2020-11-26
KAIST Software Graduate Program

When is it safe to free memory in concurrent programming?

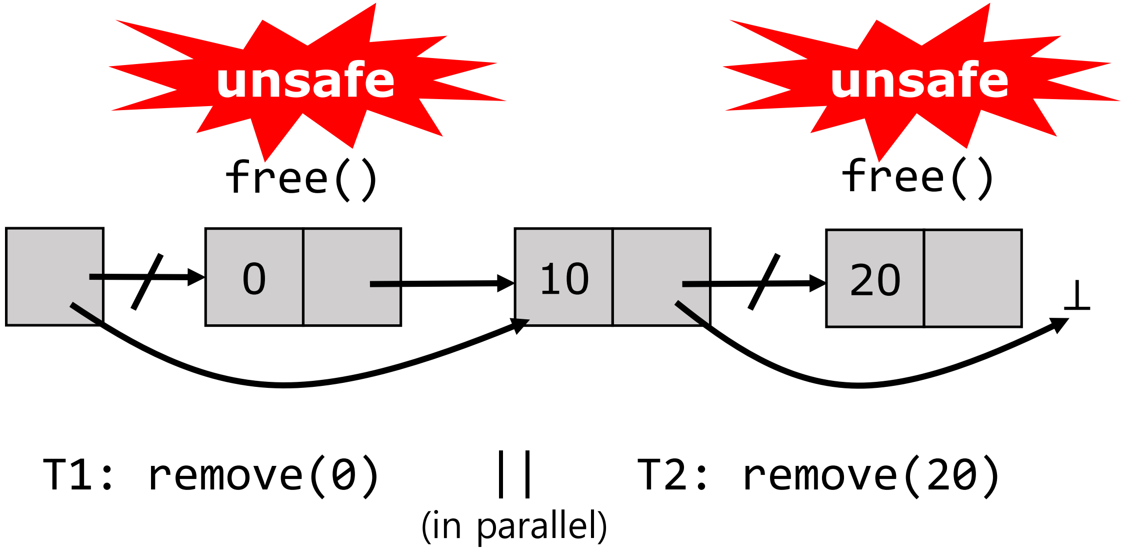
An opinionated survey of memory reclamation algorithms

Jeehoon Kang

(A joint work with Jaehwang Jung)

KAIST Korea Advanced Institute of Science and Technology

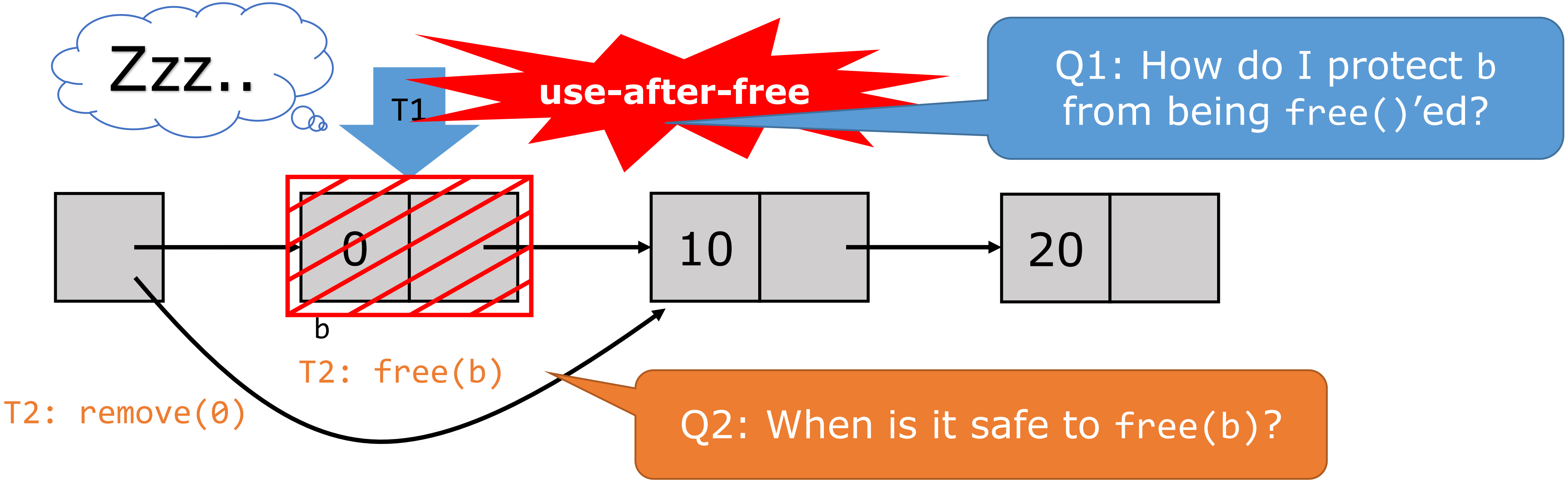
Concurrent Data Structures (CDS)



+ fast & non-blocking

- complex design
  - …
  - **memory management**

Memory Reclamation in CDS: You Are Not Free to **free()** the Removed Block

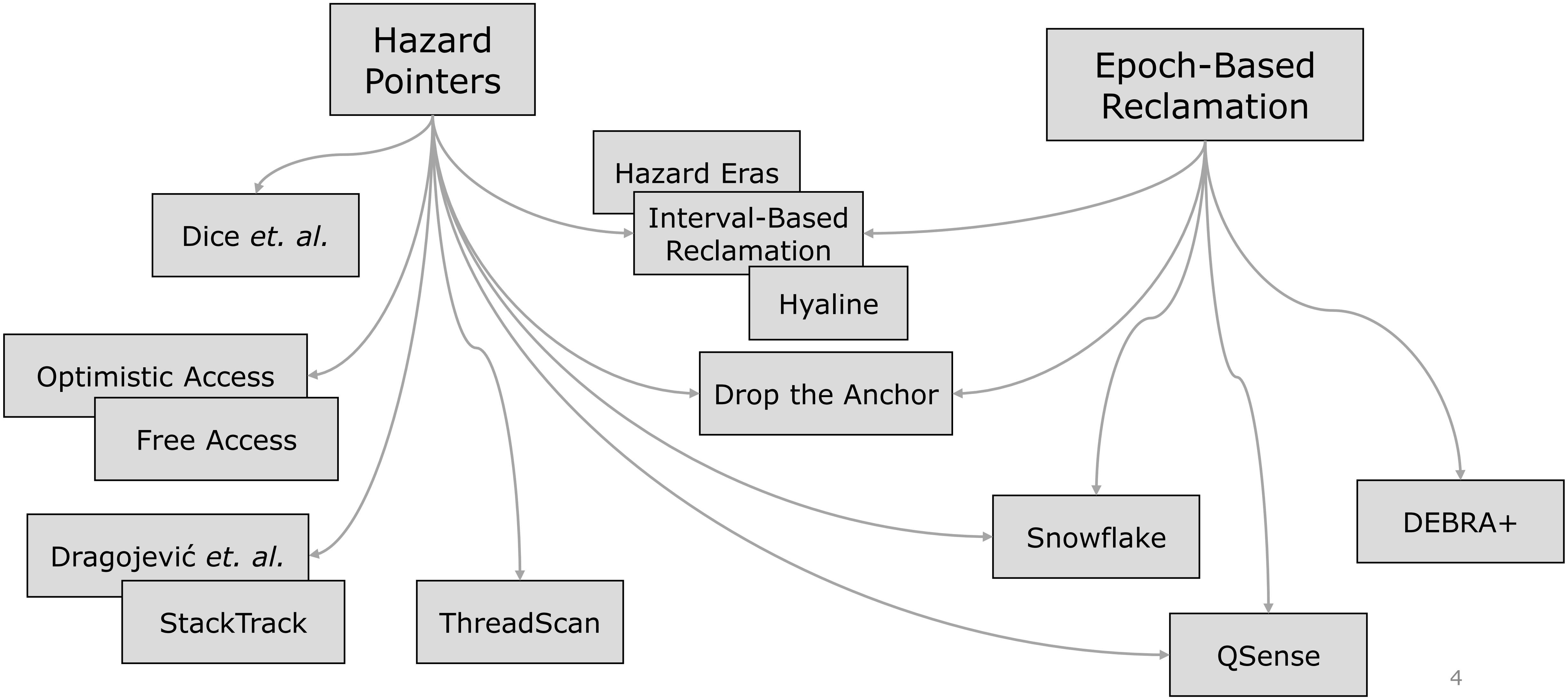


Q1: How do I protect b from being free()'ed?

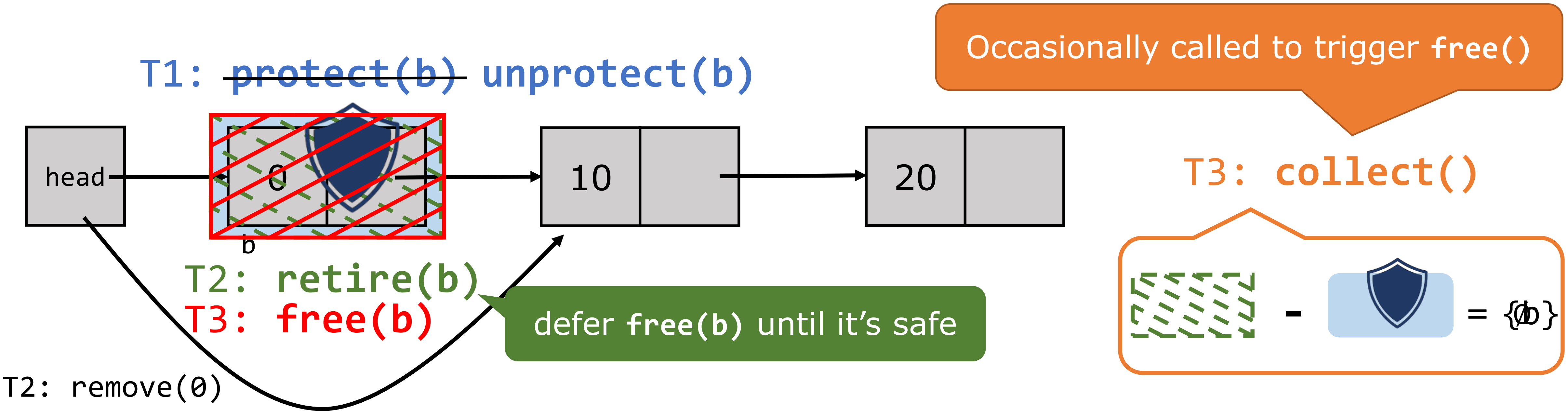
Q2: When is it safe to free(b)?

Solution: Safe Memory Reclamation **(SMR)** Algorithms

The SMR Algorithms Literature



SMR Algorithm #1
Hazard Pointers (HP)



T1: ~~protect(b)~~ unprotect(b)

T2: retire(b)

T3: free(b)

T2: remove(0)

defer **free(b)** until it's safe

Occasionally called to trigger **free()**

T3: collect()

Q1: How do I protect b from being free()'ed?

A1: **protect(b)** (called *hazards*)

Q2: When is it safe to free(b)?

A2: When it is no longer **protect()**'ed.

HP's Example
HP-Protected Treiber's Stack

```
struct Node<T> { Node<T>* next; T data; };
struct Stack<T> {
  Atomic<Node<T>*> head;        // nullptr, initially
  void push(T data) {
    auto node = new Node<T>{nullptr, data};
    do {
      auto cur = this->head.load();
      node->next = cur;
    } while (!this->head.cas(cur, node));
  }
  optional<T> Stack<T>::pop() {
    auto cur = nullptr;
    loop {
      cur = this->head.load();
      if (cur == nullptr) return {};
      auto next = cur->next;
      if (this->head.cas(cur, next)) {
        free(cur); break;       // unsafe reclamation
      }
    }
    return std::move(cur->data);
  }
}
```

unsafe

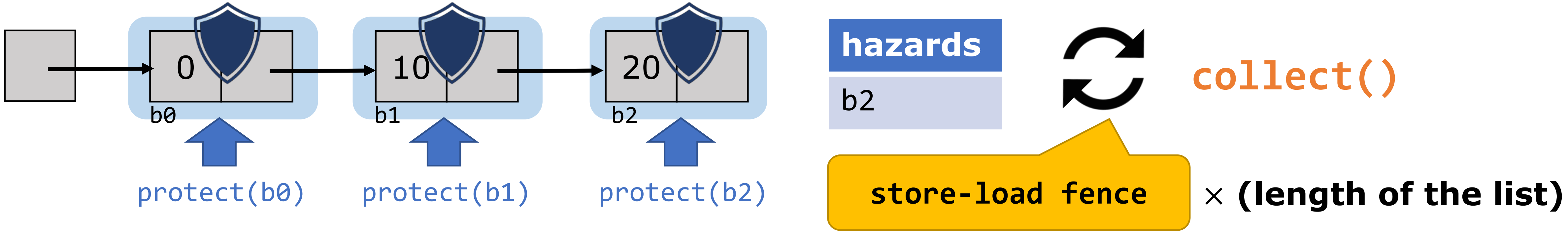
```
protect(cur);
if (this->head.load() != cur)
    continue;
```

```
unprotect(cur);
```

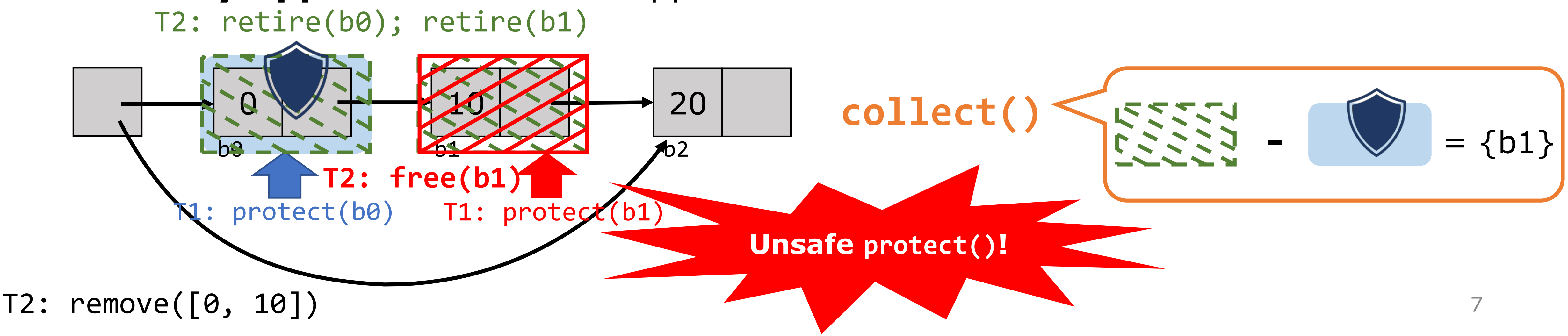
```
retire(cur);
```

HP's Drawbacks
Not Fast & Not Widely Applicable

Not fast: requires per-`protect()` synchronization with expensive store-load fence



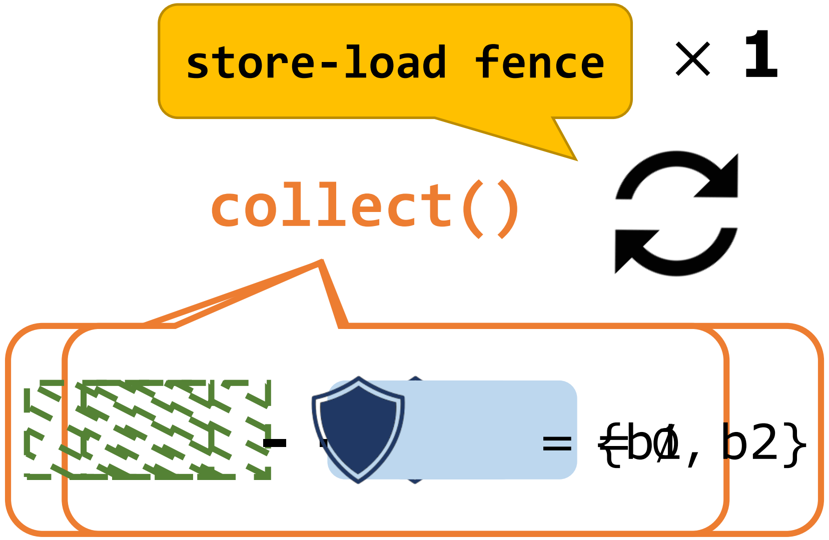
Not widely Applicable: doesn't support “chained retirement”

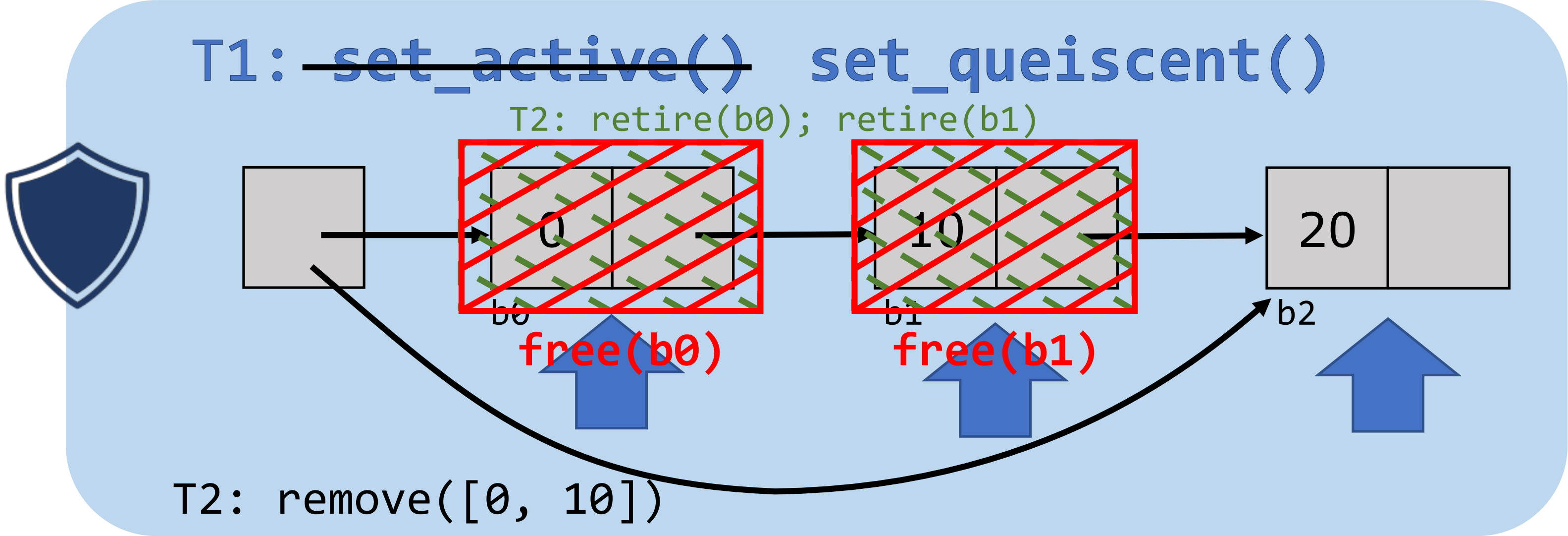


SMR Algorithm #2
Epoch-Based Reclamation (EBR)

Fast: synchronization
for each active state

Widely applicable: Protection for all potential accesses




T1: ~~set_active()~~ set_queiscent()
T2: retire(b0); retire(b1)
0
10
20
b0
b1
b2
free(b0)
free(b1)
T2: remove([0, 10])

Q1: How do I protect b from being free()'ed?

A1: set_active() for

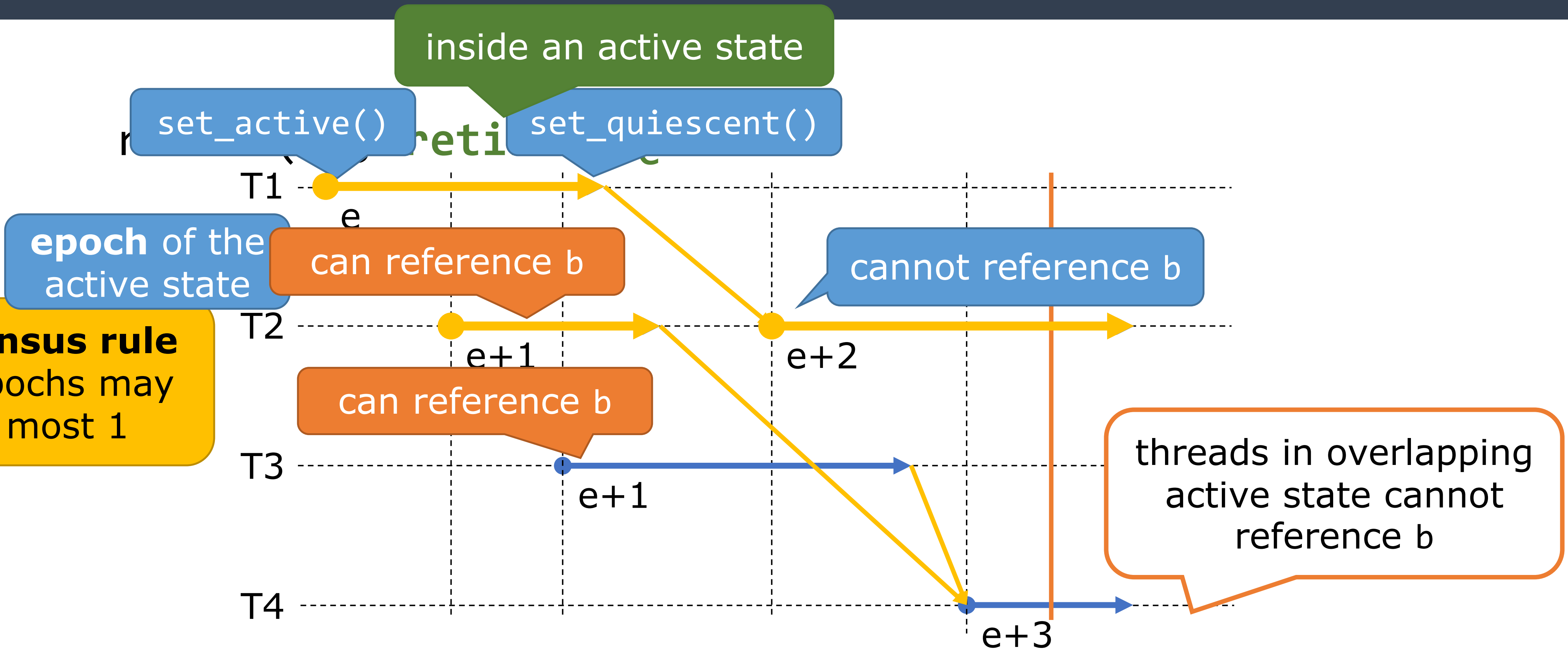
How to decide?

Q2: When is it safe to free(b)?

A2: When b can't be accessed by any active threads.

Epoch Consensus

Epoch consensus rule
Concurrent epochs may differ by at most 1



inside an active state

set_active()

set_quiescent()

epoch of the active state

can reference b

cannot reference b

can reference b

threads in overlapping active state cannot reference b

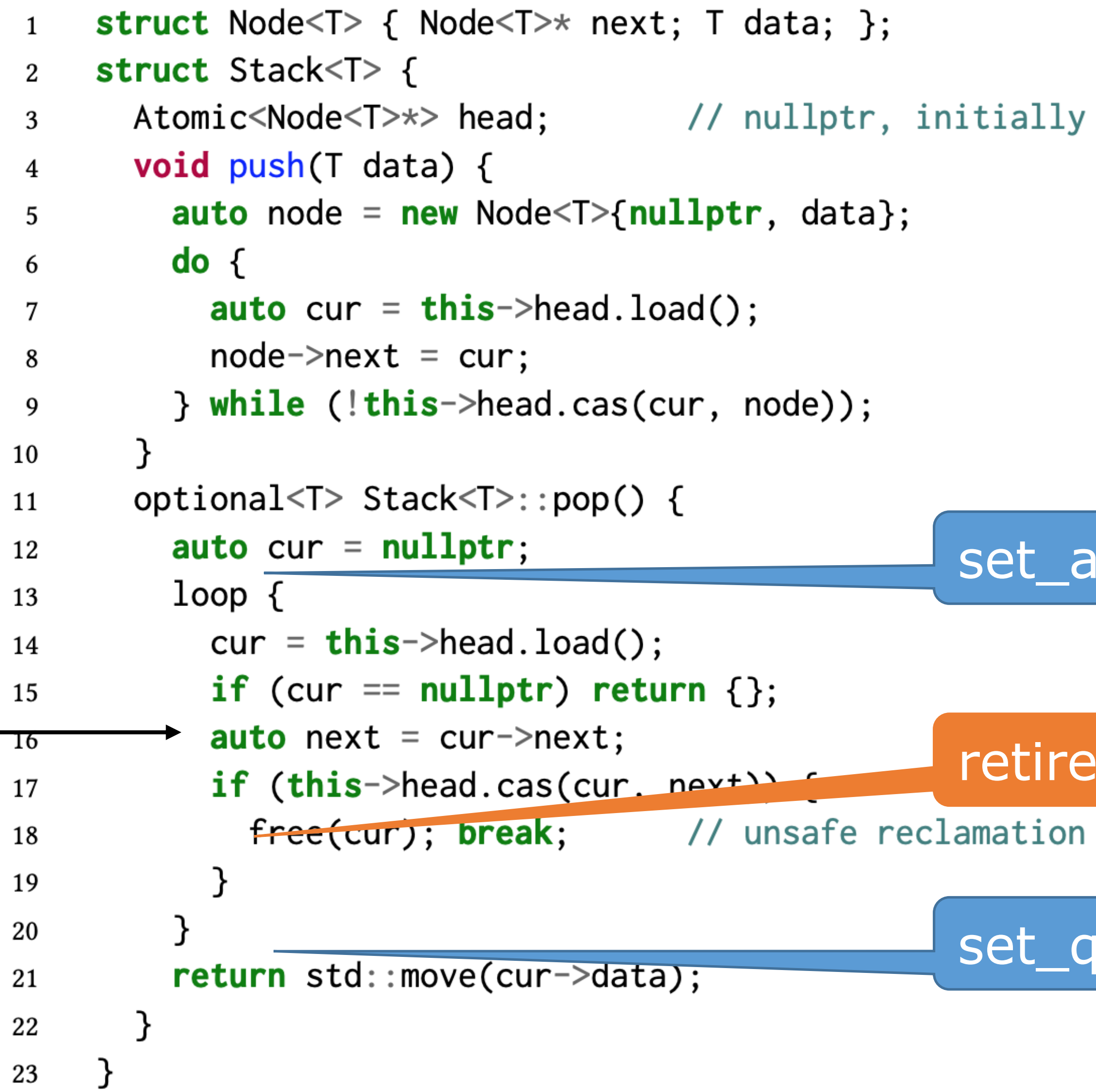
T4: collect()@e+3; free(b)

Q2: When is it safe to `free(b)`?

A2: At epoch `e+3`

EBR's Example

EBR-Protected Treiber's Stack



```
struct Node<T> { Node<T>* next; T data; };
struct Stack<T> {
  Atomic<Node<T>*> head;       // nullptr, initially
  void push(T data) {
    auto node = new Node<T>{nullptr, data};
    do {
      auto cur = this->head.load();
      node->next = cur;
    } while (!this->head.cas(cur, node));
  }
  optional<T> Stack<T>::pop() {
    auto cur = nullptr;
    loop {
      cur = this->head.load();
      if (cur == nullptr) return {};
      auto next = cur->next;
      if (this->head.cas(cur, next)) {
        free(cur); break;      // unsafe reclamation
      }
    }
    return std::move(cur->data);
  }
}
```

unsafe

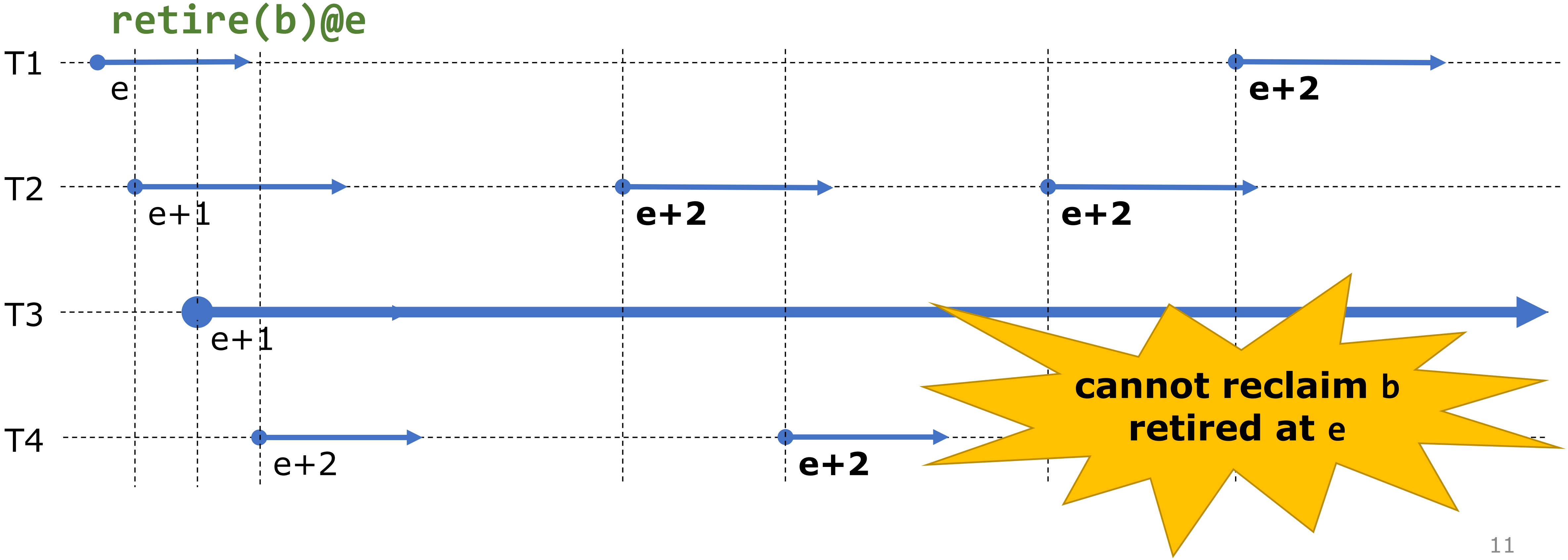
set_active();

retire(cur);

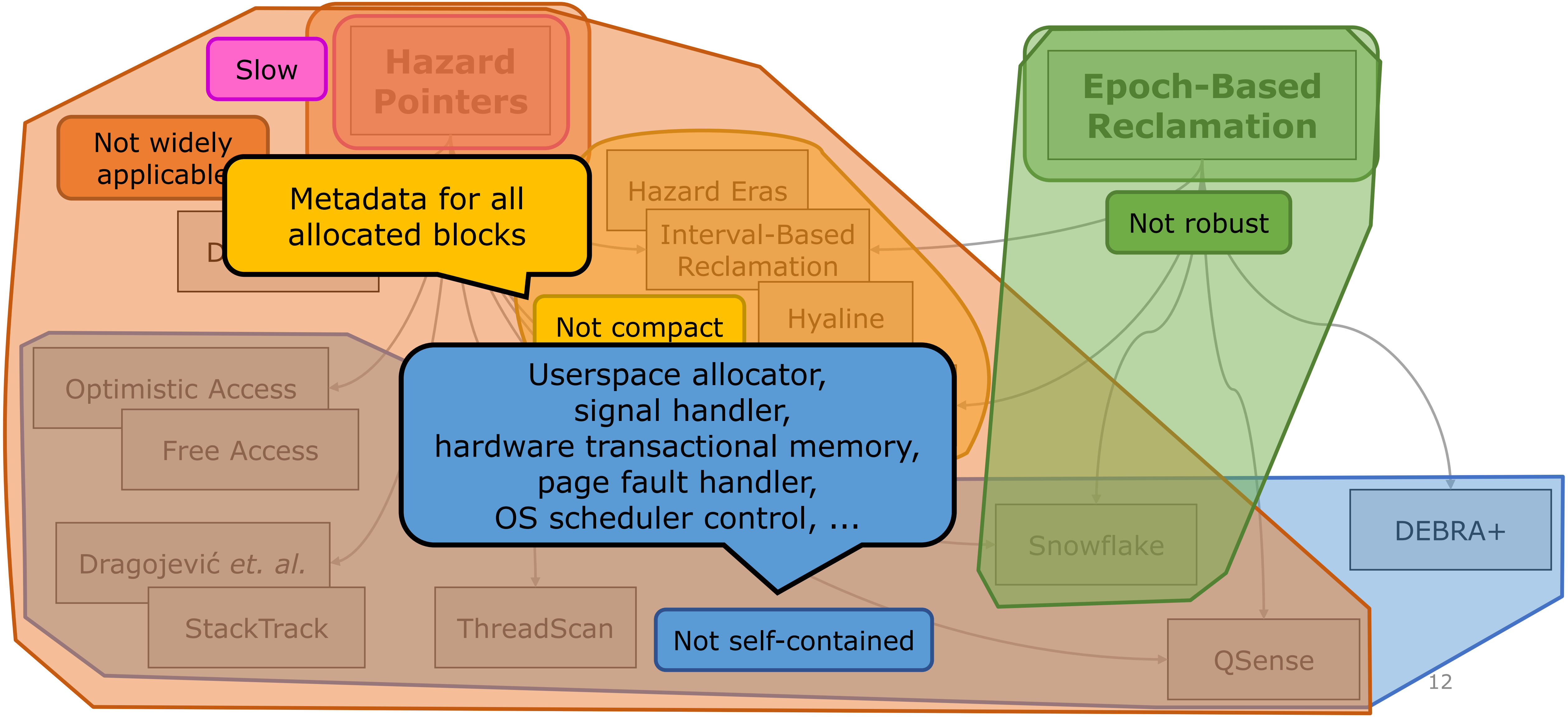
set_quiescent();

EBR's Drawback: Not Robust

If a thread doesn't exit its active state, reclamation is indefinitely blocked.



SMR Algorithms Literature



Slow

Hazard Pointers

Not widely applicable

Metadata for all allocated blocks

D

Epoch-Based Reclamation

Hazard Eras

Interval-Based Reclamation

Not robust

Hyaline

Not compact

Optimistic Access

Free Access

Userspace allocator,
signal handler,
hardware transactional memory,
page fault handler,
OS scheduler control, …

Dragojević *et. al.*

Snowflake

DEBRA+

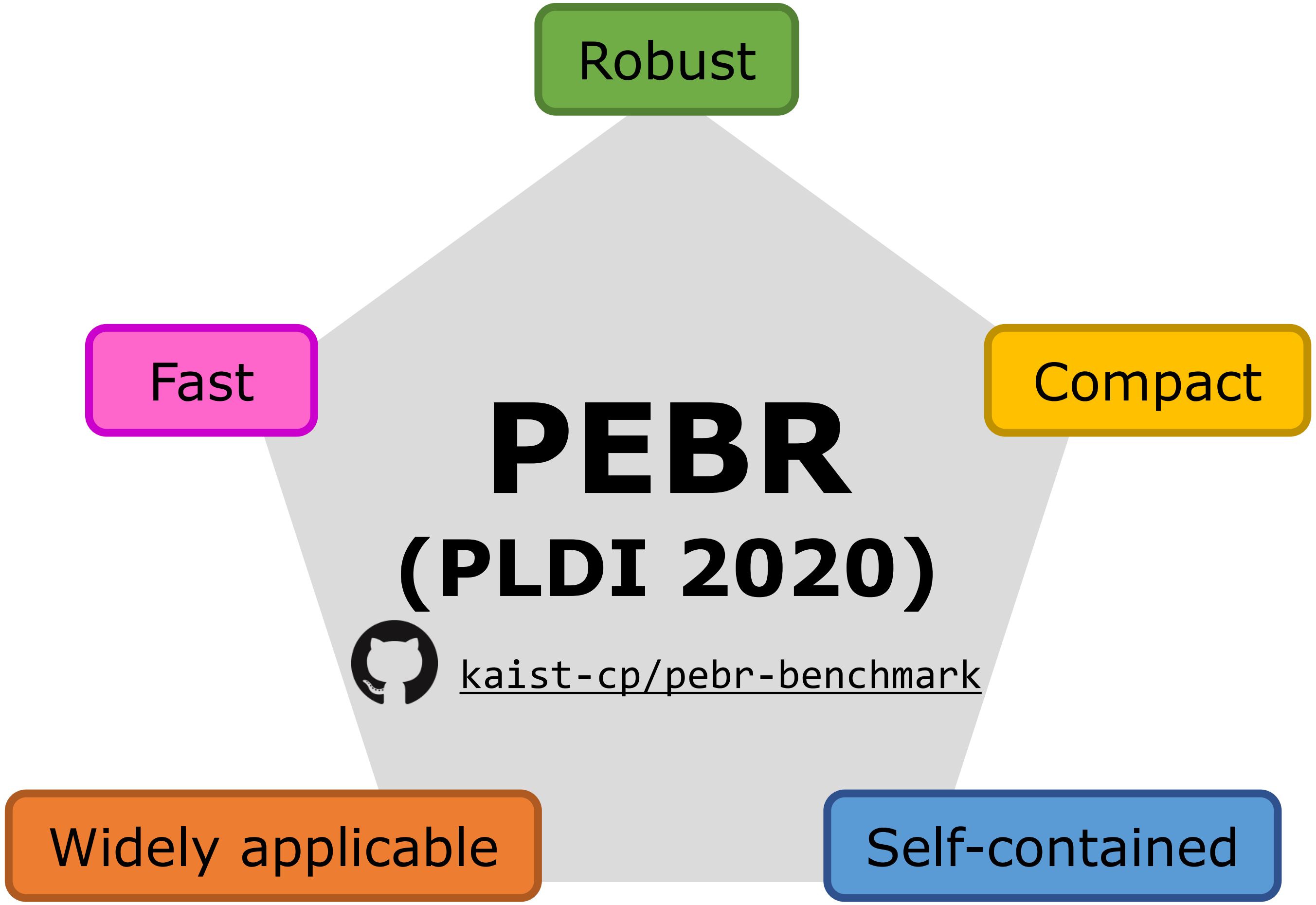
StackTrack

ThreadScan

Not self-contained

QSense

Pointer-and-Epoch-Based Reclamation



PEBR

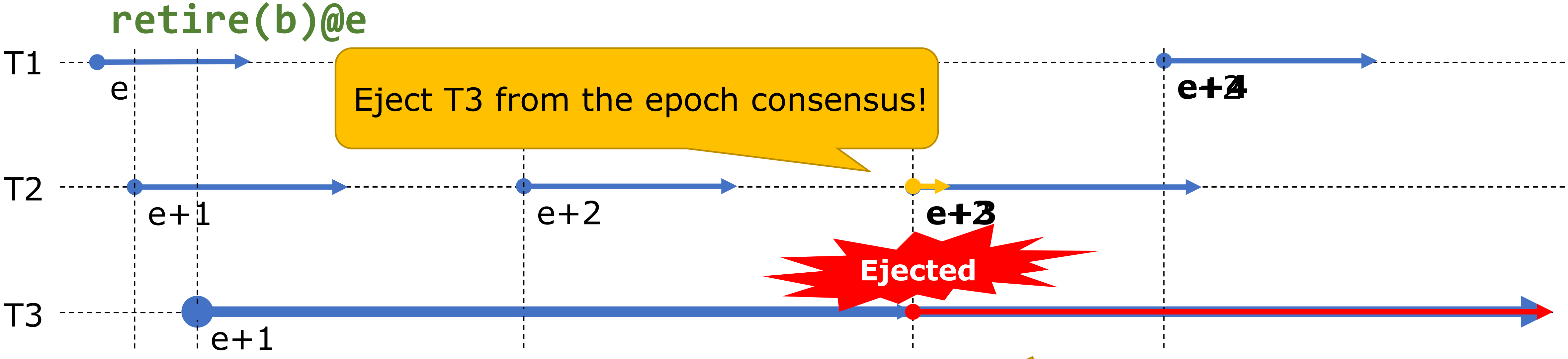
(PLDI 2020)

kaist-cp/pebr-benchmark

▶

1. Hybrid of EBR and HP using **ejection**.
2. **Widely applicable** API even in the presence of ejection
3. **Robust**, **self-contained** and **compact** ejection algorithm

Making EBR Robust with Ejection

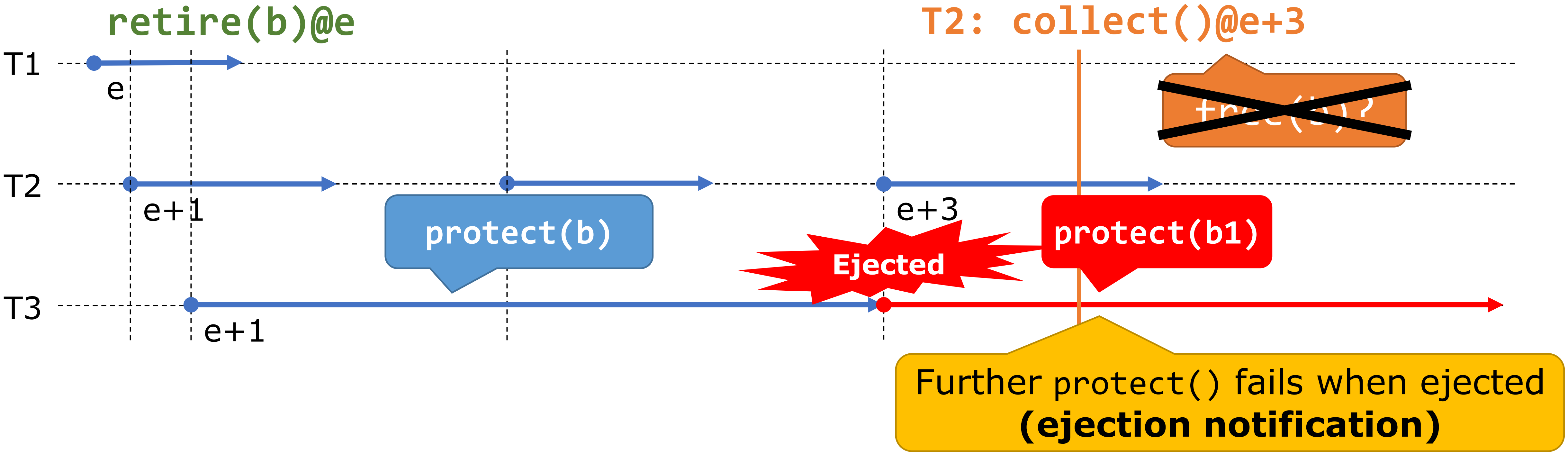


Eject T3 from the epoch consensus!

Notices ejection and
starts recovery procedure.

cannot reclaim b
retired at e

PEBR in a Nutshell



Further protect() fails when ejected **(ejection notification)**

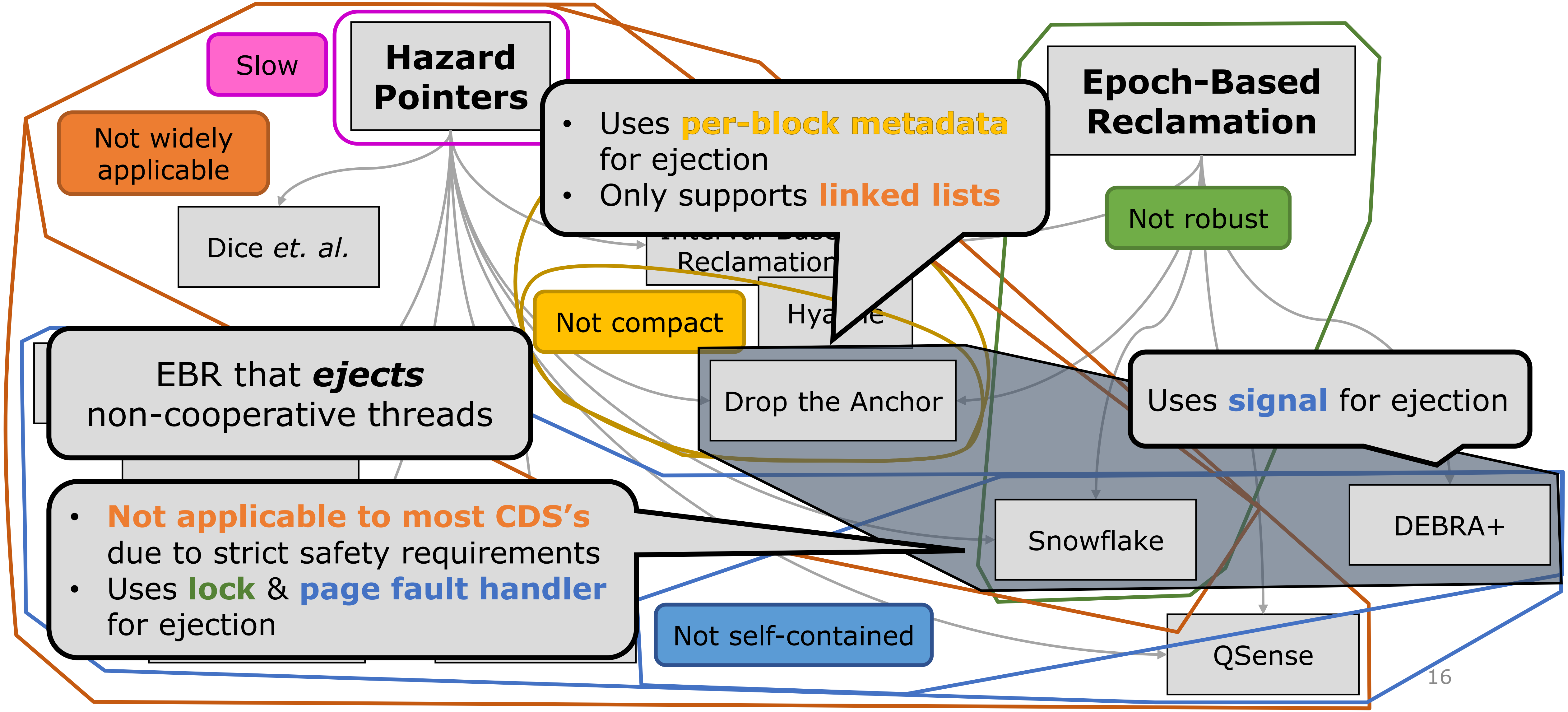
Q1: How do I protect b from being free()'ed?

A1: protect(b) inside an active state. If failed, then you're ejected.

Q2: When is it safe to free(b)?

A2: At epoch **e+3**, if isn't not protect()'ed.

SMR Algorithms Literature: Ejection



Slow

Hazard Pointers

Not widely applicable

Dice *et. al.*

- Uses **per-block metadata** for ejection
- Only supports **linked lists**

Epoch-Based Reclamation

Not robust

Reclamation

Hya le

Not compact

Drop the Anchor

EBR that ***ejects*** non-cooperative threads

Uses **signal** for ejection

- **Not applicable to most CDS's** due to strict safety requirements
- Uses **lock** & **page fault handler** for ejection

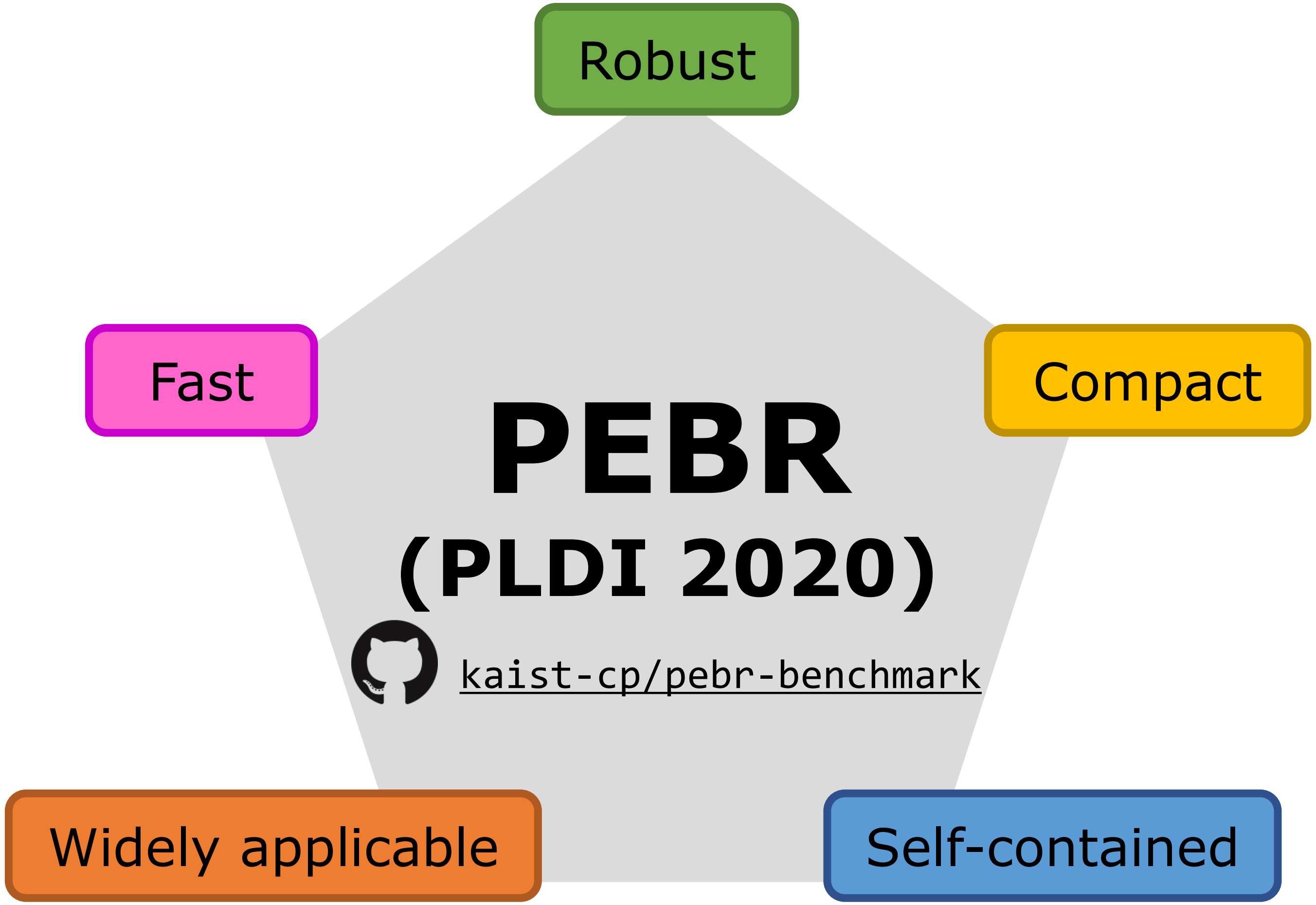
Snowflake

DEBRA+

Not self-contained

QSense

Pointer-and-Epoch-Based Reclamation



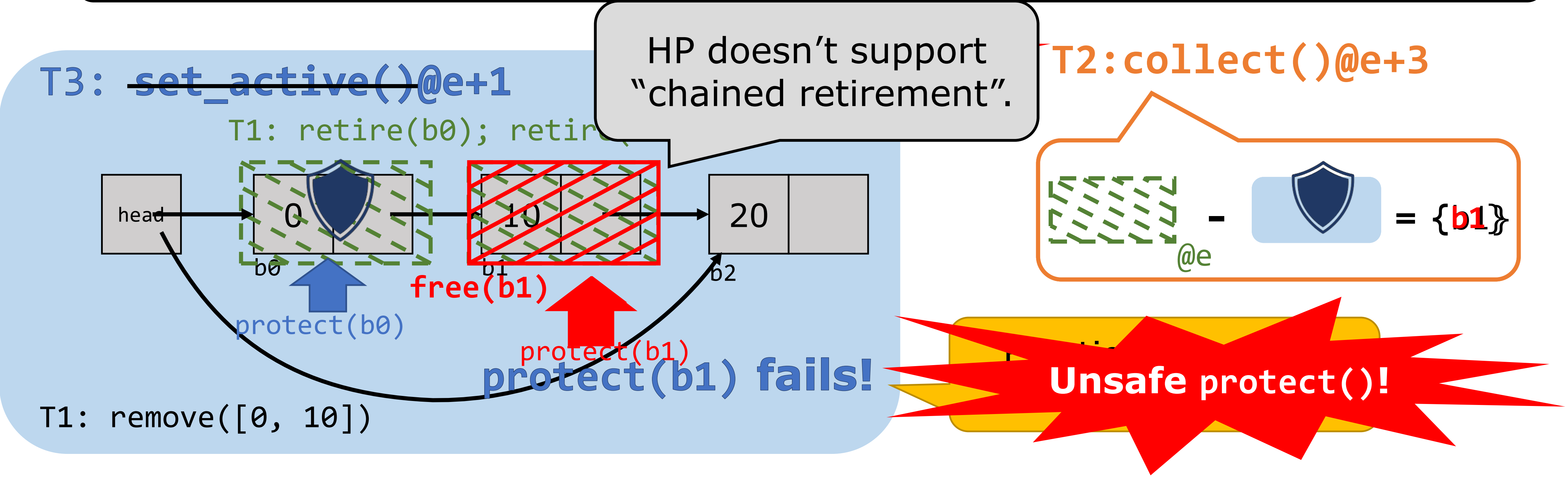
1. Hybrid of EBR and HP using **ejection**.

2. **Widely applicable** API even in the presence of ejection

3. **Robust**, **self-contained** and **compact** ejection algorithm

Wide Applicability by Ejection Notification

PEBR prevents **unsafe** **`protect()`** with **ejection notification**.



HP doesn’t support “chained retirement”.

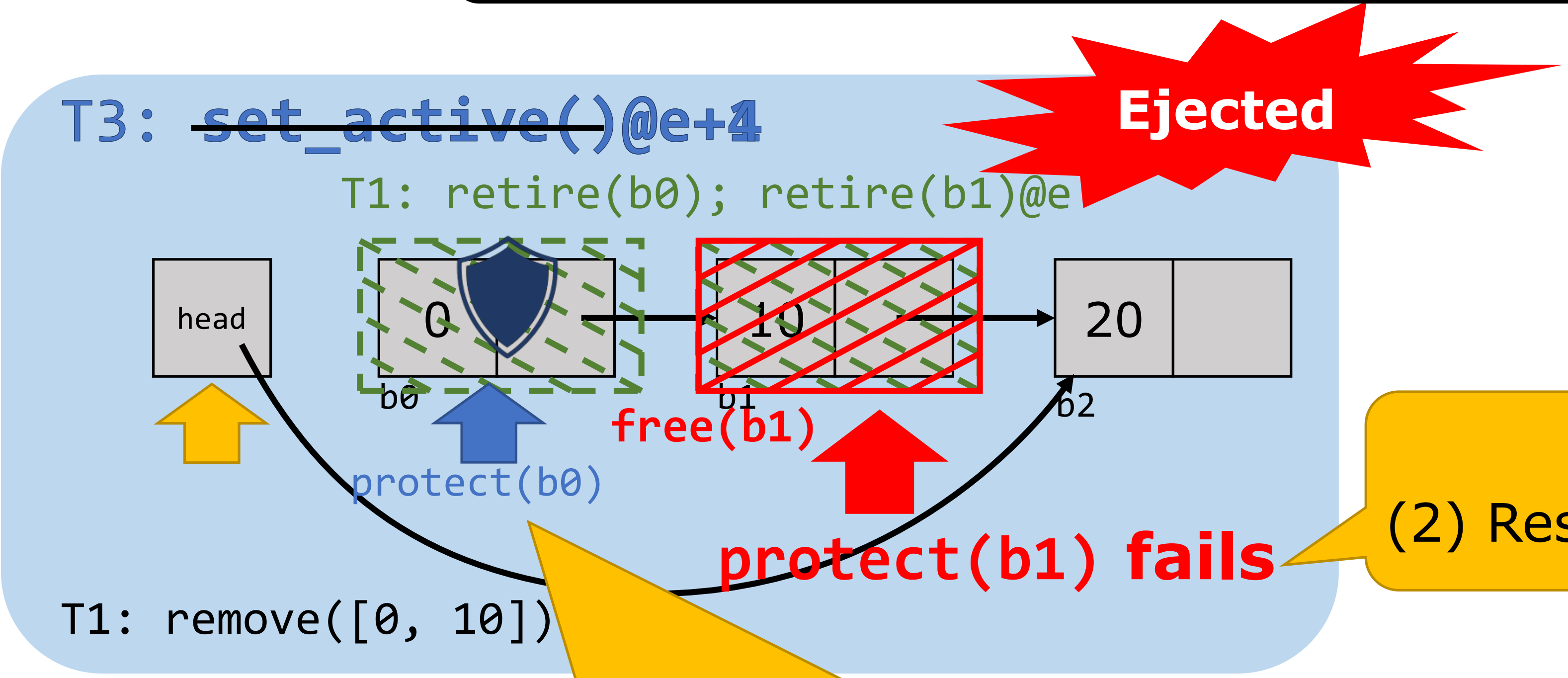
protect(b1) fails!

Unsafe protect()!

Safety Requirement:
When ejected, don’t start traversal from the previously protected blocks.

Wide Applicability in Practice

Example: Porting EBR-based data structure to PEBR



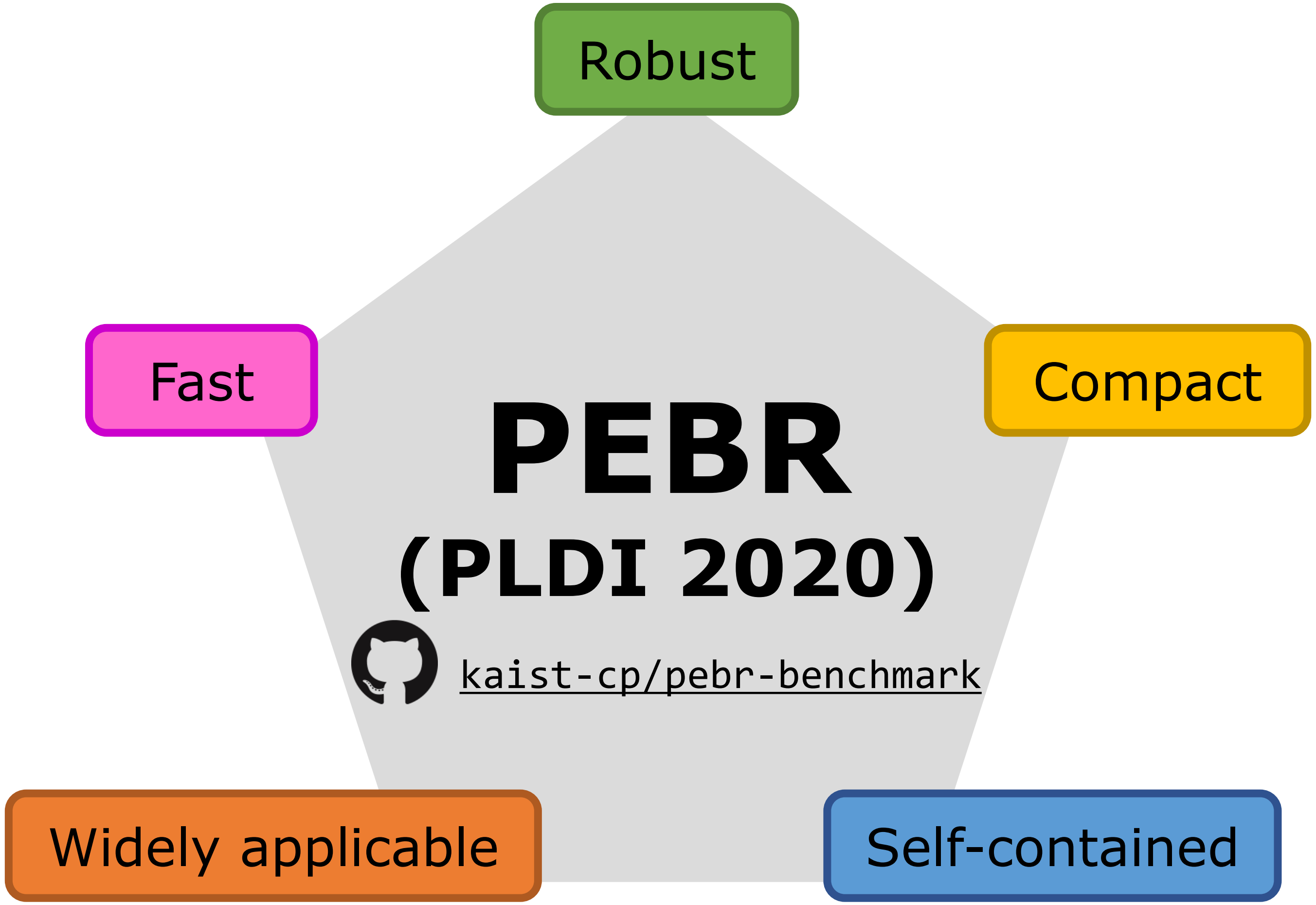
Ejection detected!
(2) Restart the operation from **head**!

(1) protect() before each dereference

Safety Req

When ejected, don’t start traversal from the previously protected blocks.

Pointer-and-Epoch-Based Reclamation



Robust

Fast

Compact

Widely applicable

Self-contained

PEBR

(PLDI 2020)

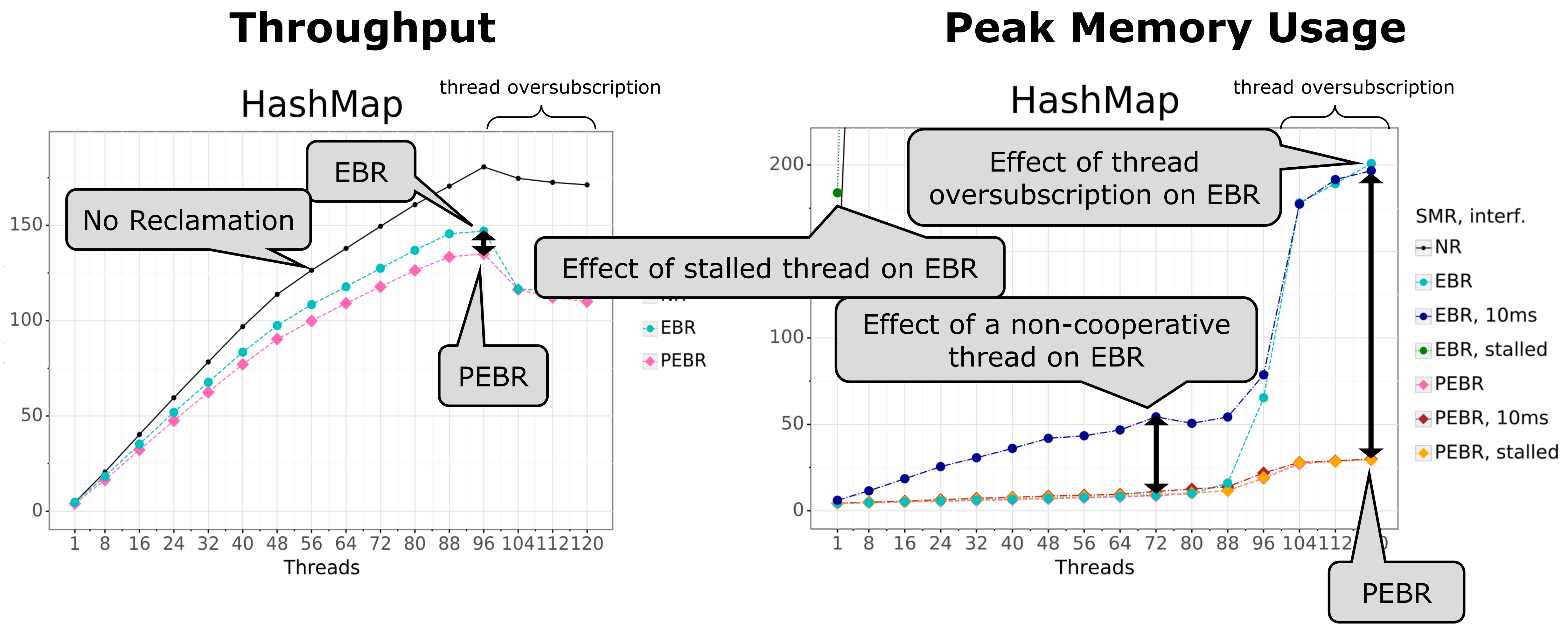
kaist-cp/pebr-benchmark

1. Hybrid of EBR and HP using **ejection**.

2. **Widely applicable** API even in the presence of ejection

3. **Robust**, **self-contained** and **compact** ejection algorithm

PEBR is Fast and Robust



Throughput

Peak Memory Usage

No Reclamation

EBR

PEBR

Effect of stalled thread on EBR

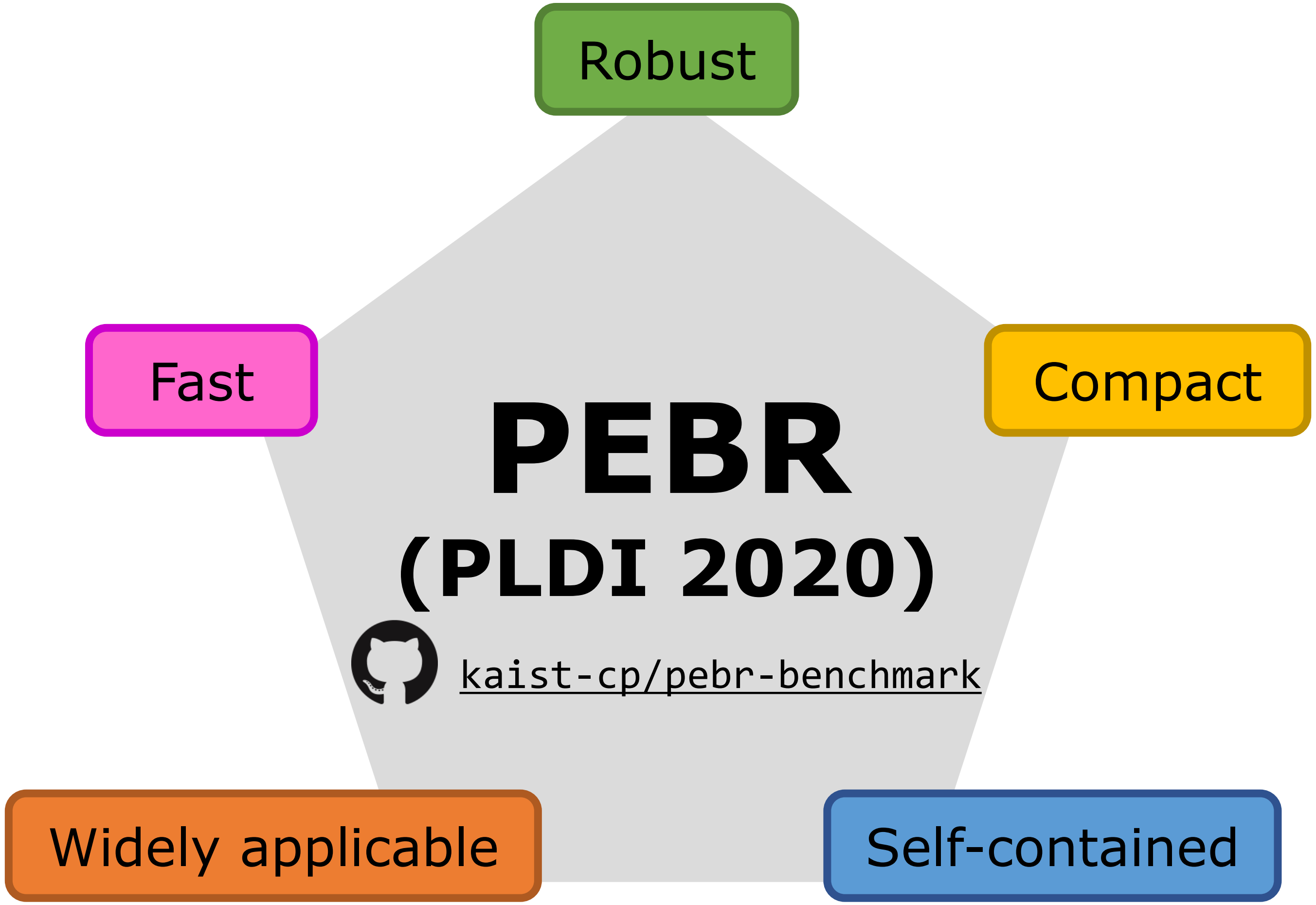
Effect of thread oversubscription on EBR

Effect of a non-cooperative thread on EBR

PEBR

(On machine with 96 hardware threads)

What Else Is in the Paper?



Robust

Fast

Compact

PEBR

(PLDI 2020)

kaist-cp/pebr-benchmark

Widely applicable

Self-contained

- Full algorithm
- Safety proof
- Full benchmark results
- Rust API that statically enforces some safety requirements