Universal *Wait-Free* Memory Reclamation

Ruslan Nikolaev, Binoy Ravindran
Systems Software Research Group
Virginia Tech, USA

Wait-Freedom

- *Wait-freedom* is the strongest form of non-blocking progress, where all threads complete operations in a finite number of steps
- Wait-free algorithms are gaining more practical relevance and efficiency (Kogan and Petrank's *fast-path-slow-path* methodology in PPoPP'12)

Problem and Motivation

- One thread wants to de-allocate a memory block which is still reachable by concurrent threads
- Need to postpone de-allocation of this memory block until it is safe to do so
- No universal wait-free memory reclamation scheme exists for hand-written data structures
  - The methodology cannot be used for reclamation

Existing Approaches

- OneFile [DSN'19]: wait-free STM with built-in memory reclamation (not for arbitrary data structures)
- Epoch-Based Reclamation: simple but blocking, memory usage is unbounded
- Hazard Pointers [TPDS'04]: lock-free in general; in certain cases can be used in a wait-free manner
- Hazard Eras [SPAA'17]: lock-free, the same API but uses "eras" (epochs)
- IBR [PPoPP'18]: simpler API but starving threads need to be handled separately
- WFE (*Wait-Free Eras*) extends Hazard Eras to implement wait-freedom

Hazard Eras API

- **get_protected()**: safely retrieve a pointer to the protected object by creating a reservation
- **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
- **clear()**: reset all prior reservations made by the current thread in get_protected()
- **alloc_block()**: allocate a memory block and initialize its *alloc era* to the *global era* clock value

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

Usage example

```
struct node_s {
   block header; // 1:HE header
   node_s* next; // 2:Next element
   void* obj;    // 3:Stored object
};

void enqueue(void* obj) {
   node_s* node =
      alloc_block(sizeof(node_s));
   node->obj = obj;
   do {
      node->next = stack;
   } while (!CAS(&stack,
                 node->next, node));
}
```

```
void* dequeue() {
   void* obj = nullptr;
   while (true) {
      node_s* node =
         get_protected(&stack, 0);
      if (!node) break;
      if (CAS(&stack, node,
            node->next) {
         obj = node->obj;
         retire(node);
         break;
      }
   }
   clear();
   return obj;
}
```

get_protected()

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

int global_era = 0;

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

```
block* alloc_block() {
   …
   increment_era();
   …
}

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

get_protected()

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

int global_era = 0;

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

```
block* alloc_block() {
   …
   increment_era();
   …
}

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

Bird's-Eye View of WFE

- Use a fast-path-slow-path method for **get_protected()**
- **alloc_block()** or **retire()** increments the global era, call a helper method *before* incrementing the era

Fast-path-slow-path

```
block* get_protected(block** ptr,
            int indx, block* parent) {
   int prev = reservations[tid][indx];
   int attempts = maxAttempts;
   while (--attempts != 0) {
      block* ret = *ptr;
      int new = global_era;
      if (prev == new) return ret;
      reservations[tid][indx] = new;
      prev = new;
   }
   …
   block* ret = get_protected_slow(
            ptr, indx, parent);
   …
   return ret;
}
```

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

Fast-path-slow-path

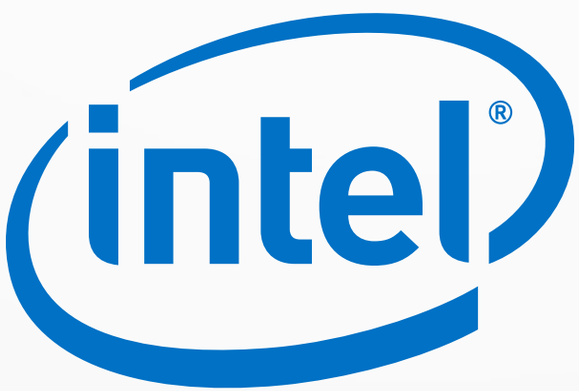
```
block* get_protected(block** ptr,
        int indx, block* parent) {
   int prev =
        reservations[tid][indx];
   int attempts = maxAttempts;
   while (--attempts != 0) {
      block* ret = *ptr;
      int new = global_era;
      if (prev == new) return ret;
      reservations[tid][indx] = new;
      prev = new;
   }
   F&A(&counter_start, 1);
   block* ret = get_protected_slow(
              ptr, indx, parent);
   F&A(&counter_end, 1);
   return ret;
}
```

```
int counter_start = 0;
int counter_end = 0;

void increment_era() {
   int ce = counter_end;
   int cs = counter_start;
   if (cs != ce) {
      for i: 0..maxThreads-1 do
         for j: 0..maxHEs-1 do
            help_thread(i, j, tid);
   }
   F&A(&global_era, 1);
}
```

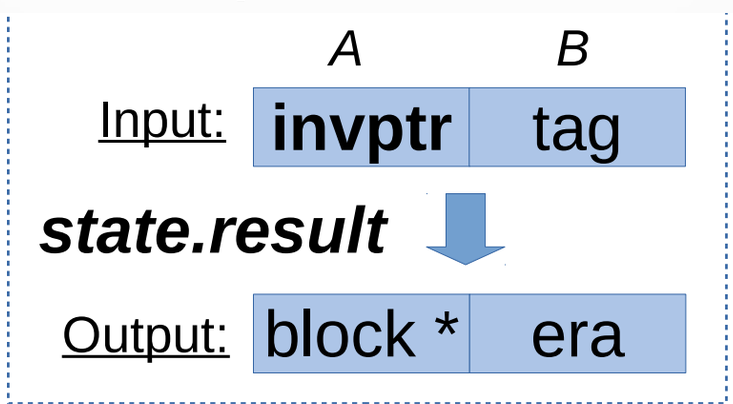
WFE's Wait-Free Consensus

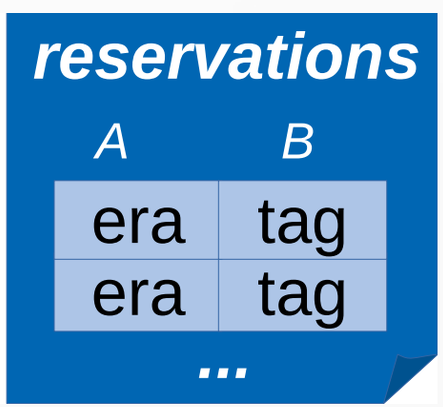
- F&A (fetch-and-add): available on x86-64 and AArch64 as of v8.1 and suitable for wait-free algorithms due to bounded execution time
- WCAS (wide CAS): also available on x86-64 and AArch64
- Both instructions help to solve wait-free consensus



WFE's state

- Tags identify slow-path cycles
- 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





get_protected_slow()

```
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 to retrieve a pointer
   WCAS(&reservations[tid][indx], { prev, tag }, { new, tag} );
   ...
   if (result.A != invptr) {
      int era = result.B;
      reservations[tid][indx].A = era;
      reservations[tid][indx].B = tag+1;
      return result.A;
   }
}
```

get_protected_slow()

```
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 to retrieve a pointer
   WCAS(&reservations[tid][indx], { prev, tag }, { new, tag} );
   ...
   if (result.A != invptr) {
      int era = result.B;
      reservations[tid][indx].A = era;
      reservations[tid][indx].B = tag+1;
      return result.A;
   }
}
```

get_protected_slow()

```
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 to retrieve a pointer
   WCAS(&reservations[tid][indx], { prev, tag }, { new, tag} );
   ...
   if (result.A != invptr) {
      int era = result.B;
      reservations[tid][indx].A = era;
      reservations[tid][indx].B = tag+1;
      return result.A;
   }
}
```

Helper Method

```
void 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;
   }

   …

}
```

Helper Method

```
void 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) {
         // CONVERGED: can produce the result
         break;
      }
      prev = new;
   } while (state[i][j].result == { invptr, tag });

   reservations[tid][maxHEs+1].era = ∞;
   reservations[tid][maxHEs].era = ∞;
}
```

Helper Method

```
void 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) {
         // CONVERGED: can produce the result
         break;
      }
      prev = new;
   } while (state[i][j].result == { invptr, tag });

   reservations[tid][maxHEs+1].era = ∞;
   reservations[tid][maxHEs].era = ∞;
}
```

Helper Method (converged)

```
if (prev == new) {
   if (WCAS(&state[i][j].result,
                { invptr, tag }, { ret_ptr, new })) {
      // Change helpee's reservation (2 iterations at most)
      do {
         old = reservation[i][j];
         if (old.B != tag)
            break;
         complete = WCAS(&reservation[i][j], old, { new, tag+1 });
      } while (!complete);
   }
   break;
}
```

Race conditions

- When moving reservations from one thread to another, we need to avoid race conditions
  - Check reservations 0..maxHEs-1
  - Check reservations maxHEs, maxHEs+1
  - Check reservations 0..maxHEs-1 again

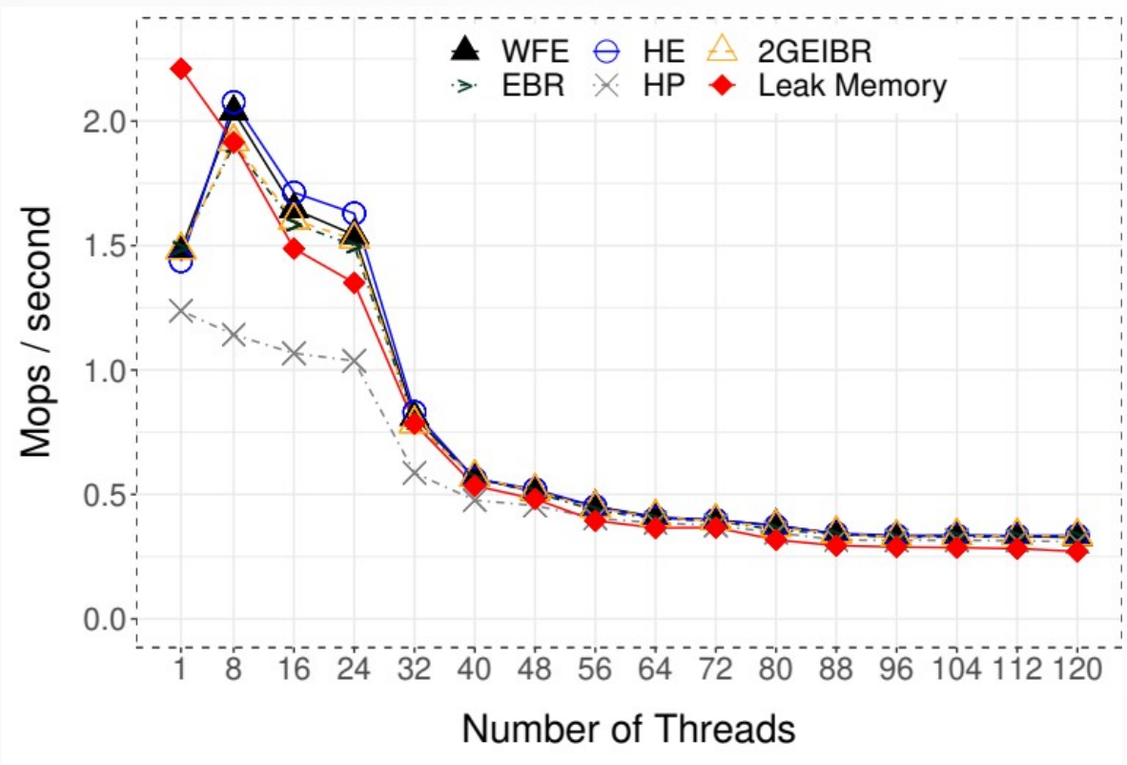
Evaluation

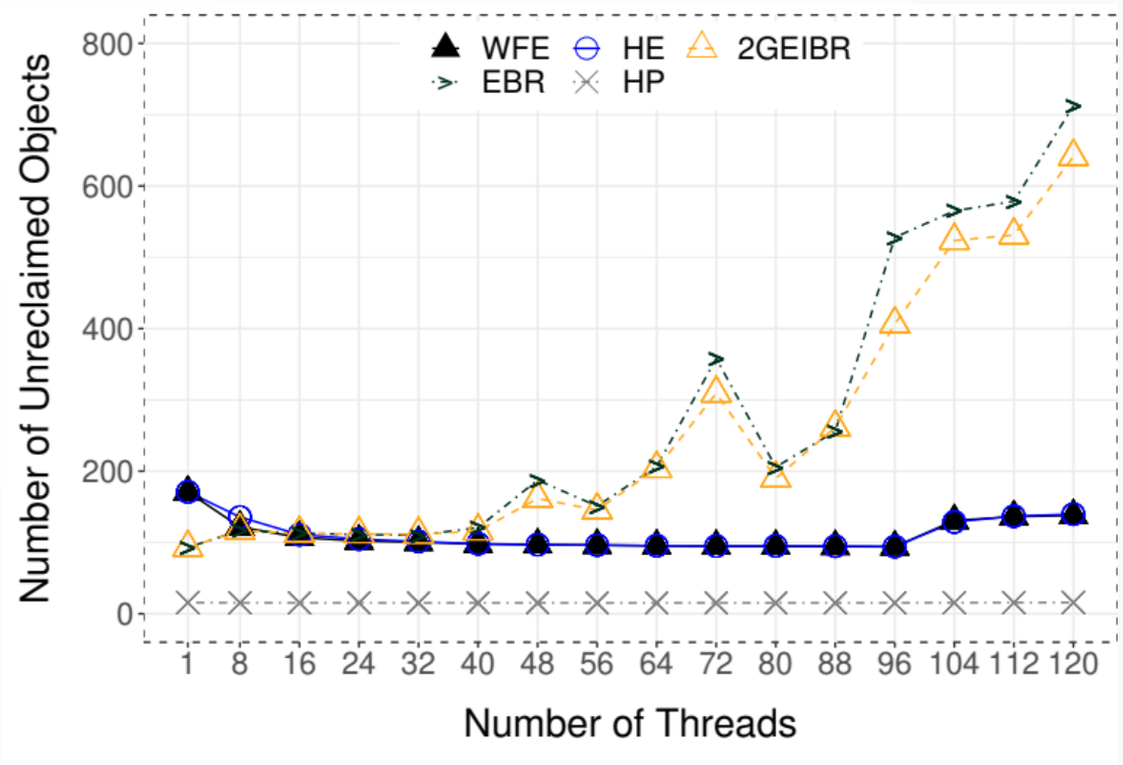
- 4x24 Intel Xeon E7-8890 v4 (2.20GHz)
- 256GB RAM
- gcc 8.3.0 with -O3
- Benchmark from IBR/PPoPP'18 (by Wen et al.)
- Extended with Kogan-Petrank (PPoPP'11) and CRTurn (by Ramalhete, et al., PPoPP'17) wait-free queues
- Queues are derived from an implementation for Hazard Pointers in CRTurn/PPoPP'17

Evaluation

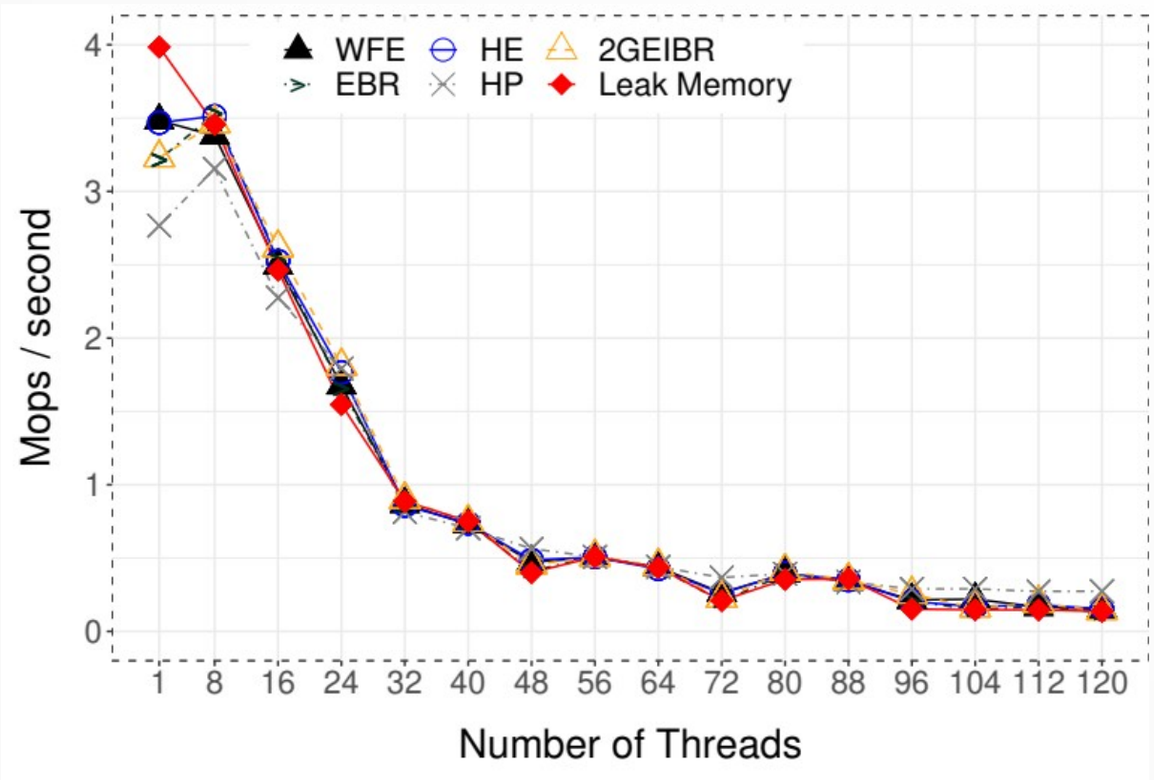
- WFE is compared against:
  - Hazard Eras (HE) [SPAA'17]
  - Interval-Based Reclamation, 2GEIBR (IBR) [PPoPP'18]
  - Epoch-based Reclamation (EBR)
  - Hazard Pointers (HP) [TPDS'04]
- Default benchmark parameters (as in IBR/PPoPP'18)
- Paper presents results for:
  - Write-intensive (50% insert, 50% delete) tests
  - Read-mostly (90% get, 10 put) tests

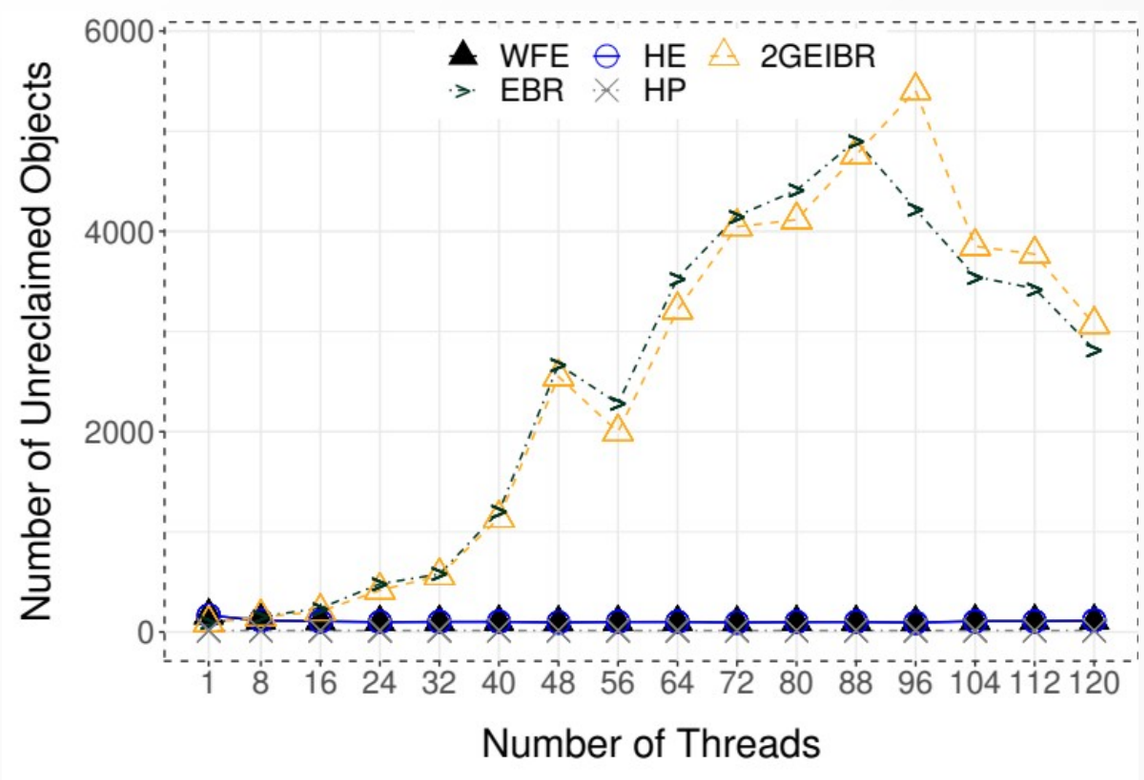
Kogan-Petrank's Queue



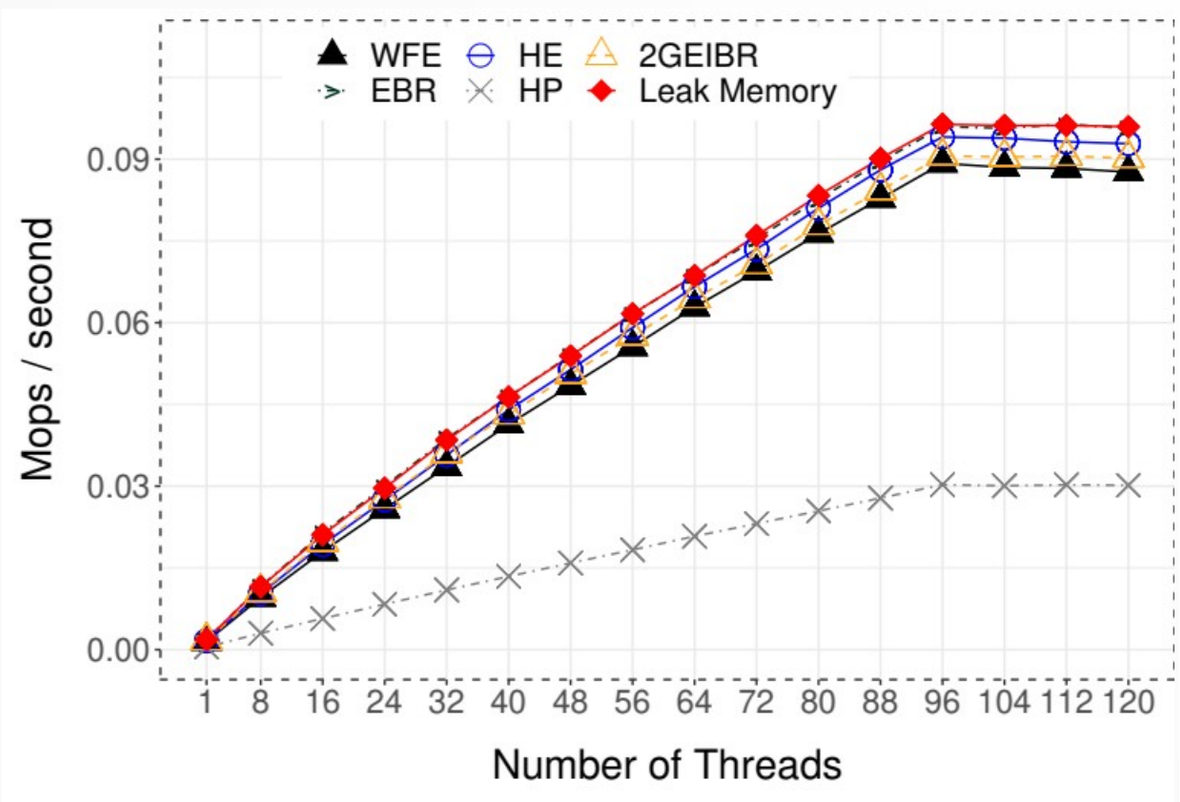


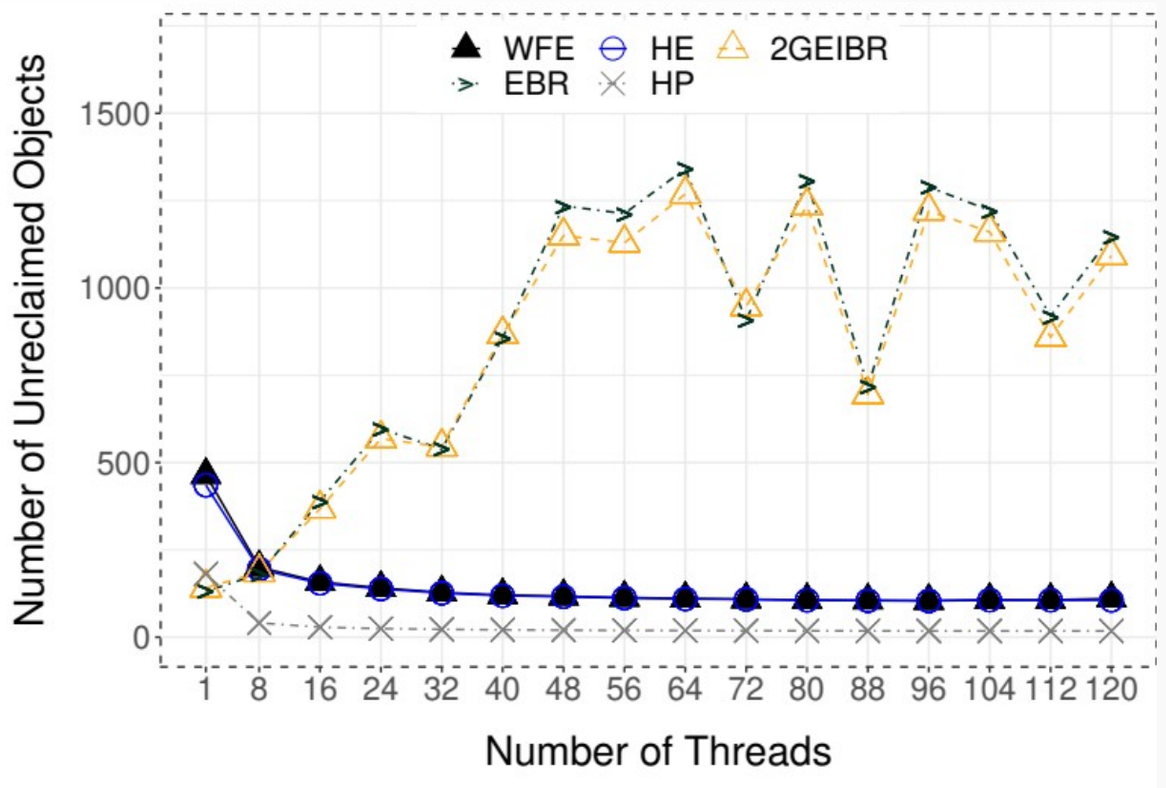
CRTurn Wait-Free Queue



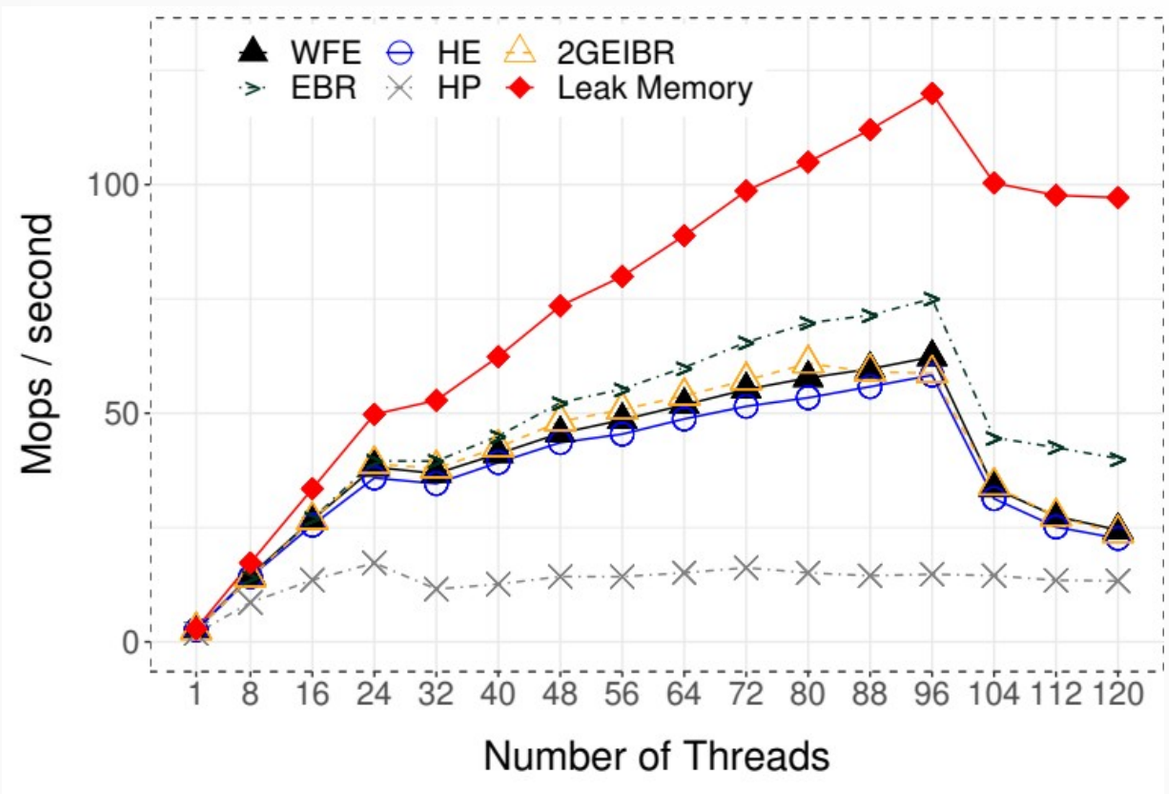


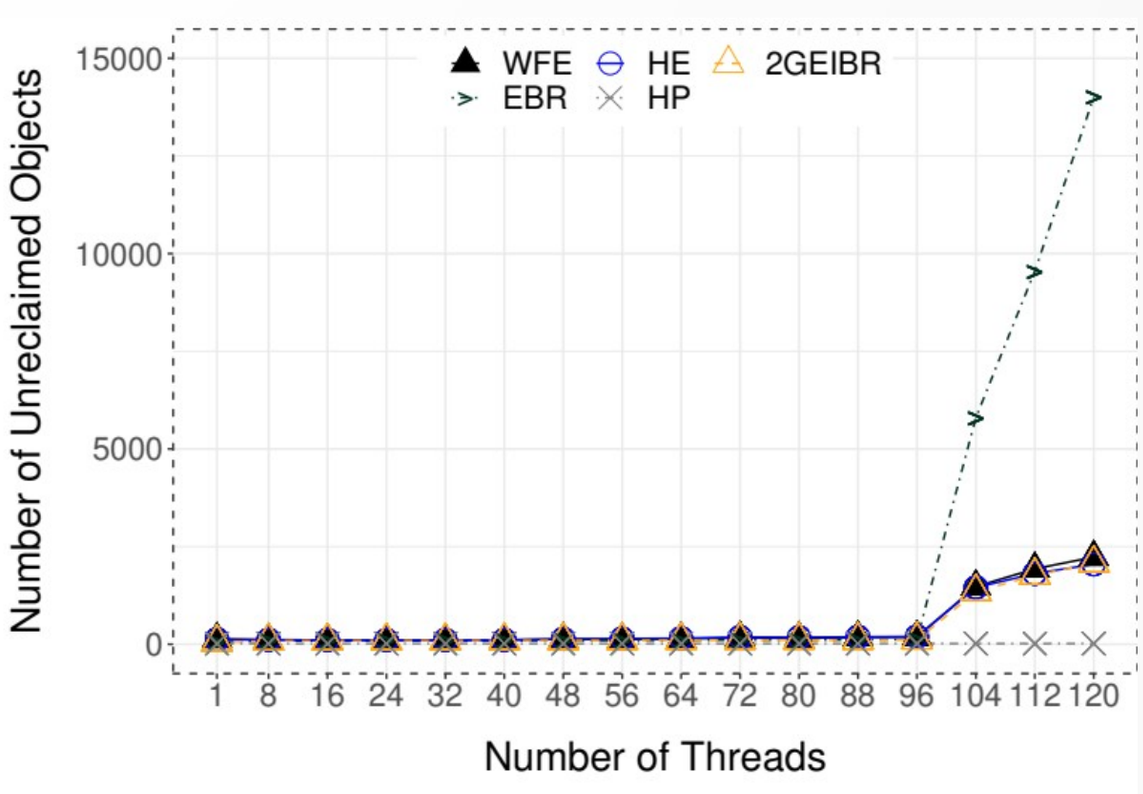
Sorted Lock-Free LinkedList



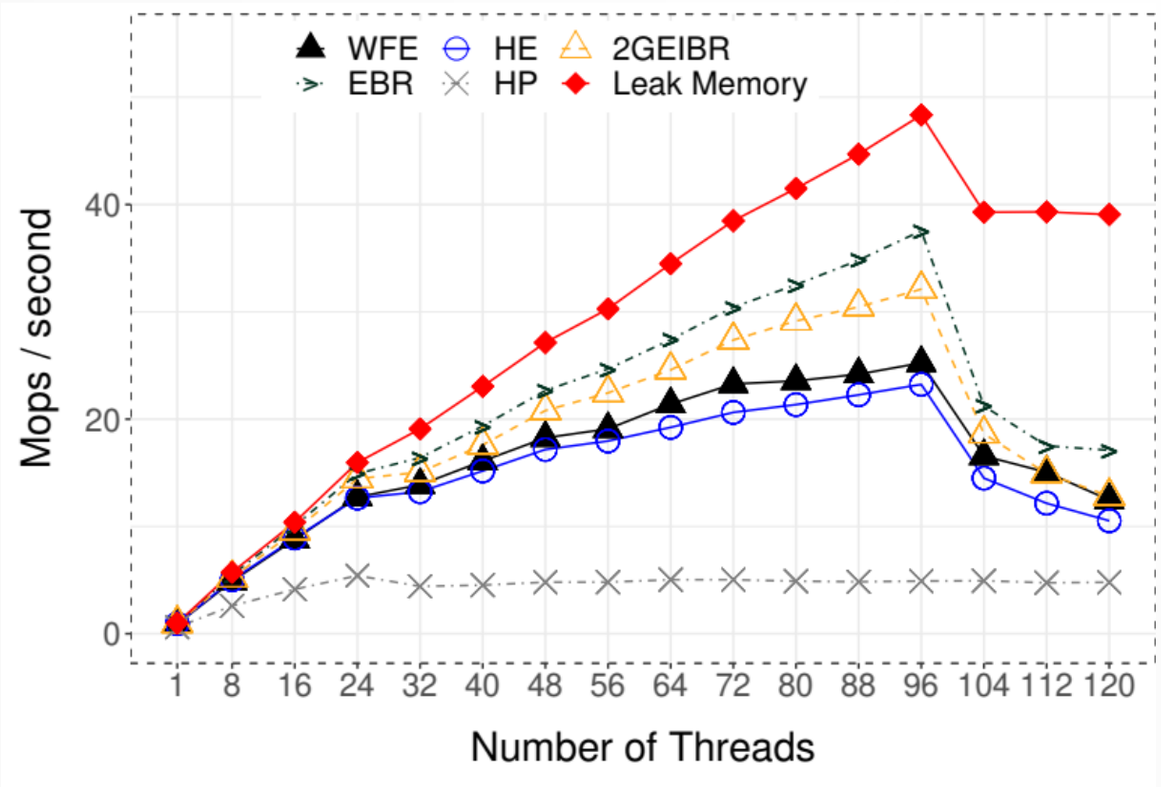


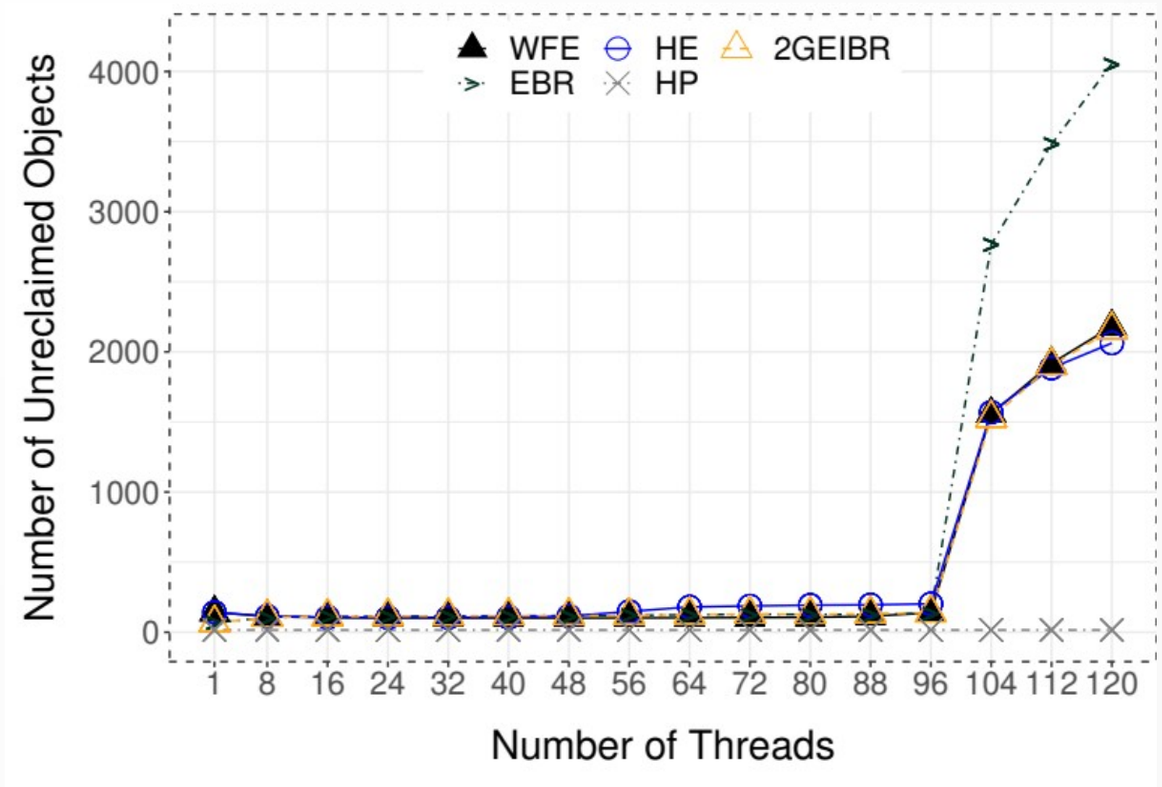
Lock-Free HashMap





Lock-Free NatarajanTree





Conclusions

- Memory reclamation is challenging
  - Any non-blocking progress implies bounded memory usage
  - Classical methodologies are not directly applicable (e.g., they need to allocate or reclaim descriptors)
    - *The “chicken-and-egg” problem*
  - Hazard Eras seemed easier to extend than Hazard Pointers because we can control how eras are incremented
  - We had to use special CPU instructions (WCAS and F&A) for our wait-free consensus protocol

Code Availability

- WFE's code is available at https://github.com/rusnikola/wfe

Thank you!

Artwork attribution: *wikipedia.org (Intel, AMD64, ARM logos) and intel.com (Xeon logo)*