WAIT-FREE MEMORY RECLAMATION AND DATA STRUCTURES

RUSLAN NIKOLAEV

RESEARCH ASSISTANT PROFESSOR

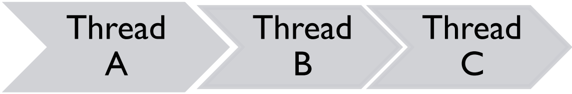
VIRGINIA TECH, SSRG

ABOUT ME

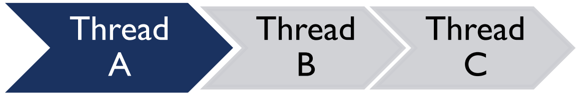
- Worked in industry (Microsoft, Pure Storage)
- Joined Virginia Tech, Electrical and Computer Engineering Department in 2017 as a Research Assistant Professor
  - Working on different projects in systems and concurrency
- Have research publications at SOSP, VEE, PODC, DISC, and PPoPP
  - Today's talk partially overlaps with my recent PPoPP '20 publication "Universal Wait-Free Memory Reclamation", which is co-authored with Prof. Binoy Ravindran from Virginia Tech

CONCURRENT DATA STRUCTURES

- Many-core systems today require efficient access to data
  - Concurrent data structures
- Multiple threads need to ***safely*** manipulate data structures (similar to sequential data structures)
  - "nothing bad will happen"



- Concurrency also adds a ***liveness*** property, which stipulates how threads will be able to make progress
  - "something good will happen eventually"



NON-BLOCKING PROGRESS GUARANTEES

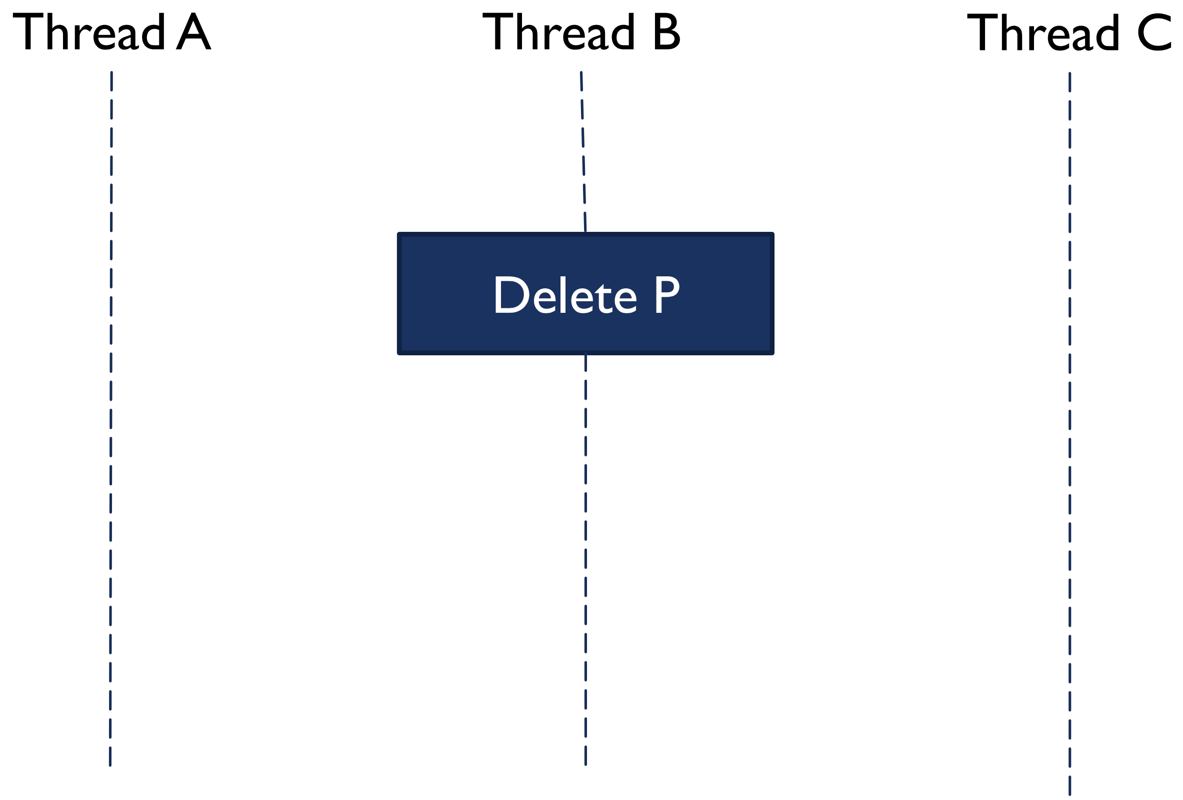
- ***Obstruction-free***: a thread performs an operation in a finite number of steps if executed in isolation from other threads
- ***Lock-free***: at least **one** thread always makes progress in a finite number of steps
- ***Wait-free:*** **all** threads make progress in a finite number of steps

NON-BLOCKING PROGRESS GUARANTEES

- ***Obstruction-free***: a thread performs an operation in a finite number of steps if executed in isolation from other threads
- ***Lock-free***: at least **one** thread always makes progress in a finite number of steps
- ***Wait-free:*** **all** threads make progress in a finite number of steps

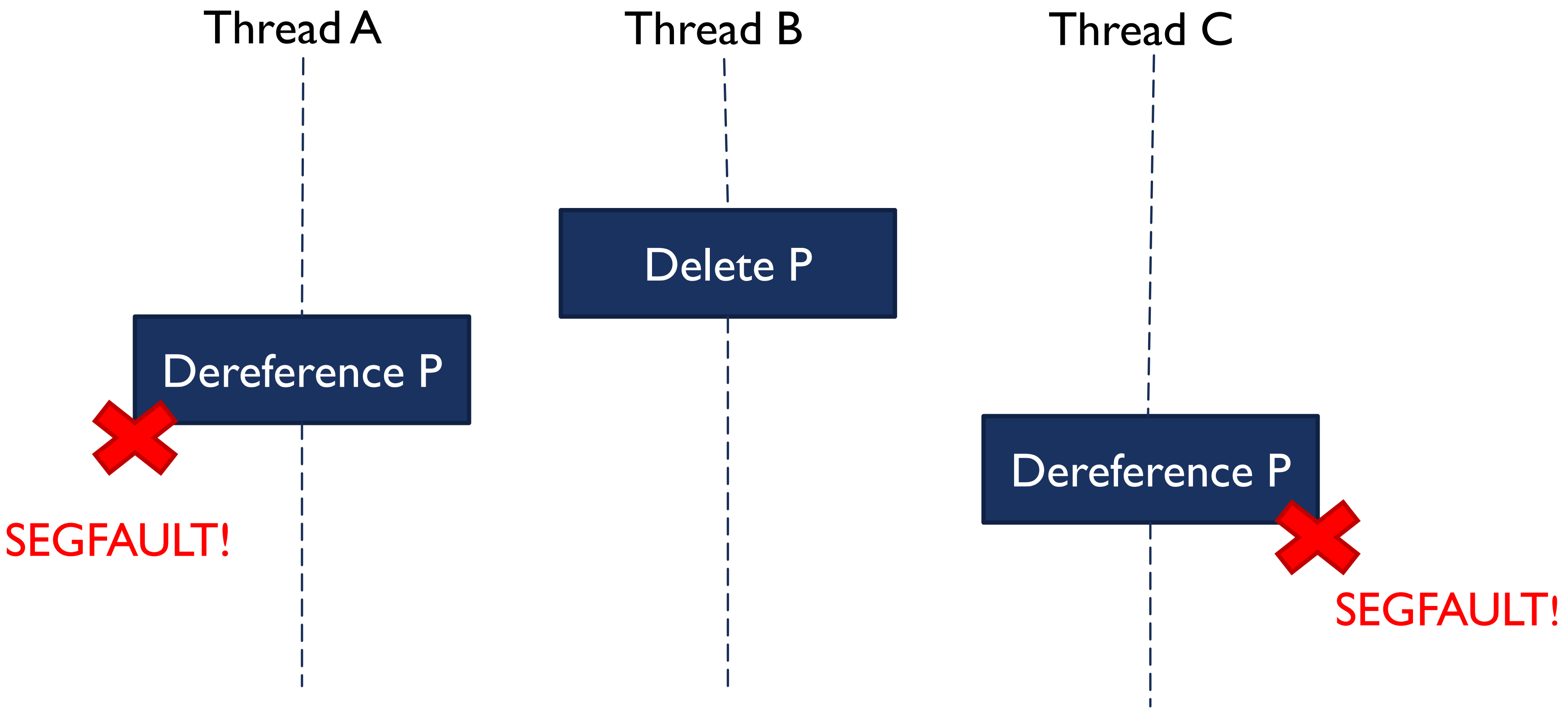
- ***Wait-freedom*** is the strongest form of non-blocking progress
- Wait-free algorithms are gaining more practical relevance and efficiency (Kogan and Petrank's *fast-path-slow-path* methodology, see [PPoPP '12])
- **CAS** (compare-and-swap) is used universally in lock-free and wait-free algorithms
- **F&A** (fetch-and-add) is often available as a specialized instruction

MEMORY RECLAMATION PROBLEM



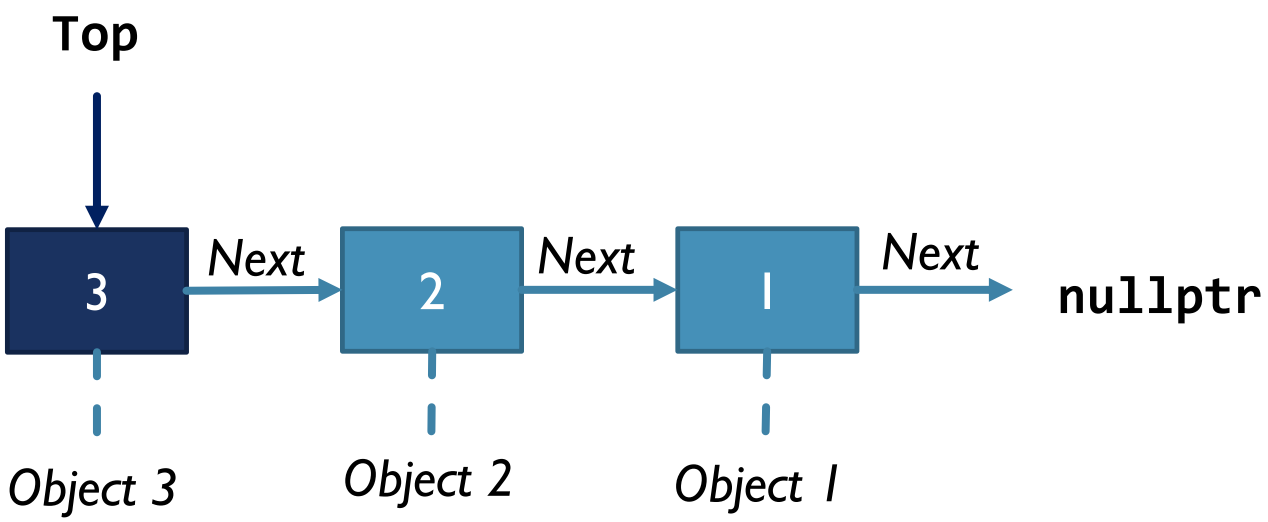
One thread wants to de-allocate a memory block which is still reachable by concurrent threads

MEMORY RECLAMATION PROBLEM



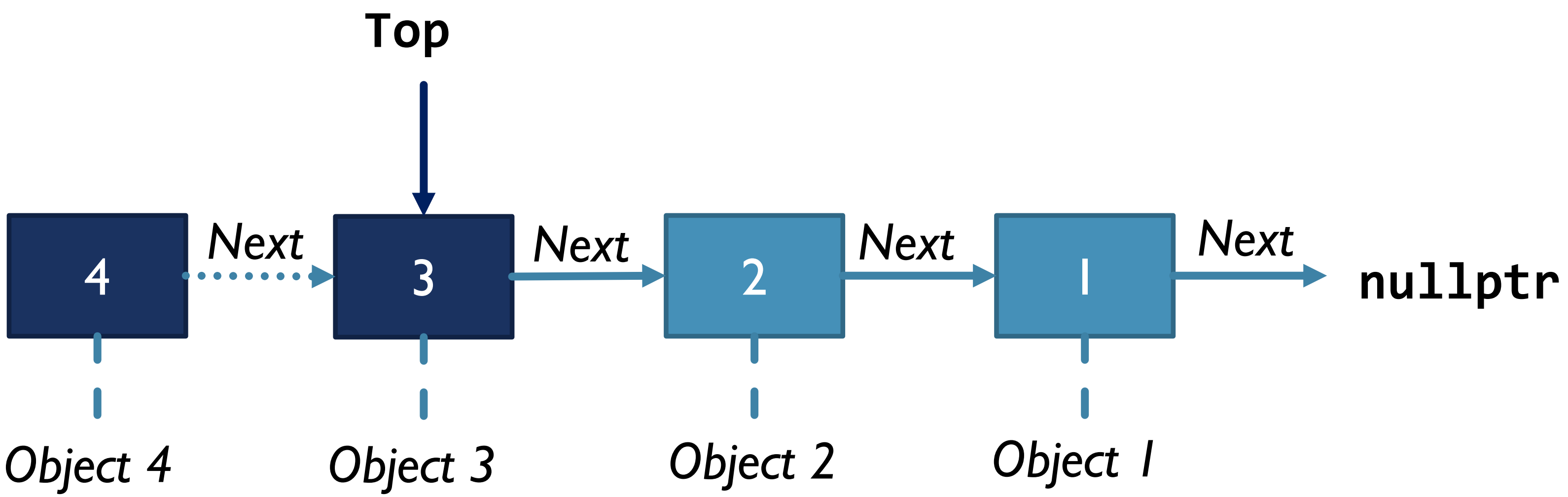
One thread wants to de-allocate a memory block which is still reachable by concurrent threads

TREIBER'S LOCK-FREE STACK



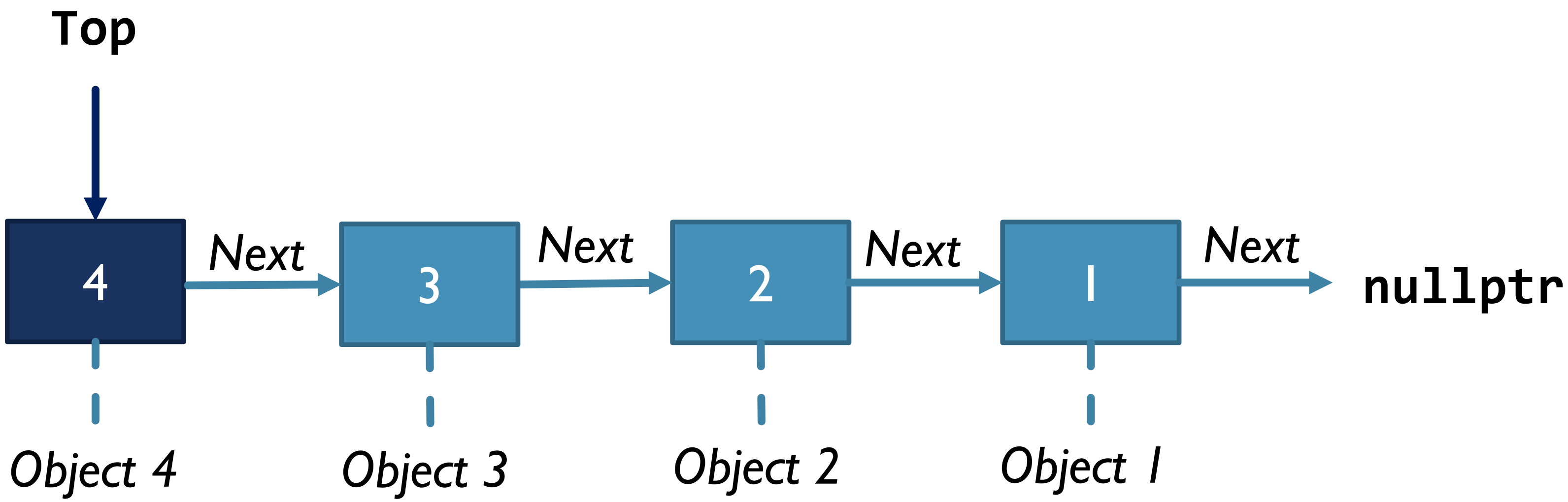
- PUSH and POP operations are implemented by updating **Top** using CAS

TREIBER'S LOCK-FREE STACK



- PUSH and POP operations are implemented by updating **Top** using CAS

TREIBER'S LOCK-FREE STACK



- PUSH and POP operations are implemented by updating **Top** using CAS

EXAMPLE: NO RECLAMATION

```
struct Node {
   Node* next;   // Next element
   Object* obj;  // Stored object
};
Node* Top = nullptr;
```

EXAMPLE: NO RECLAMATION

```
struct Node {
   Node* next;   // Next element
   Object* obj;  // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
   Node* node = malloc(…);
   node->obj = obj;
   while (true) {
      node->next = Top;
      if (CAS(&Top, node->next, node))
         break;
   }
}
```

EXAMPLE: NO RECLAMATION

```
struct Node {
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node = Top;
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            [ delete node ]
            break;
        }
    }
    return obj;
}
```

RECYCLING ELEMENTS

- If we can avoid returning memory to the OS, the simplest approach is to recycle elements
- With simple data structures (such as Treiber's stack) we can easily do so but
  - When calling POP, the same pointer value may point to an already recycled element
  - The problem is known as "the ABA problem" and leads to the data structure corruption
  - Can be solved by using a "tag", which is adjacent to the stack top pointer and incremented each time; the tag uniquely identifies the object
  - Need to use **WCAS** (wide CAS), i.e., cmpxchg16b for x86-64

EXAMPLE: RECYCLING ELEMENTS

```
struct Node {
   Node* next;    // Next element
   Object* obj;   // Stored object
};
<Node*,Int> Top = { nullptr, 0 };

PUSH(Object* obj) {
   Node* node = [ allocate node ]
   node->obj = obj;
   while (true) {
      node->next = Top.Pointer;
      if (WCAS(&Top,
            { node->next, Top.Tag },
            { node, Top.Tag+1 }))
         break;
}  }
```

```
Object* POP() {
   Object* obj = nullptr;
   while (true) {
      Node* node = Top.Pointer;
      if (node == nullptr)
         break;
      if (WCAS(&Top,
            { node, Top.Tag }
            { node->next, Top.Tag+1 })) {
         obj = node->obj;
         [ recycle node ]
         break;
      }
   }
   return obj;
}
```

MORE GENERAL SOLUTION

- Need to postpone de-allocation of this memory block until it is safe to do so
  - But memory usage must be bounded for non-blocking progress guarantees
- Wait-free reclamation is especially difficult
  - No ***universal*** wait-free memory reclamation scheme existed for hand-crafted data structures until recently
  - The fast-path-slow-path [PPoPP ’12] methodology cannot be applied to reclamation directly

QUESTIONS?

EPOCH-BASED RECLAMATION (EBR)

- Uses a ***global*** epoch counter (aka "era" in other algorithms)
- As part of per-thread state, each thread keeps a ***reservation***
- Many variations of EBR exist, which differ on how to increment the epoch counter (conditionally vs. unconditionally) and when to trigger memory reclamation
  - For the original EBR only 3 distinct epoch values are needed
- As an example, consider a variant with unconditional epoch increments presented in [PPoPP '18]

global_epoch = 2

reservations:

Thread 1	[epoch = 1]
Thread 2	[epoch = ∞]
Thread 3	[epoch = 2]
Thread 4	[epoch = ∞]

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

```
PUSH_EBR(Object* obj) {
    begin_op();
    PUSH(obj);
    end_op();
}
```

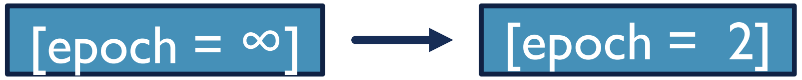
```
Object* POP_EBR() {
    begin_op();
    Object* obj = POP();
    end_op();
    return obj;
}
```

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

global_epoch = 2

```
begin_op() {
   reservations[TID] = global_epoch;
}
```



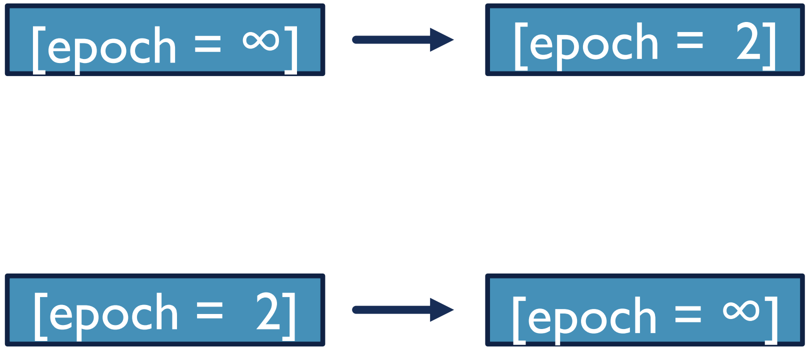
EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

global_epoch = 2

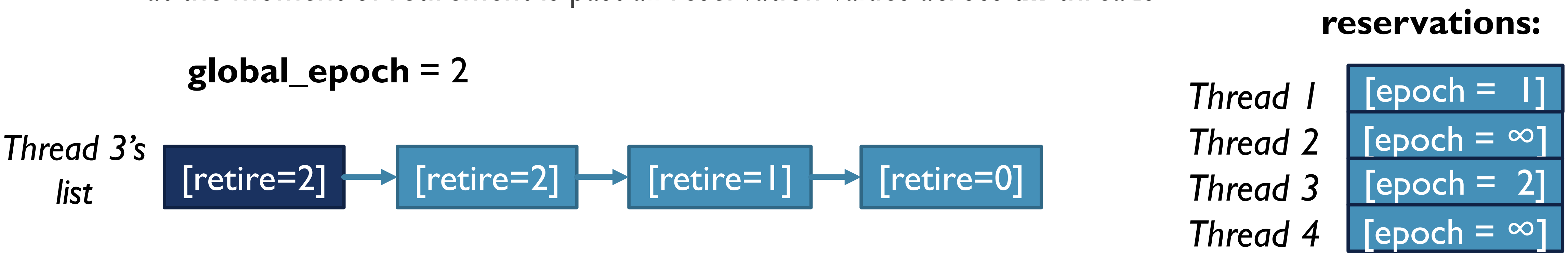
```
begin_op() {
    reservations[TID] = global_epoch;
}

end_op() {
    reservations[TID] = ∞;
}
```



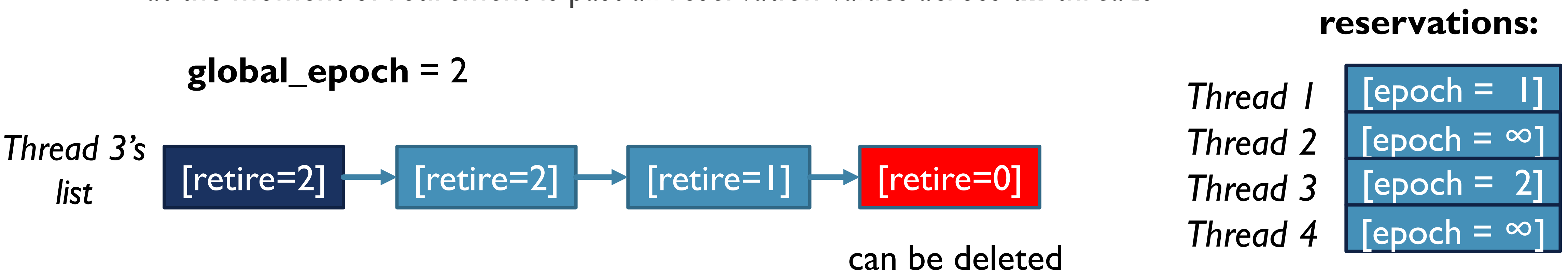
EPOCH-BASED RECLAMATION (EBR)

- When deleting, postpone the actual deallocation by ***retiring*** a memory block
  - Store the global epoch counter at the moment of retiring (“retire epoch”) and place the retired block to a thread-local list
  - Periodically increment the global epoch counter when retiring
  - Periodically scan previously retired blocks from the thread-local list and deallocate those for which the epoch at the moment of retirement is past all reservation values across ***all*** threads



EPOCH-BASED RECLAMATION (EBR)

- When deleting, postpone the actual deallocation by ***retiring*** a memory block
  - Store the global epoch counter at the moment of retiring (“retire epoch”) and place the retired block to a thread-local list
  - Periodically increment the global epoch counter when retiring
  - Periodically scan previously retired blocks from the thread-local list and deallocate those for which the epoch at the moment of retirement is past all reservation values across ***all*** threads



EBR SUMMARY

- EBR tracks memory using "epochs"
  - Simple API
  - Very fast, especially when finding a good balance of how frequently retired nodes need to be scanned
- The scheme is ***blocking***
  - If one thread is stuck and never calls **end_op**(), an unbounded number of blocks can be allocated and never deleted
  - Memory usage is thus unbounded
  - The program can eventually crash when memory is exhausted

HAZARD POINTERS

- Originally published in [TPDS ’04]
- Wrap all pointer dereferences
  - ***Reservations*** keep pointers rather than epochs
  - Since a thread may reserve multiple pointers, several reservations per thread are needed
  - An ***index*** identifies a specific reservation in a thread
- When ***retiring*** a block, put it in a thread-local list
  - Periodically scan the list to check if any of the retired block **pointers** do not overlap with reservations across ***all*** threads
  - Deallocate such blocks

EXAMPLE: HAZARD POINTERS' API

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

- **get_protected():** safely retrieve a pointer to the protected object by creating a reservation

```
PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

- **retire():** mark an object for deletion
  - the retired object must be deleted from the data structure first, i.e., only in-flight threads can still access it

```
PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

EXAMPLE: HAZARD POINTERS' API

- **clear():** reset all prior reservations made by the current thread in get_protected()

```
PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

HAZARD POINTERS' SUMMARY

- Hazard Pointers track memory blocks using pointers
  - Lock-free in general
  - In certain cases can be used in wait-free manner
  - Typically much slower than EBR

COMBINATION OF EBR AND HAZARD POINTERS

- Combine EBR and Hazard Pointers
  - Use epochs (or "eras") for ***reservations***, as in EBR (64-bit values)
  - Wrap all pointer dereferences, as in Hazard Pointers, using **get_protected()**
  - When allocating blocks, initialize them with the current ***global*** epoch value
- Each block records an interval ("allocation" and "retire" epochs)
  - To safely delete a block, its interval must not overlap with ***all*** reservations

COMBINATION OF EBR AND HAZARD POINTERS

- Hazard Eras [SPAA ’17]
  - API is mostly compatible with Hazard Pointers, except when allocating memory blocks
  - Generally much faster than Hazard Pointers
- Interval-Based Reclamation (IBR) [PPoPP ’18]
  - Simpler EBR-like API, but data structures need to modified to restart operations for starving threads
- Turns out that Hazard Eras (unlike Hazard Pointers) can be modified to guarantee wait-freedom
  - Wait-Free Eras (WFE) [PPoPP ’20] is based on Hazard Eras but is wait-free

HAZARD ERAS' API CHANGES

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = alloc_block();
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

HAZARD ERAS' API CHANGES

- **alloc_block():** allocate and initialize a memory block
  - Wraps malloc()
  - Not in the original Hazard Pointers scheme but in Hazard Eras and WFE

```
PUSH(Object* obj) {
    Node* node = alloc_block();
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

OTHER MEMORY RECLAMATION SCHEMES

- Schemes based on lock-free garbage collection
  - Can be unsuitable for C++, especially when using low-level programming models
- Schemes that rely on certain OS primitives or mechanisms
  - QSense [SPAA ’16], DEBRA+ [PODC ’15]
  - Can be convenient for user-space programs but problematic for kernel-space code or for strict non-blocking guarantees since typical OSes use locks

IMPORTANCE OF API FOR NON-BLOCKING PROGRESS

- IBR's API is similar to that of EBR, except it additionally wraps pointer dereferences (no indices needed)
  - Relatively simple, can be hidden inside smart pointers
  - Not always memory-bounded, e.g., when having ***starving*** threads
- The Hazard Eras' and WFE's APIs are based on Hazard Pointers' API
  - Hazard Pointers's API is carefully designed to make sure that a ***finite*** number of blocks are ***reserved*** (i.e., protected from reclamation)

QUESTIONS?

WAIT-FREEDOM CHALLENGE

```
struct Node {
    Reclamation header;
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = alloc_block();
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node =
            get_protected(&Top, 0);
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            retire(node);
            break;
        }
    }
    clear();
    return obj;
}
```

WAIT-FREEDOM CHALLENGE: HAZARD ERAS

```
int reservations[maxThreads][maxHEs];

int global_era = 0;

Node* get_protected(Node** ptr, int indx) {
    int prev = reservations[tid][indx];
    while (true) {
        Node* ret = *ptr;
        int new = global_era;
        if (prev == new)
            return ret;
        reservations[tid][indx] = new;
        prev = new;
    }
}
```

```
retire(Node* node) {
    …
    increment_era();
    …
}

increment_era() {
    F&A(&global_era, 1);
}
```

WAIT-FREEDOM CHALLENGE: HAZARD ERAS

```
int reservations[maxThreads][maxHEs];

int global_era = 0;

Node* get_protected(Node** ptr, int indx) {
   int prev = reservations[tid][indx];
   while (true) {
      Node* ret = *ptr;
      int new = global_era;
      if (prev == new)
         return ret;
      reservations[tid][indx] = new;
      prev = new;
   }
}
```

```
retire(Node* node) {
   …
   increment_era();
   …
}

increment_era() {
   F&A(&global_era, 1);
}
```

TIMNAT AND PETRANK'S FORMULATION

- [PPoPP '14] proposed a method to automatically convert lock-free data structures into wait-free ones
- The original lock-free data structure needs to be written in a "normalized" form
- Normalized data structures are defined in [PPoPP '14]
  - One of the key requirements is "*Any modification of the shared data structure is executed using a CAS operation*"
- Operations can be restarted if things go wrong, therefore **get_protected()** does not need to be unbounded
  - Examples include [PPoPP '17]'s implementations of CRTurnQueue and KPQueue using Hazard Pointers

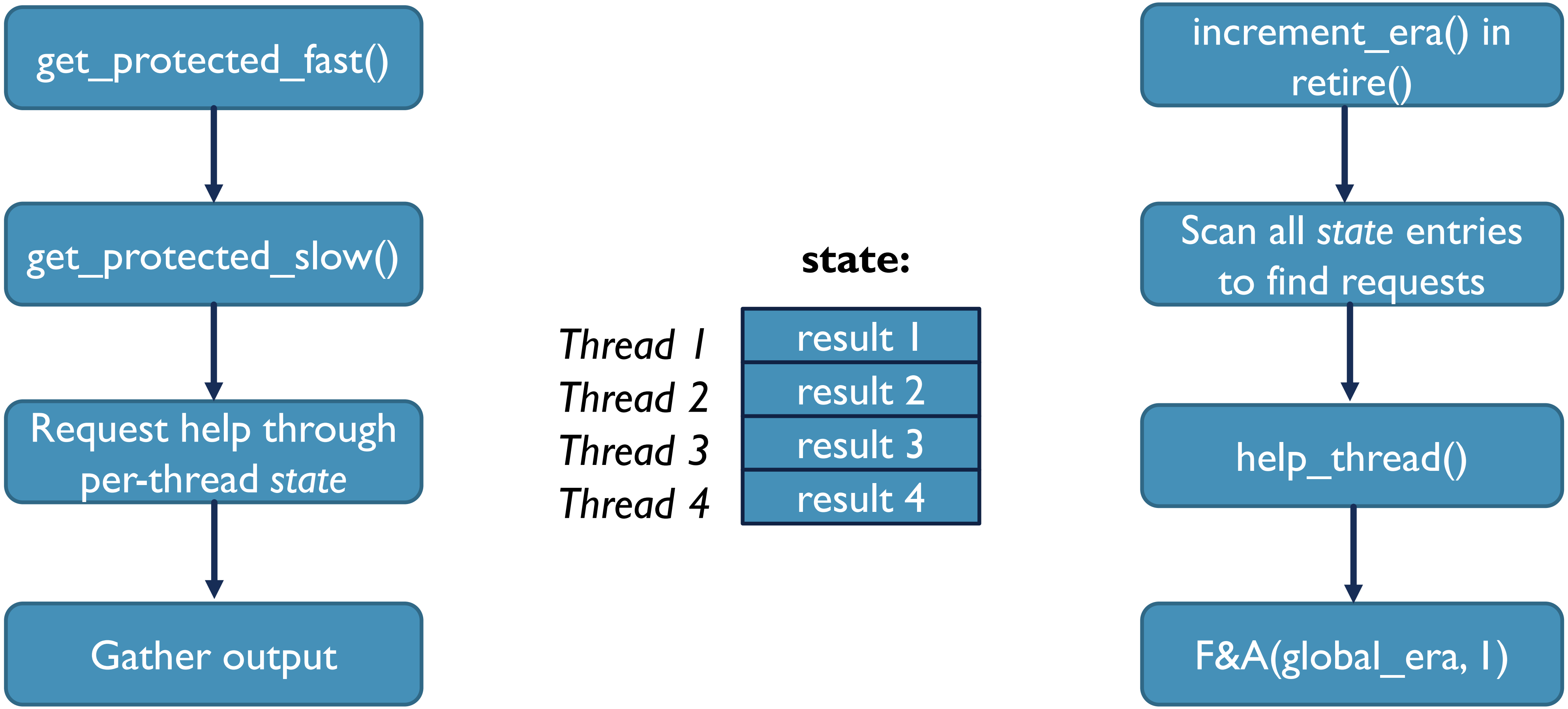
WAIT-FREE ERAS (WFE)

- Although wait-free reclamation is feasible in special cases, it is much harder to guarantee for arbitrary formulated wait-free data structures
  - Specialized instructions such as F&A can still be useful in wait-free data structures for performance reasons
  - Even CAS-only wait-free data structures are not necessarily derived from "normalized" form
- Our recent [PPoPP '20] publication, Wait-Free Eras (WFE), solves wait-free memory reclamation for a more general case

WAIT-FREE ERAS (WFE)

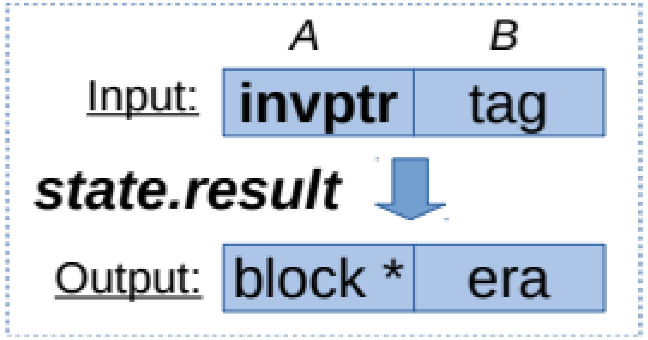
- Bird's-eye view
  - Use a fast-path-slow-path method for **get_protected()**
  - **retire()** increments the global era (or alternatively **alloc_block()**): it calls a helper method before incrementing the era clock
- Wait-free consensus is achieved with the help of
  - **F&A**: available on x86-64 and AArch64 as of v8.1 and suitable for wait-free algorithms due to bounded execution time
  - **WCAS**: also available on x86-64 and AArch64

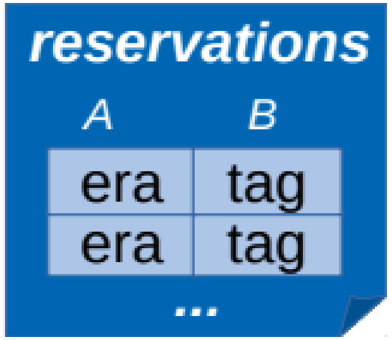
WAIT-FREE ERAS (WFE)



WAIT-FREE ERAS (WFE)

- Introduce tags to identify slow-path cycles
  - They prevent spurious (belated) updates
- Per-thread state: result is used for both input and output
  - Use pairs for result { .A, .B }
- Reservations also use pairs { .A, .B }
  - Two special reservations for helpers (maxHEs, maxHEs+1), i.e., the total number is maxHEs+2





WAIT-FREE ERAS (WFE)

```
block* get_protected_slow(block** ptr, int indx, block* parent) {
   int allocEra = parent->allocEra;
   int tag = reservations[tid][indx].B;

   state[tid][indx].ptr = ptr;
   state[tid][indx].era = allocEra;
   state[tid][indx].result = { invptr, tag };
```

WAIT-FREE ERAS (WFE)

```
block* get_protected_slow(block** ptr, int indx, block* parent) {
   int allocEra = parent->allocEra;
   int tag = reservations[tid][indx].B;

   state[tid][indx].ptr = ptr;
   state[tid][indx].era = allocEra;
   state[tid][indx].result = { invptr, tag };
   …
   // Try retrieving a pointer in a loop
```

WAIT-FREE ERAS (WFE)

```
block* get_protected_slow(block** ptr, int indx, block* parent) {
    int allocEra = parent->allocEra;
    int tag = reservations[tid][indx].B;

    state[tid][indx].ptr = ptr;
    state[tid][indx].era = allocEra;
    state[tid][indx].result = { invptr, tag };
    …
    // Try retrieving a pointer in a loop
    …
    if (result.A != invptr) {
        int era = result.B;
        reservations[tid][indx].A = era;
        reservations[tid][indx].B = tag+1;
        return result.A;
}   }
```

WAIT-FREE ERAS (WFE)

```
help_thread(int i, int j, int tid) {
    int_pair result = state[i][j].result;
    if (result.A != invptr)
        return;
    int era = state[i][j].era;
    reservations[tid][maxHEs].era = era;
    block** ptr = state[i][j].ptr;
    int tag = reservations[i][j].B;
    if (result.B != tag) {
        reservations[tid][maxHEs].era = ∞;
        return;
    }
    …
}
```

WAIT-FREE ERAS (WFE)

```
help_thread(int i, int j, int tid) {
    …
    int prev = global_era;
    do {
        reservations[tid][maxHEs+1].A = prev;
        block* ret_ptr = *ptr;
        int new = global_era;
        if (prev == new) {
            // DONE! Can produce the result
            break;
        }
        prev = new;
    } while (state[i][j].result == { invptr, tag });
    reservations[tid][maxHEs+1].era = ∞;
    reservations[tid][maxHEs].era = ∞;
}
```

WAIT-FREE ERAS (WFE)

```
help_thread(int i, int j, int tid) {
   …
   int prev = global_era;
   do {
      reservations[tid][maxHEs+1].A = prev;
      block* ret_ptr = *ptr;
      int new = global_era;
      if (prev == new) {
         // DONE! Can produce the result
         break;
      }
      prev = new;
   } while (state[i][j].result == { invptr, tag });
   reservations[tid][maxHEs+1].era = ∞;
   reservations[tid][maxHEs].era = ∞;
}
```

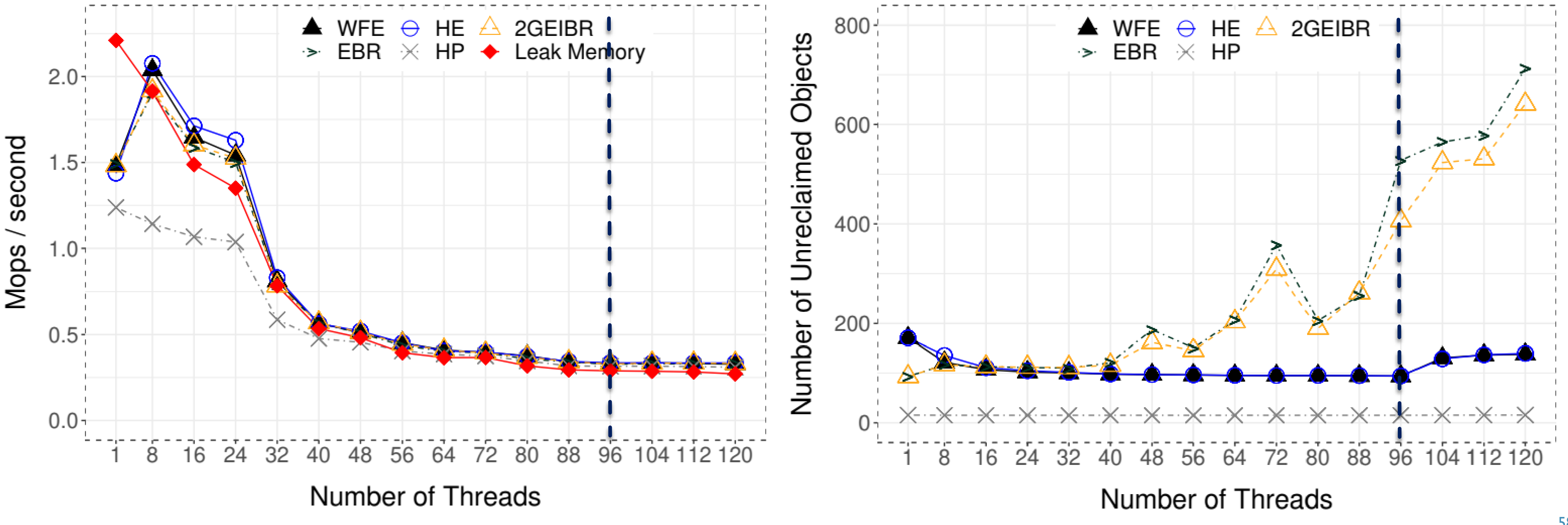
WAIT-FREE ERAS (WFE)

- Avoiding race conditions when scanning deleted nodes
  - Check reservations 0..maxHEs-1
  - Check reservations maxHEs, maxHEs+1
  - Check reservations 0..maxHEs-1 again

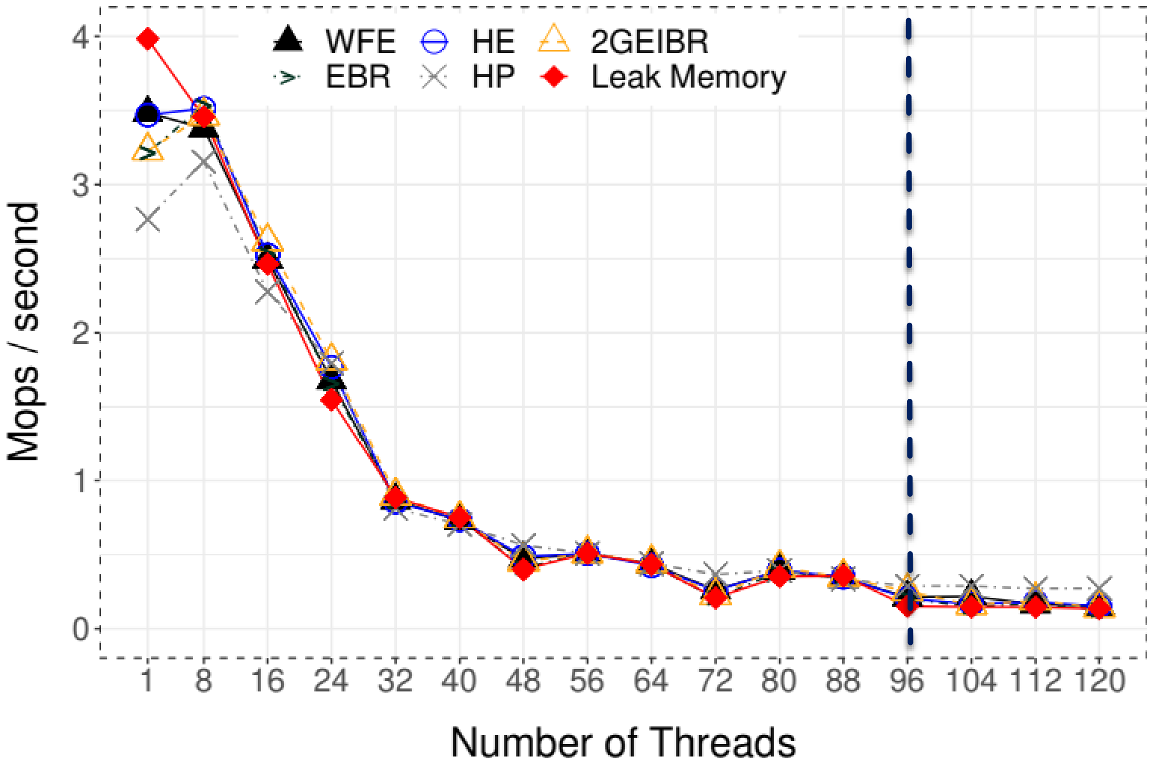
EVALUATION

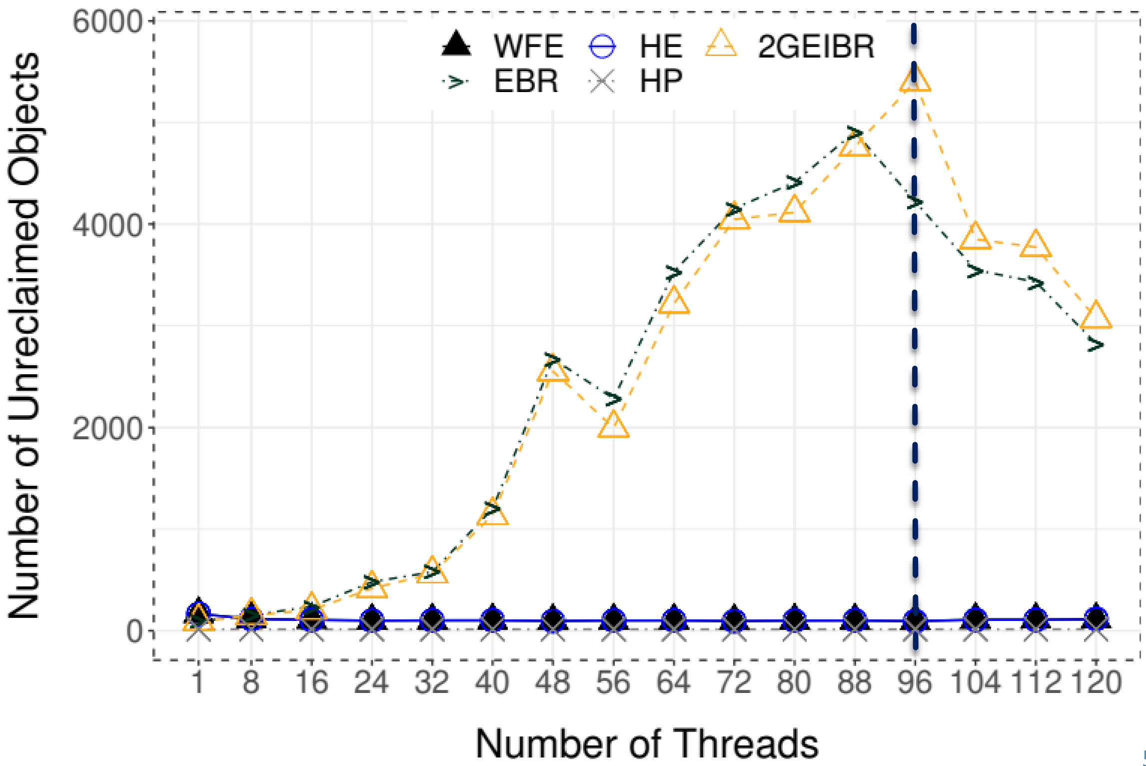
- 4x24 Intel Xeon E7-8890 v4 (2.20GHz) 256GB RAM, GCC 8.3.0 with -O3
- Using the benchmark from IBR/PPoPP ’18 (by Wen et al.) comparing:
  - *Wait-Free Eras (**WFE**)* [PPoPP ’20]
  - *Hazard Eras (**HE**)* [SPAA ’17]
  - *Interval-Based Reclamation, 2GEIBR (**IBR**)* [PPoPP ’18]
  - *Epoch-Based Reclamation (**EBR**)*
  - *Hazard Pointers (**HP**)* [TPDS ’04]
  - *No reclamation (**Leak Memory**)*
- Results are for write-intensive (50% insert, 50% delete) tests
  - See WFE/PPoPP ’20 for read-mostly (90% get, 10% put) results

EVALUATION: KOGAN AND PETRANK'S WAIT-FREE QUEUE

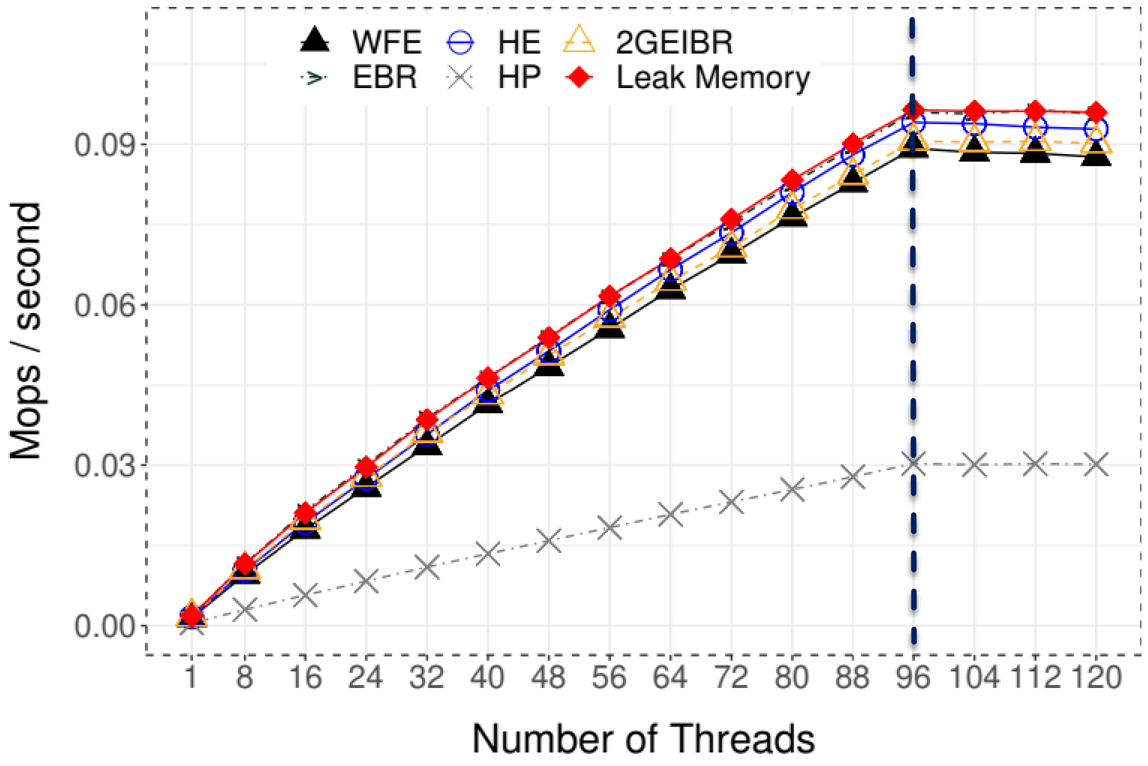


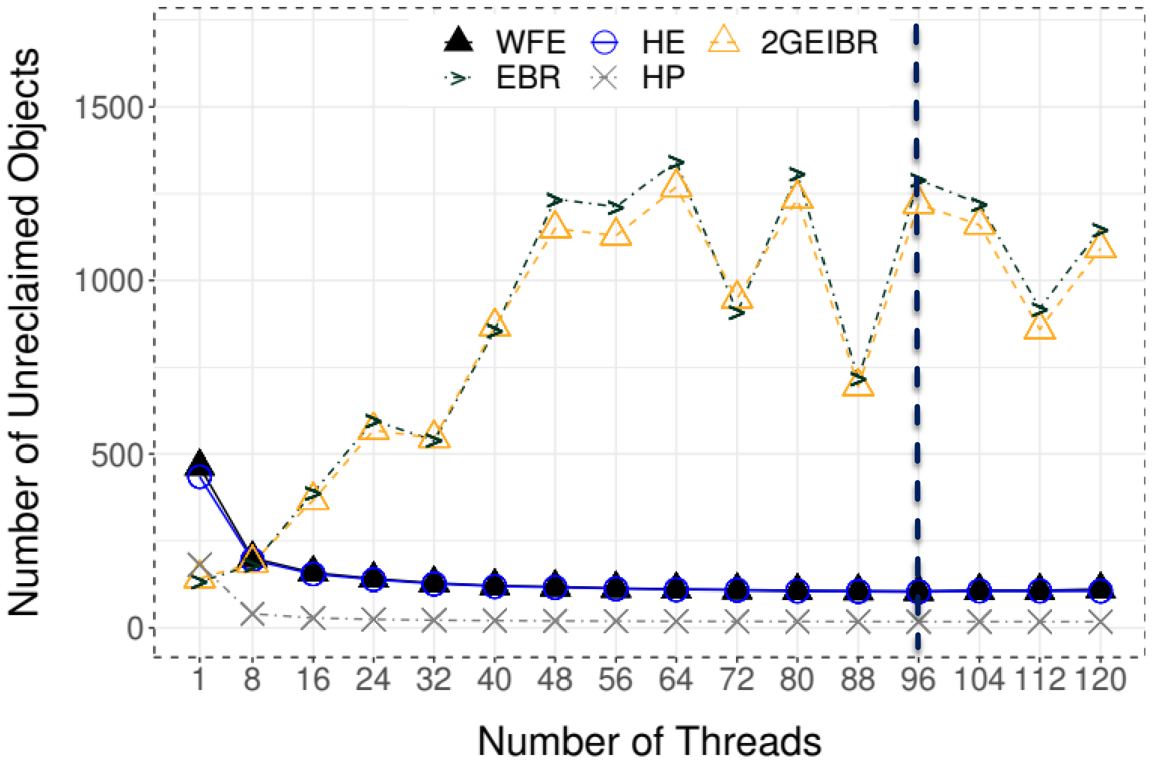
EVALUATION: CRTURN WAIT-FREE QUEUE



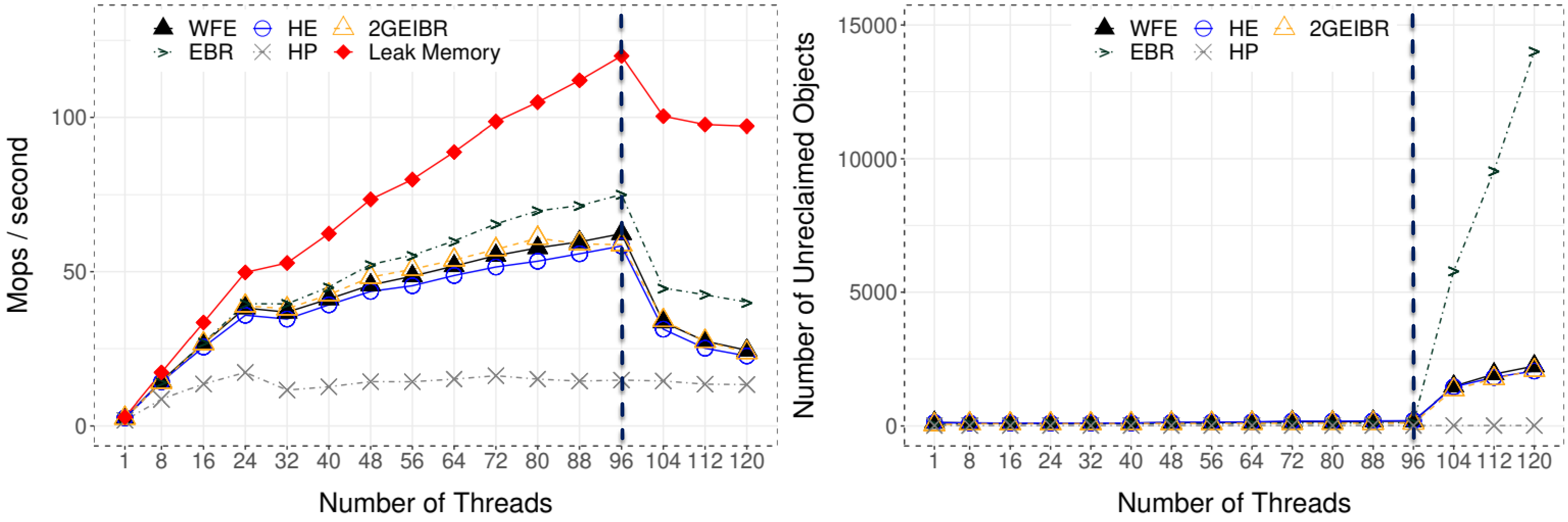


EVALUATION: SORTED LOCK-FREE LINKED LIST

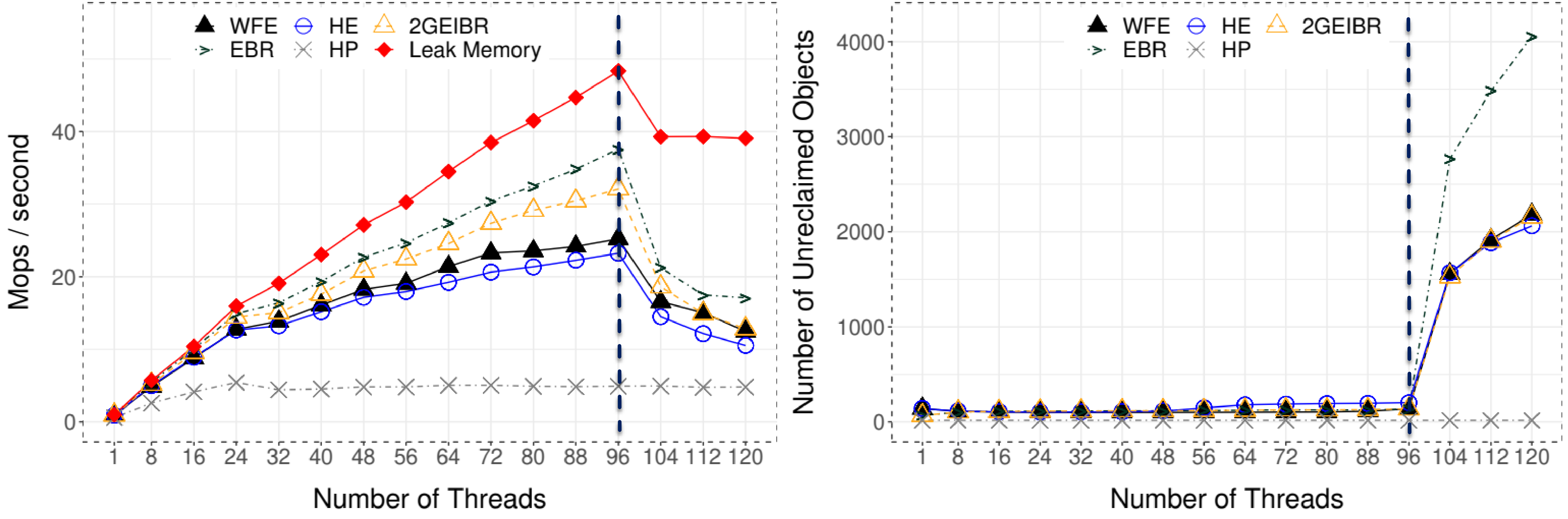




EVALUATION: LOCK-FREE HASH MAP



EVALUATION: LOCK-FREE NATARAJAN TREE



CONCLUSIONS

- Concurrent data structures require careful consideration of the memory reclamation problem
- Memory reclamation itself is subject to progress guarantee requirements
- Wait-free reclamation is feasible through WFE
  - Opens the way for wide adoption of wait-free data structures
  - The only remaining obstacle is efficient wait-free allocation and deallocation
  - Can spur further research in wait-free reclamation

AVAILABILITY

- My personal website
  - https://rusnikola.github.io
- WFE's code
  - https://github.com/rusnikola/wfe

AVAILABILITY

- My personal website
  - https://rusnikola.github.io
- WFE's code
  - https://github.com/rusnikola/wfe

THANK YOU!