# A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation

Supplemental Material (Anonymized)

## 1 Extended Correctness Discussion

Crystalline-L/-LW/-W are based on Hyaline-1S, for which correctness is discussed in [2]. We present memory bounds arguments which are critical for lock-free guarantees. We further discuss aspects that pertain to wait-free progress.

**Theorem 1.** *Crystalline-L and Crystalline-W are fully memory bounded.*[1]

*Proof.* Batch sizes are constrained by MAX_THREADS×MAX_IDX+1 nodes. In the worst case, each batch is attached to every reservation. The total number of reservations is MAX_THREADS×MAX_IDX. Consequently, the memory usage is bounded by $(\text{MAX\_THREADS}\times\text{MAX\_IDX}+1)^2$. Since era increments are also amortized, the total cost is $\text{RETIRE\_FREQ}\times(\text{MAX\_THREADS}\times\text{MAX\_IDX}+1)^2$. (Note that for Crystalline-W, MAX_IDX+2 should be used rather than MAX_IDX since the total number of indices is bigger.)

Note that this theoretical upper bound is worse than that of HE [3] or WFE [1]. However, in practice, batches do not accumulate this (worst-case) number of nodes and are retired much faster, often resulting in better practical efficiency. Regardless of that, this worst-case bound is still reasonable and finite. □

**Lemma 1.** `traverse()` *calls are wait-free bounded.*

*Proof.* The loop in `traverse()` is bounded by the length of the list of retired batches at a given reservation. `try_retire()` attaches (Line 58, Figure 2) only those batches for which the minimum birth era overlaps with the reservation's era. (Note that every time the era is updated by `update_era()`, the list is emptied.) Since the eras are periodically incremented in `alloc_node()`, the number of such nodes, and consequently – batches, is finite. □

**Lemma 2.** `detach_nodes()`*'s loop is bounded by MAX_THREADS iterations.*

*Proof.* `detach_nodes()` makes the tag->tag+1 (odd) transition in the slow path. The CAS operation in Line 24, Figure 3, changes the era tag unless it was already changed by a concurrent thread. For the loop in Lines 26-31, Figure 3 to continue, the list tag should have not yet moved to the tag+1 state. Because the era tag moves to the tag+2 (even) state only after that transition (Line 43 or Lines 83-87, Figure 4), i.e., after `detach_nodes()` in Line 38 or 81 (Figure 4), the era tag is still tag+1. Consequently, all contending threads in `try_retire()` will find that the era tag is odd (Line 46, Figure 2) and will skip the corresponding reservation for retirement. (Note that skipping nodes is safe because this race window is handled later by Lines 46 and 89, Figure 4.) Only threads that are already in-progress will proceed and potentially contend because of unconditional list pointer updates in Line 58, Figure 2. The number of such threads is bounded by MAX_THREADS, as subsequent `try_retire()` calls will observe that the era tag is odd. □

**Lemma 3.** *The loop in Lines 92-99, Figure 4 is bounded by at most MAX_THREADS iterations.*

[1] This is not generally true for Hyaline-1S due to starving threads.

*Proof.* The proof is similar to that of Lemma 2. The only difference is that it makes the tag+1->tag+2 (even) transition in the slow path. Consequently, `try_retire()` will find that the list tag is odd (Line 44, Figure 2). □

**Lemma 4.** *The loop in Lines 13-37, Figure 4 is bounded by at most MAX_THREADS iterations.*

*Proof.* In Line 11, Figure 4, a thread advertises that it needs help. The loop in Lines 13-37 can only fail to converge because of *global_era* updates. At most MAX_THREADS already in-progress threads are executing *increment_era()* from *alloc_block()*, prior to Line 35, Figure 2, which updates *global_era*, but after Line 32, Figure 2, which detects what threads need helping. All these threads will execute Line 35. That will cause the loop in Lines 13-37 (Figure 4) fail and repeat. However, all newer *increment_era()* calls will only update *global_era* after *help_thread()* is complete. □

**Lemma 5.** *The loop in Lines 74-106, Figure 4 is bounded by at most MAX_THREADS iterations.*

*Proof.* The same idea as in Lemma 4. We also need to make sure that the loop will not go beyond one slow path cycle. This is achieved by comparing the tag component in Line 106. □

**Theorem 2.** `retire()` *is wait-free bounded.*

*Proof.* `retire()` periodically calls `try_retire()`. The loop in Lines 55-67, Figure 2, is bounded by the number of nodes in a batch. (MAX_THREADS×MAX_IDX+1 at most). Extra RETIRE_FREQ-1 nodes can be retired since `retire()` calls `try_retire()` with the corresponding frequency. Regardless of the status of the CAS operations (Lines 61 and 64), the loop moves on to the next node. The `traverse()` call (Line 65) is bounded due to Lemma 1. □

**Theorem 3.** `alloc_node()` *is wait-free bounded.*

*Proof.* `alloc_node()` calls `increment_era()`. The latter includes a bounded loop with calls to `help_thread()`. Finally, `help_thread()` is bounded due to Lemmas 2, 3, 5. □

**Theorem 4.** `protect()` *is wait-free bounded.*

*Proof.* The fast path includes a finite number of iterations. It may call `update_era()`, which calls `traverse()`. `traverse()` is bounded due to Lemma 1. `slow_path()` contains a loop which is bounded due to Lemma 4. `slow_path()` also calls `detach_nodes()`, which is bounded due to Lemma 2. Finally, `slow_path()` can call `traverse()`, which is bounded due to Lemma 1. □

**Theorem 5.** `clear()` *is wait-free bounded.*

*Proof.* The method has a bounded loop which can call `traverse()`, which is bounded due to Lemma 1. □

## 2 Crystalline-W's Extended Discussion and Pseudocode

Crystalline-W's high-level changes (with respect to Crystalline-L) are shown in Figure 1. Note that reservation's list and era are now tagged. Tags are used in slow-path procedures only; fast-path procedures simply use the value component. Crystalline-W defines per-thread *state* (for each corresponding reservation) used in slow-path procedures and *slow_counter* to identify if any thread needs helping. Those are somewhat similar to WFE's [1] corresponding slow-path variables. Finally, Crystalline-W defines the *parent* array to facilitate object handover, as discussed previously. Object handover is unique to Crystalline-W since it cannot simply scan the list of retired objects twice, as WFE, to avoid race conditions. Figure 1 also modifies `alloc_node()` to internally call `increment_era()` in lieu of doing FAA on the global era directly. Finally, `protect()` calls `slow_path()` if it fails to converge after MAX_TRIES.

Figure 2 shows changes to `try_retire()` and `traverse()`. These methods use list tainting, as previously discussed. Also, unlike WFE [1], we use two slow-path tag transitions (odd and even). This is needed to make

```
template <typename type> struct Tag {
  type   V;   // Value
  uint64 T;   // Tag, or Era for State::result
};

struct Reservation {
  Tag<Node*>  list;  // Init: {.V = nullptr, .T = 0}
  Tag<uint64> era;   // Init: {.V = 0, .T = 0}
};

struct State {
  Tag<void*> result; // Init: {.V = nullptr, .T = 0}
  uint64     era;    // Init: 0
  Node*      parent; // Init: nullptr
  Node**     obj;    // Init: nullptr
};

Reservation rsrv[MAX_THREADS][MAX_IDX+2];
State state[MAX_THREADS][MAX_IDX];
Node* parents[MAX_THREADS]; // Init: (all) nullptr
int slow_counter = 0;
```

```
// Help other threads before incrementing the era
Node* alloc_node(int size) {
  if (alloc_cnt++ % ALLOC_FREQ == 0) increment_era();
  Node* node = malloc(size);
  node->birth = global_era;
  node->blink = nullptr; // Retired if != nullptr
  return node;
}

// Use the fast-path-slow-path method
Node* protect(Node** obj, int index, Node* parent) {
  int tries = MAX_TRIES;
  uint64 prev_era = rsrv[TID][index].era.V;
  while (--tries != 0) {
    Node* ptr = *obj;
    uint64 curr_era = global_era;
    if (prev_era == curr_era) return ptr;
    prev_era = update_era(curr_era, index);
  }
  return slow_path(obj, index, parent);
}
```

Figure 1: Crystalline-W (API function changes).

the number of iterations finite in some loops (Lemmas 2 and 3) by collaborating with `try_retire()` which will skip odd tags. Figure 2 also shows `increment_era()`'s implementation as used by `alloc_node()`.

Crystalline-W's slow-path and helper thread routines are demonstrated in Figure 4. These routines use several utility methods shown in Figure 3. The idea is similar to that of WFE [1], with one major difference: we use the *parent* array to keep parent references to facilitate object hand-overs, as previously discussed.

Utility methods in Figure 3 are needed to facilitate the slow path. `get_birth_era()` uses a trick to retrieve the birth era irrespective of whether the node is retired. WFE [1] always keeps the birth era. However, the Hyaline and Crystalline schemes recycle the birth era field after retirement so that they still use 3 words per each memory object. Since the birth era does not survive node retirements (except REFS which stores the minimum era for the batch), that presents a challenge for Crystalline-W which needs to transfer the parent's era in `slow_path()`, and the parent object can already end up being retired. We use the following trick. When the parent node is still *not* retired, we simply retrieve the birth era from the node. Otherwise, we retrieve REFS' value of the minimum era.

```
// Redefine RNODE to encode REFS links: steal one bit
// to indicate REFS nodes (also applies to other
// functions that previously used dummy RNODE)
#define IS_RNODE(x) (x & 0x1) // Check if a REFS link
#define RNODE(x)    (x ^ 0x1) // Encode or decode REFS

// Another huge addend for the slow path
// (in addition to previously defined REFC_PROTECT)
const uint64 REFC_PROTECT_HANDOVER = 1 << 62;

// Adds a special REFS-terminal node and list tainting
void traverse(Node* next) {
  while (next != nullptr) {
    Node* curr = next;
    if (IS_RNODE(curr)) { // REFS-terminal node
      // It is always the last node, exit
      Node* refs = RNODE(curr);
      if (FAA(&refs->refc, -1) == 1) free_batch(refs);
      break;
    }
    next = SWAP(&curr->next, invptr); // Tainting
    Node* refs = curr->blink;
    if (FAA(&refs->refc, -1) == 1) free_batch(refs);
} }

// Increments the global era, replaces regular FAA
// (needs to help other threads first)
void increment_era() {
  if (slow_counter != 0) {
    for (int i = 0; i < MAX_THREADS; i++) {
      for (int j = 0; j < MAX_IDX; j++) {
        if (state[i][j].result.V == invptr)
          help_thread(i, j);
  } } }
  FAA(&global_era, 1);
}
void try_retire() { // This replacement is wait-free
  uint64 min_birth = batch.refs->birth;
  Node* last = batch.first;
  // Also check odd tags to bound slow-path loops
  for (int i = 0; i < MAX_THREADS; i++) {
    for (int j = 0; j < MAX_IDX+2; j++) {
      if (rsrv[i][j].list.V == invptr ||
          (rsrv[i][j].list.T & 0x1)) continue;
      if (rsrv[i][j].era.V < min_birth ||
          (rsrv[i][j].era.T & 0x1)) continue;
      if (last == batch.refs)
        return; // Ran out of nodes, exit
      last->slot = &rsrv[i][j];
      last = last->bnext;
  } }
  // Retire, make it wait-free by list tainting
  Node* curr = batch.first;
  int64 cnt = -REFC_PROTECT;
  for (; curr != last; curr = curr->bnext) {
    Reservation* slot = curr->slot;
    if (slot->list.V == invptr) continue;
    Node* prev = SWAP(&slot->list.V, curr);
    if (prev != nullptr) {
      if (prev == invptr) { // Inactive previously
        if (CAS(&slot->list.V, curr, invptr))
          continue; // Try to rollback
      } else { // Tainted: traverse the chopped tail
        if (!CAS(&curr->next, nullptr, prev))
          traverse(prev);
    } }
    cnt++;
  }
  if (FAA(&batch.refs->refc, cnt) == -cnt)
    free_batch(batch.refs);
  batch.first = nullptr; batch.counter = 0;
}
```

Figure 2: Crystalline-W's try_retire(), traverse(), and increment_era().

```
// Hand over the parent object if it is retired
void handover_parent(Node* parent) {
  if (parent && parent->blink != nullptr) {
    Node* refs = get_refs_node(parent);
    FAA(&refs->refc, REFC_PROTECT_HANDOVER);
    int64 cnt = -REFC_PROTECT_HANDOVER;
    for (int i = 0; i < MAX_THREADS; i++)
      if (CAS(&parents[i], parent, nullptr)) cnt++;
    FAA(&refs->refc, cnt);
} }

uint64 get_birth_era(Node* node) { // Get parent's
  if (node == nullptr) return 0;   // birth era
  uint64 birth_era = node->birth;
  Node* link = node->blink;
  // For already retired SLOT nodes, use REFS' value
  if (link != nullptr && !IS_RNODE(link))
    birth_era = link->birth;
  return birth_era;
}
// Makes the tag+1 transition and detaches an old list
void detach_nodes(int i, int j, int tag) {
  // A simple era tag transition: tag -> tag+1
  CAS(&rsrv[i][j].era.T, tag, tag+1);
  // Detach nodes and increment the list tag
  do { // Bounded by MAX_THREADS (try_retire checks
    old = rsrv[i][j].list; // the era tag)
    if (old.T != tag) break;
    bool success = WCAS(&rsrv[i][j].list,
                        old, { nullptr, tag+1 });
  } while (!success);
  return success ? old.V : invptr; // Previous value
}

// Get REFS node from any node in a batch
Node* get_refs_node(Node* node) {
  Node* refs = node->blink;
  if (IS_RNODE(refs)) refs = node; // This node itself
  return refs;
}
```

Figure 3: Crystalline-W's utility functions for the slow path.

```
void slow_path(Node** obj, int index, Node* parent) {
  // Getting parent's birth is tricky: for non-retir-
  // ed nodes use 'birth', else retrieve the minimum
  // birth from REFS, see get_birth_era() for details
  uint64 parent_birth = get_birth_era(parent);
  FAA(&slow_counter, 1);
  state[TID][index].obj = obj;
  state[TID][index].parent = parent;
  state[TID][index].era = parent_birth;
  uint64 tag = rsrv[TID][index].era.T;
  state[TID][index].result = { invptr, tag };
  uint64 prev_era = rsrv[TID][index].era.V;
  do { // Bounded by MAX_THREADS
    Node* list, * ptr = *obj;
    uint64 curr_era = global_era;
    if (curr_era == prev_era &&
        WCAS(&state[TID][index].result,
              { invptr, tag }, { nullptr, 0 })) {
      rsrv[TID][index].era.T = tag+2;
      rsrv[TID][index].list.T = tag+2;
      FAA(&slow_counter, -1);
      return ptr; // DONE
    }
    // Dereference previous nodes and update the era
    if (rsrv[TID][index].list.V != nullptr) {
      list = SWAP(&rsrv[TID][index].list.V,
                   nullptr);
      if (rsrv[TID][index].list.T != tag)
        goto produced; // Result was just produced
      if (list != invptr) traverse(list);
      curr_era = global_era;
    }
    // WCAS fails only when the result is produced
    WCAS(&rsrv[TID][index].era,
          { prev_era, tag }, { curr_era, tag });
    prev_era = curr_era;
  } while (state[TID][index].result.V == invptr);
  list = detach_nodes(TID, index, tag); //tag+1 state
produced:
  ptr = state[TID][index].result.V;
  uint64 era = state[TID][index].result.T;
  rsrv[TID][index].era.V = era;
  rsrv[TID][index].era.T = tag+2;
  rsrv[TID][index].list.T = tag+2;
  // Check if the obtained node is already retired
  if (ptr && ptr->blink != nullptr) {
    Node* refs = get_refs_node(ptr);
    FAA(&refs->refc, 1);
    if (list != invptr) traverse(list);
    list = SWAP(&rsrv[TID][index].list.V,
         RNODE(refs)); // Put a REFS-terminal node
  }
  FAA(&slow_counter, -1);
  // Traverse the previously detached list
  if (list != invptr) traverse(list);
  // Hand over the parent to all helper threads
  handover_parent(parent);
  return ptr; // DONE
}
```

```
void help_thread(int i, int j) {
  Tag<void*> result = state[i][j].result;
  if (result.V != invptr) return;
  uint64 era = state[i][j].era;
  Node* parent = state[i][j].parent;
  if (parent != nullptr) {
    rsrv[TID][MAX_IDX].list.V = nullptr;
    rsrv[TID][MAX_IDX].era.V = era;
    parents[TID] = parent; // Advertise for a handover
  }
  Node** obj = state[i][j].obj;
  uint64 tag = rsrv[i][j].era.T;
  if (tag != result.T) goto changed;
  uint64 curr_era = global_era;
  do { // Bounded by MAX_THREADS
    prev_era = update_era(curr_era, MAX_IDX+1);
    Node* ptr = *obj;
    uint64 curr_era = global_era;
    if (prev_era == curr_era) {
      if (WCAS(&state[i][j].result, // Published the
               result, { ptr, curr_era })) { // result
        Node* list = detach_nodes(i, j, tag);
        if (list != invptr) traverse(list);
        do { // Set the new era, <= 2 iterations
          old = rsrv[TID][index].era;
          if (old.T != tag+1) break;
        } while (!WCAS(&rsrv[TID][index].era,
                     old, { curr_era, tag+2}));
        // If the obtained node is already retired
        if (ptr && ptr->blink != nullptr) {
          Node* refs = get_refs_node(ptr);
          FAA(&refs->refc, 1);
          do { // Bounded by MAX_THREADS
            old = rsrv[TID][index].list;
            if (old.T != tag+1) break;
            ok = WCAS(&rsrv[TID][index].list,
                      old, { RNODE(refs), tag+2 });
            if (ok && old.V != invptr) traverse(old.V);
            if (ok) goto done;
          } while (!ok);
          FAA(&refs->refc, -1); // Already inserted
        } else { // A simple tag transition
          CAS(&rsrv[TID][index].list.V, tag+1, tag+2);
      } }
      break;
    }
  } while (state[i][j].result == result);
done:
  Node* lst = SWAP(&rsrv[TID][MAX_IDX+1].list.V,invptr)
  traverse(lst);
changed: // If handover occurs, dereference the parent
  if (parent != nullptr) {
    if (SWAP(&parents[TID], nullptr) != parent) {
      Node* refs = get_refs_node(parent);
      if (FAA(&refs->refc, -1) == 1) free_batch(refs);
    }
    Node* lst = SWAP(&rsrv[TID][MAX_IDX].list.V,invptr)
    traverse(lst);
} }
```

Figure 4: Crystalline-W's slow-path methods.

# References

[1] R. Nikolaev and B. Ravindran. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 130–143, New York, NY, USA, 2020. ACM.

[2] R. Nikolaev and B. Ravindran. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, pages 987–1002, New York, NY, USA, 2021. ACM.

[3] P. Ramalhete and A. Correia. Hazard Eras - Non-Blocking Memory Reclamation (Full Version). 2017.