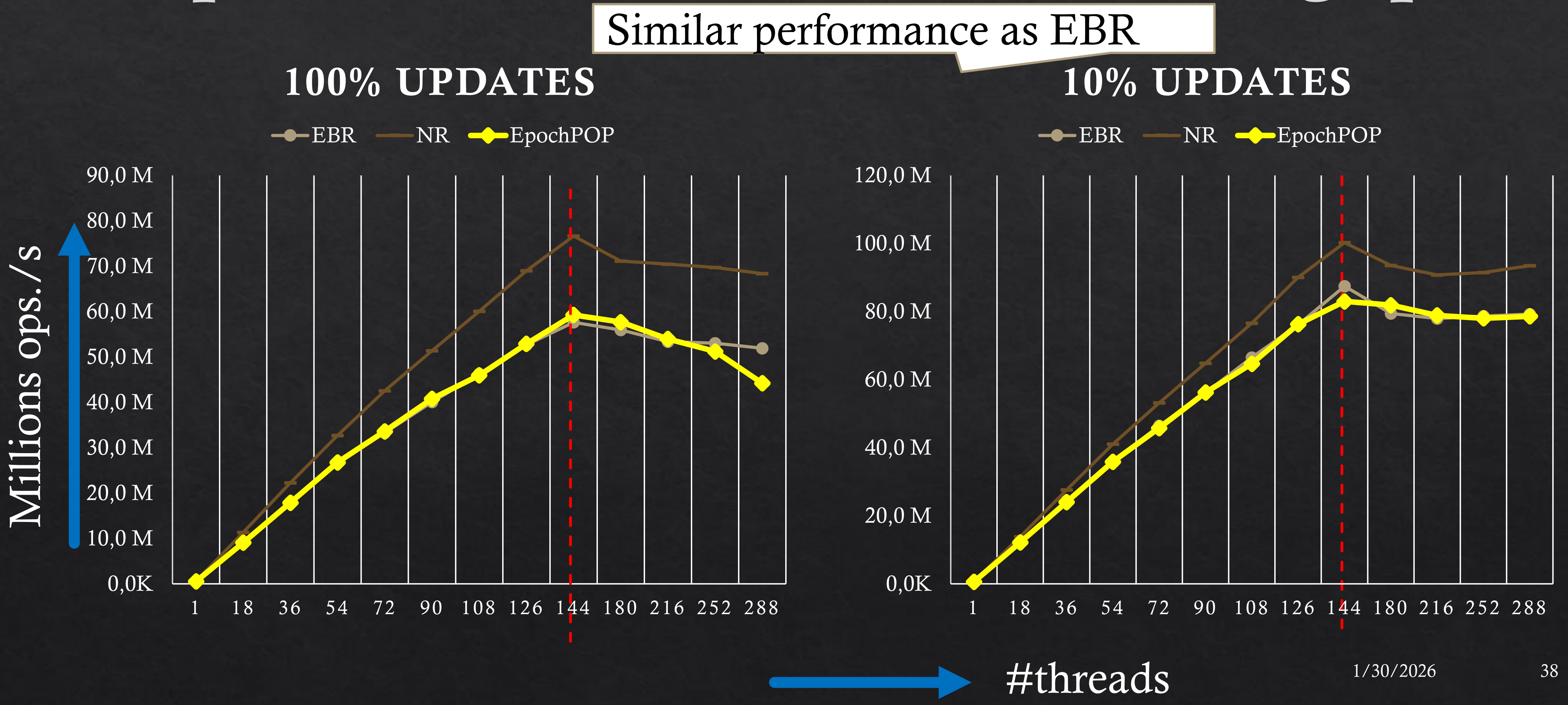


Wait for rdr to go quiescent.
rec
RetSet:
n1
RetSet:
n1
n2
RetSet:
n1
n2
…
rdr may never go quiescent.
Unbounded garbage.
rdr
time
t

EBR + HazardPtrPOP (EpochPOP)



EpochPOP: Search Tree Throughput



POP reduces Uneven Overhead in HP

- Publishing reservations on ping reduces uneven overhead on readers in SMRs like Hazard Pointers [PPoPP 2025].
- Fast and backward compatible and also works with Hazard Eras.
- Local Node Reservations: solves the issue of unbounded garbage in epoch-based reclamation (EBR).

Problem 3

Deferred reclamation paradigm has drawbacks

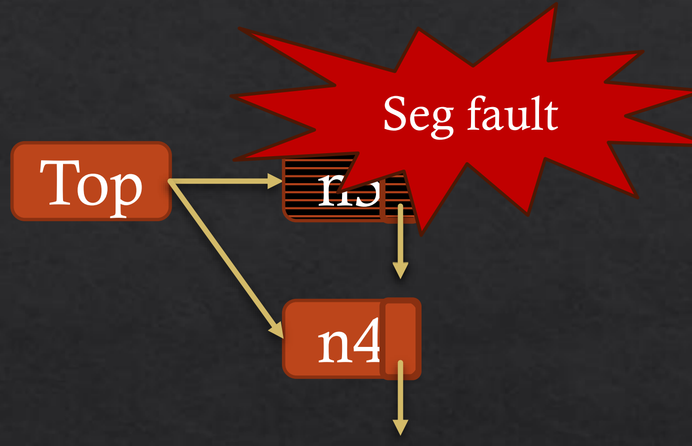
Deferred Reclamation & Batching

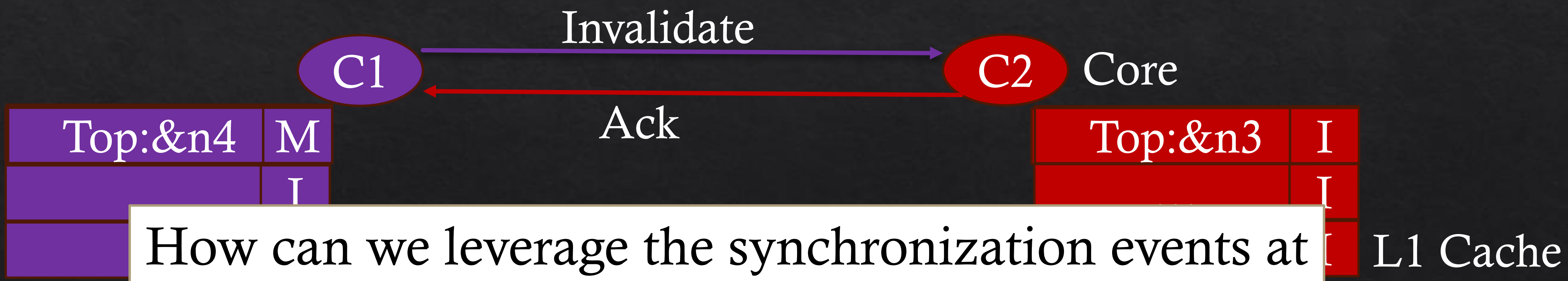
- Existing SMRs defer reclamation for safety and reclaim in batches for performance.
- **Trade-off:** Memory footprint vs. performance.
- Interference with allocator performance [Amort. Freeing, PPOPP '24].
- Hinders memory overcommitment in virtualized data centers.

Problem3: Deferred reclamation has several downsides can we reclaim immediately and yet be fast?

Cache Events Precede Read-Reclaim Races

```
t1  Node* t = Top;                t2
    ...
    Node* next = t->next;         t2
t1  CAS(&Top, t, next)
    ...
t1  delete t;
```





How can we leverage the synchronization events at cache level to resolve read-reclaim races in programs?

Conditional Access (CA)

- A simple hardware extension utilizing **hardware-software co-design** to **capture cache-level** synchronization **events**.

- **Exposes** these events to programs via a novel set of **memory access instructions**, effectively **resolving read-reclaim races**.

(I) Capture Cache Events

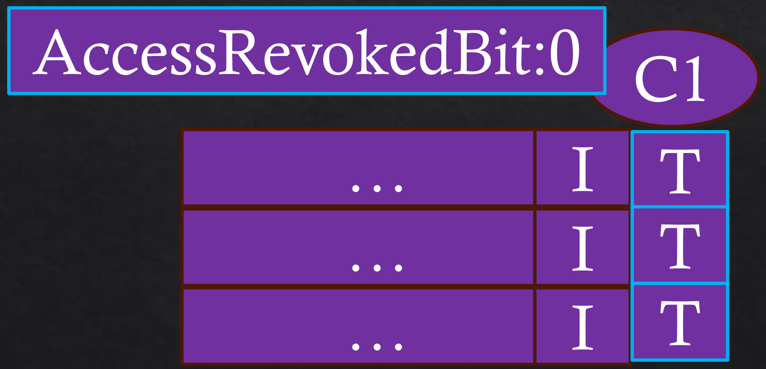
Tag Set: Tagged when accessed first time.

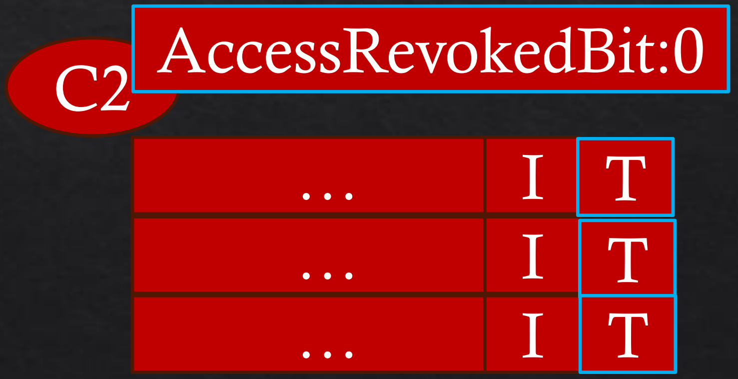
- subsequent <u>invalidations</u> indicate a possible read-reclaim race.
- Enables <u>**monitoring**</u> of possible read-reclaim race to the lines in Tag Set.

TagBit: One bit per cache line.

AccessRevokedBit:

- Initially clear & Set when any of the tagged lines are invalidated or leave the cache.
- Enables <u>**Recording**</u> of possible read-reclaim race events for tagged cache lines.





(II) Expose Cache Events to Programmers

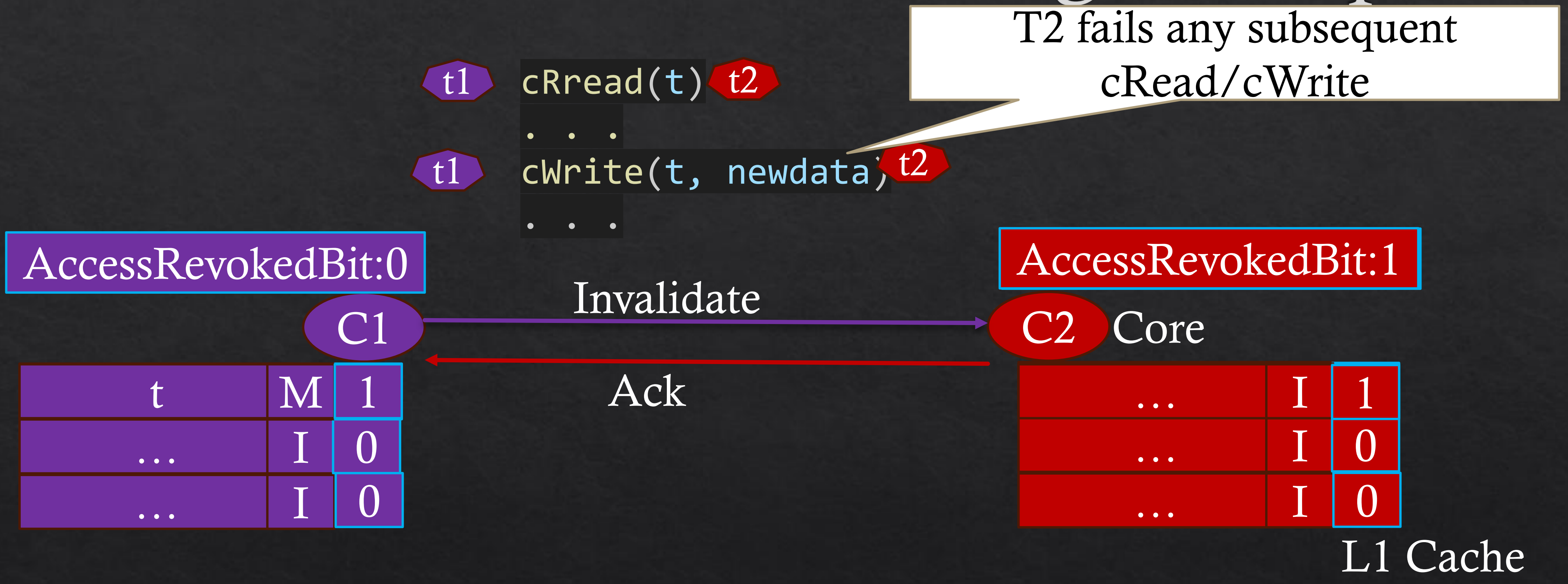
cRead addr, dest
Atomically
- Add addr to TagSet if not in TagSet.
- **If** AccessRevokedBit is set, skip the load, and set a processor flag to indicate error to program.
- **Else** do a normal load.

cWrite addr, v
Atomically
- **If** AccessRevokedBit is set or addr $\notin$ TagSet. Skip store and set a processor flag.
- **Else** do a normal store.

untagOne addr
Remove Address from TagSet

untagAll
- Clear TagSet and AccessRevokedBit

Conditional Access: Working Example



Using Conditional Access

```
int pop () {
    while (true) {
     Node* t = Top;
     if (t == nullptr) return EMPTY;

     Node* next = t->next;
      if (CAS(&Top, t, next)) {
       int res = t->data
       delete t;
       return res;
      }
   }
}
```

```
#define CACHECK if CAFAIL then untagAll(); goto
retry;

int pop () {
    retry:
      Node* t = CREAD(Top); CACHECK;
      if (t == nullptr) untagAll();return EMPTY;

      Node* next = CREAD(t->next); CACHECK;
      CWRITE(&Top, next); CACHECK;
      int res = t->data
      delete t;
      untagAll();
      return res;
}
```

Replace and Evaluate

Conditional Access Prevents Read-Reclaim Race

```
#define CACHECK if CAFAIL then untagAll(); goto
retry;

int pop () {
    retry:
rec    Node* t = CREAD(Top); CACHECK;    rd1  rd2
       if (t == nullptr) untagAll();return EMPTY;

       Node* next = CREAD(t->next); CACHECK;  rd2
rec    CWRITE(&Top, next); CACHECK;   rd1
       int res = t->data
       delete t;
       untagAll();
       return res;
}
```

rec rd1 rd2

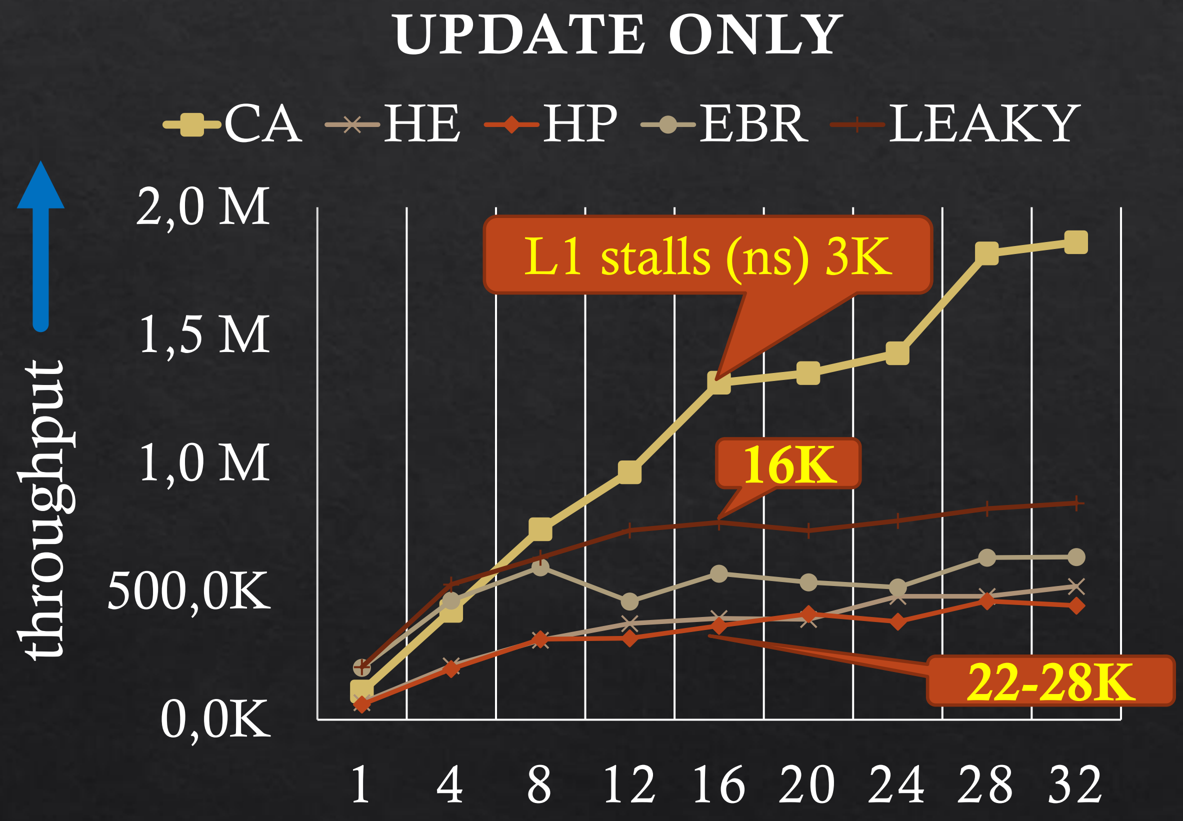
AccessRevokedBits

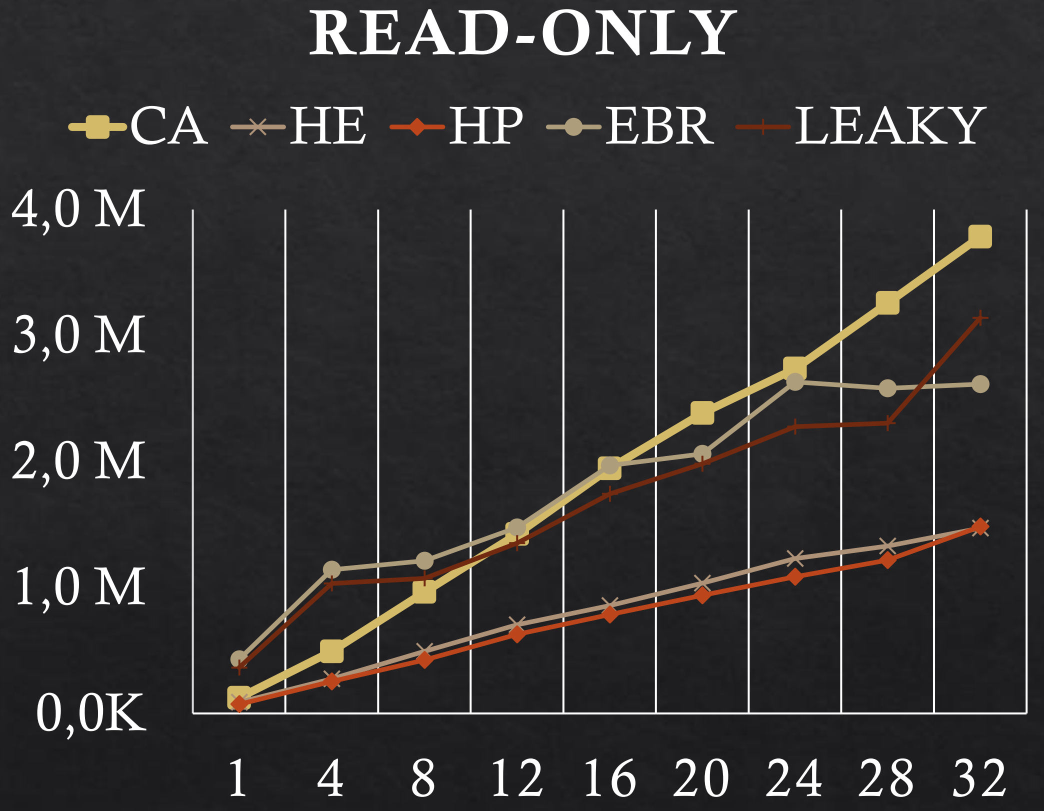
rec	rd1	rd2
0	1	1

TagSet

rec	rd1	rd2
Top	Top	Top

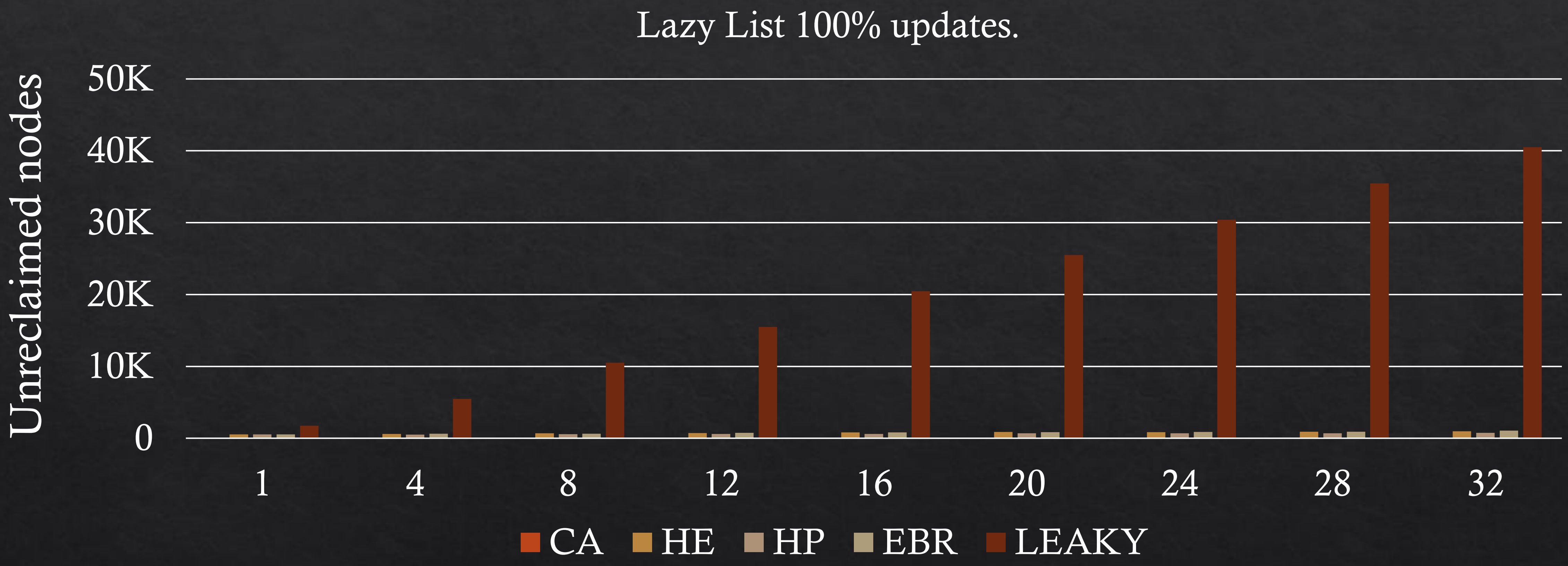
Lazy List Throughput





#threads

Memory Consumption



CA addresses downsides of deferred reclamation

- Conditional Access: a set of new memory access instructions [IPDPS '2023].
- Key Idea: Leverage hardware-software codesign to capture cache events and expose them to programmers to enable safe memory reclamation.
- Sequential data structure like ideal memory footprint along with concurrent data structure like throughput.

Three Problems Three Solutions

- **Problem 1:** Difficult to achieve several desirable properties simultaneously. → Neutralization based reclamation
- **Problem2:** high uneven overhead in Hazard Pointers. → Publish on Ping
- **Problem3:** Deferred reclamation paradigm has drawbacks. → Conditional Access

Co-Designing with others layers of the system stack helps solve three problems in safe memory reclamation.

What other problems such approaches can help with?

Thank You

Travel to HACDA Workshop in DISC 2025 is funded by project HAR.S.H. (project no. ΥΠ3ΤΑ-0560901), within the framework of the National Recovery and Resilience Plan "Greece 2.0" with funding from the European Union – NextGenerationEU.

persist

FORTH
INSTITUTE OF COMPUTER SCIENCE