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

RUSLAN NIKOLAEV, Pennsylvania State University, USA
BINOY RAVINDRAN, Virginia Tech, USA

Historically, memory management based on lock-free reference counting was very inefficient, especially for read-dominated workloads. Thus, approaches such as epoch-based reclamation (EBR), hazard pointers (HP), or a combination thereof have received significant attention. EBR exhibits excellent performance but is blocking due to potentially unbounded memory usage. In contrast, HP are non-blocking and achieve good memory efficiency but are much slower. Moreover, HP are only lock-free in the general case. Recently, several new memory reclamation approaches such as WFE and Hyaline have been proposed. WFE achieves wait-freedom, but is less memory efficient and performs suboptimally in oversubscribed scenarios; Hyaline achieves higher performance and memory efficiency, but lacks wait-freedom.

We present a family of non-blocking memory reclamation schemes, called Crystalline, that simultaneously addresses the challenges of high performance, high memory efficiency, and wait-freedom. Crystalline can guarantee complete wait-freedom even when threads are dynamically recycled, asynchronously reclaims memory in the sense that any thread can reclaim memory retired by any other thread, and ensures (an almost) balanced reclamation workload across all threads. The latter two properties result in Crystalline's high performance and memory efficiency. Simultaneously ensuring all three properties requires overcoming unique challenges. Crystalline supports ubiquitous x86-64 and ARM64 architectures, while achieving superior throughput than prior fast schemes such as EBR as the number of threads grows.

We also accentuate that many recent approaches, unlike HP, lack strict non-blocking guarantees when used with *multiple* data structures. By providing full wait-freedom, Crystalline addresses this problem as well.

CCS Concepts: • **Theory of computation** → **Concurrent algorithms**; **Shared memory algorithms**.

Additional Key Words and Phrases: memory reclamation, wait-free, hazard pointers

ACM Reference Format:
Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation. *Proc. ACM Program. Lang.* 8, PLDI, Article 235 (June 2024), 25 pages. https://doi.org/10.1145/3658851

1 INTRODUCTION

Exploiting parallelism on multi-core architectures often requires scalable non-blocking data structures as opposed to more traditional, lock-based designs. As non-blocking data structures do not use simple mutual exclusion, their memory management is challenging: a concurrent thread may hold an obsolete pointer to an object which is about to be freed by another thread. Responding to this problem, *safe memory reclamation* (SMR) schemes for unmanaged C/C++ code have been proposed [10, 14, 19, 29, 35, 37, 43, 54]. However, they typically trade off memory efficiency for high throughput, or high throughput (and memory efficiency) for stronger progress properties.

Authors' addresses: Ruslan Nikolaev, Pennsylvania State University, University Park, USA, rnikola@psu.edu; Binoy Ravindran, Virginia Tech, Blacksburg, USA, binoy@vt.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART235
https://doi.org/10.1145/3658851

Exacerbating the problem, only a few schemes [10, 14, 29, 35, 37, 43] are *truly*[1] non-blocking and bound memory usage, i.e., a suspended thread has no adverse impact on progress in other threads. Moreover, very few schemes [29, 35] are non-blocking when using with multiple data structures (Section 2) and only one scheme [35] is wait-free.

SMR schemes with high performance *and* memory efficiency[2] are desirable. The significance of SMR workload balancing – the task of reclaiming deleted memory objects – across all threads have not received adequate attention in the literature. Consider the common scenario of read-dominated workloads, i.e., majority of the operations are reads, but data can still be modified. If the reclamation workload is unbalanced, as in most existing SMR schemes [19, 29, 43, 54], most threads are not actively reclaiming memory, which can cause memory waste. As a side-effect, throughput can also decrease due to the consequent increased pressure on the memory management system. In oversubscribed scenarios where there are more threads than cores available, this can have a cascading effect: threads that reclaim memory are preempted, further degrading performance.

The vast majority of SMR schemes (e.g., [29, 43, 54]) are inherently *synchronous*, i.e., they periodically examine which objects marked for deletion can be safely reclaimed. In these schemes, only threads that modify data can reclaim memory.[3] Thus reclamation becomes unbalanced, especially if most threads are only reading data, which degrades memory efficiency as well as performance. In contrast, reference counting [16, 20, 30, 53] is *asynchronous*: a thread with the last reference to an object frees it. Since such reclaiming threads are more or less arbitrary, the reclamation workload is generally balanced. Unfortunately, reference counting has high overheads when accessing objects. Hyaline [34, 37] solves this problem by using reference counters only when objects are deleted, and thereby achieves high performance. However, Hyaline can be blocking since the memory usage is unbounded when threads starve.

Motivated by these concerns, we present a family of SMR schemes, Crystalline, that achieves high performance, high memory efficiency, and lock-/wait-freedom. Crystalline is inspired by Hyaline, is asynchronous, and balances the reclamation workload. Whereas Hyaline is very conservative in terms of when reclamation begins, Crystalline's key insight is *optimism*, i.e., it attempts to reclaim deleted objects as soon as possible, while backtracking without mutating the global state if it is impossible to do so. Crystalline's design is complex due to the inherent challenge in achieving all three properties, and we tame this complexity by an incremental design approach: (1) Crystalline-L, a lock-free scheme which bounds memory usage even when threads starve; (2) Crystalline-LW, a wait-free scheme for wait-free data structures that use automatic restarting (e.g., Timnat-Petrank's formulation [51]); and (3) Crystalline-W, a wait-free scheme for all data structures.

Crystalline's flexibility is also important as the wait-free property of SMR is not necessarily related to the wait-free property of the underlying data structures. For example, while Crystalline-W does not need Timnat-Petrank's formulation to bound reclamation operations, some data structures such as linked lists will still typically need this formulation regardless of the SMR scheme used to bound the number of *unavoidable* restarts and guarantee wait-freedom (see Section 7 for more details). Thus, a more lightweight version, Crystalline-LW, can be used for such data structures. This helps to tailor Crystalline to different needs without introducing unnecessary complexity.

The three Crystalline schemes bound memory usage and provide roughly similar performance (Section 6), while even outperforming in some tests prior fast schemes such as Hyaline and EBR. The schemes differ in terms of their complexity and hardware requirements. All schemes achieve outstanding performance as well as excellent memory efficiency over a broad range of workloads.

---

[1] Technically speaking, this is just *non-blocking* but due to discrepancy in the literature, we clarify strict progress properties.
[2] A lower theoretical bound does not automatically imply better practical efficiency, see Section 6.
[3] Although deferring is possible for EBR [2], it is unclear how to bound memory usage, let alone achieve wait-freedom.

Table 1. Crystalline vs. existing SMR. *Restart* specifies progress depending on whether restarting is allowed, *1 DS*: one data structure progress, *2+ DS*: multiple DS progress. *BLK*: blocking, *LF*: lock-free, *WF*: wait-free.

Scheme	Balanced	Fast	[With Restart] 1 DS	[With Restart] 2+ DS	[W/o Restart] 1 DS	[W/o Restart] 2+ DS	S-Free	Header
EBR	✘	✔	BLK	BLK	BLK	BLK	✔	1 word[4]
IBR	✘	✓	WF	BLK	BLK	BLK	✘	3 words
HP	✘	✘	WF	**WF**	LF	**LF**	✘	1 word[4]
HE	✘	✓	WF	BLK	LF	BLK	✘	3 words
WFE	✘	✓	WF	**WF**	WF	**WF**	✘	3 words
Hyaline-1	✔	✔	BLK	BLK	BLK	BLK	✔	3 words
Hyaline-1S	✔	✔	LF	BLK	BLK	BLK	✔	3 words
Crystalline-L	✔	✔	LF	BLK	LF	BLK	✔	3 words
Crystalline-LW	✔	✔	WF	BLK	LF	BLK	✔	3 words
Crystalline-W	✔	✔	WF	**WF**	WF	**WF**	✔	3 words

2 BACKGROUND

Progress. In the classical *non-blocking* categorization [25], an algorithm is *lock-free* if one or more threads complete an operation in a finite number of steps, and *wait-free* if *all* threads complete their operations in a finite number of steps. Wait-freedom is harder to achieve, but is quite desirable, especially for applications with tail latency constraints [15]. Non-blocking progress should also be considered from a memory usage perspective, which must be bounded [29, 43, 44]. Otherwise, when memory is exhausted, and no thread makes progress, any algorithm becomes *blocking*.

Challenges. One can argue that bounding memory usage is the whole point of SMR, as otherwise we could simply leak memory. EBR [19] is an easy-to-use SMR scheme, but is blocking due to potentially unbounded memory usage. In contrast, hazard pointers (HP) [29] and hazard eras (HE) [44] are non-blocking for *one data structure* (see below for multiple data structures). In addition, they are mostly wait-free except one method that safely retrieves pointers in an unbounded loop. If a data structure is specifically designed to restart unsuccessful operations [45], then reclamation also becomes fully wait-free. Otherwise, these SMR schemes are only lock-free. A similar distinction exists with other SMR schemes, such as interval-based reclamation (IBR) [54] and Hyaline-1S [37]: they can be either non-blocking or blocking depending on whether restarting is allowed.

There are even more challenges when software uses *multiple data structures*. SMR must typically be shared for more predictable performance and memory usage irrespective of which data structures are used more frequently. However, progress properties become more subtle. For example, HE, IBR, and Hyaline-1S are blocking in this case because they use a global era clock, which may *never* converge for *any* thread that operates on one data structure even though some threads succeed in other data structures. Restarting for just one data structure in this case is useless, while restarting across multiple data structures is practically infeasible. HP, on the other hand, need to converge on a pointer local to each data structure and thus is still non-blocking in this case. Due to locality, restarting still makes sense in HP. Finally, WFE [35], a wait-free extension to HE, is also non-blocking as threads across *all* data structures cooperate to guarantee wait-freedom.

[4]HP and EBR reserve 1 word for a local limbo list pointer. This overhead can be eliminated altogether by allocating an intermediate container object when retiring, but that causes undesirable circular allocator dependency (avoided in Section 6). In the same vein, other schemes can reduce the overhead, e.g., IBR, HE, Hyaline-1S, and Crystalline-L's overhead is only 1 word with container objects. In practice, overheads are often irrelevant due to objects often being cache-line sized to avoid false sharing, causing inevitable memory waste. (Though some exceptions also do exist.)

```
1  struct StkNode : Node {
2    StkNode* next;
3    Object*  obj;
4  };
5  StkNode* stack_top = nullptr;
6
7  void push(Object* obj) {
8    StkNode* node = new StkNode;
9    node->obj = obj;
10   do // Replace the stack top
11     node->next = stack_top;
12   while (!CAS(&stack_top,
13          node->next, node));
14 }
```

```
15 Object* pop() {
16   Object* obj = nullptr;
17   while (true) {
18     // Fetch the stack top
19     StkNode* node = stack_top;
20     if (!node) break;
21     if (CAS(&stack_top, node,
22             node->next)) {
23       obj = node->obj;
24       delete node;
25       break;
26   } }
27   return obj;
28 }
```

Fig. 1. Lock-free stack (without proper reclamation).

Snapshot Freedom. Many SMRs (HP, IBR, HE, WFE) take a *snapshot* of the entire state (all hazard pointers or hazard eras) to avoid repeated access of that state when checking if not-yet-freed blocks can be safely deleted. These schemes typically examine fundamentally the *same* state (but not necessarily *exactly* the same state) many times per each *iteration*, which, due to contention, results in expensive cache misses without taking local snapshots.[5] The need to take snapshots affects *usability* of SMR when new threads are created dynamically at runtime as the size of snapshots depends on the number of threads, n, which is unknown beforehand. Snapshots also incur a non-negligible $O(n^2)$ *memory overhead*. While EBR needs no snapshots, IBR, HE, and WFE trade this quality for lock- or wait-freedom. Hyaline-1S is still snapshot-free, making it a candidate for extensions.

Comparison. Table 1 compares Crystalline with the existing SMR schemes. Reclamation of (asynchronous) Crystalline schemes is generally balanced across threads. As described before, balanced reclamation is especially important for read-dominated workloads, wherein most threads are not actively modifying data structures, and enables faster memory reclamation. Here, we implicitly assume that all threads can be treated equally, which is also typical for wait-free algorithms that use the *helping* technique [25].

Crystalline-L/-LW/-W reserve 3 words in each memory object for a header (similar to HE, WFE, IBR). EBR and HP need 1 word, but EBR is blocking while HP is slow. Only Crystalline-W, HP, and WFE provide non-blocking progress when using multiple data structures. Crystalline-W is faster than HP and WFE, as we show in Section 6.

Atomics. CAS (compare-and-swap) is used universally by most lock- and wait-free algorithms. FAA (fetch-and-add) and SWAP are two specialized instructions that are often available in hardware (e.g., x86-64 and ARM64) and are used by some algorithms [18, 32, 33, 35, 38, 55]. The execution time of FAA and SWAP is typically bounded, allowing them to be used directly in wait-free algorithms. WCAS (wide CAS)[6] updates two *contiguous* words and is available on x86-64 and ARM64. Typically, it extends a pointer with a monotonically increasing *tag* to prevent the ABA problem [25].

Note that Crystalline-L needs no specialized instructions for lock-freedom. Crystalline-LW needs hardware FAA and SWAP for wait-freedom, which is similar to IBR and HE that require hardware FAA for that purpose. Finally, Crystalline-W also needs WCAS to identify slow-path cycles and prevent spurious updates, similarly to WFE [35], the only existent fully wait-free scheme. Compared

[5] S-Free schemes can still examine global state several times but only in different iterations that are sufficiently separated in time, i.e., taking snapshots is not helpful due to the global state divergence. Put differently, this simply means that an efficient implementation is feasible without taking snapshots.

[6] WCAS is not to be confused with double-word CAS, which updates two *distinct* words and is not widely available.

to prior schemes, Crystalline-LW/-W require SWAP in addition to FAA for wait-freedom. This is not burdensome as all current architectures that implement WCAS and FAA also support SWAP. This trend will likely continue for the foreseeable future.

Reclamation Model. For all discussed schemes, the API is called explicitly. Each thread retains a *reservation*, which is a globally-accessible per-thread state. Memory blocks, which incorporate all necessary SMR headers, are allocated via standard operating system (OS) means. After memory blocks appear in a data structure, they can only be reclaimed in two steps. First, after deleting a pointer from the data structure, a memory block is *retired*. When the block finally becomes unreachable by any other concurrent thread, it is returned to the OS.

Usage Example. We discuss SMR using a lock-free stack [52], shown in Figure 1. The stack inserts or deletes arbitrary objects which are stored in a list consisting of `StkNode` elements. Each element incorporates SMR's `Node`, an opaque structure that is attached to every reclamation unit.

2.1 Hyaline Reclamation Schemes

Hyaline's [37] high-level idea is to keep track of all active threads and record the number of active threads for the corresponding memory object when it is retired. As each thread completes its operation, it has to go through the list of previously retired objects (accumulated while it was active) and decrement the corresponding thread reference counters. The last thread to decrement the reference counter will also deallocate corresponding memory objects. Hyaline is similar to EBR and HP in its concept of "retiring" objects by putting them into a limbo list, but Hyaline's lists are global and two-dimensional (more on this below), whereas EBR's and HP's lists are local (per-thread) and one-dimensional. Unlike classical reference counting, this expensive operation (a reference counter update) only takes place in Hyaline during retirement, i.e., a simple read on a data structure does not incur a significant cost.

Four schemes are presented in [37]. Hyaline and Hyaline-1 are blocking just like EBR. Two "robust" schemes, Hyaline-S/-1S extend former schemes to detect stalled threads with the help of special progress stamps known as *eras*, as discussed below. These schemes are lock-free *only* if data structures are modified to restart excessively long operations to prevent unbounded memory usage [37] (e.g., iterating an excessively long and ever-changing list may easily result in unbounded memory reservation until a thread operation is restarted). However, restarting is not always feasible, a problem that we address in this paper.

The difference between the Hyaline schemes lies in whether threads can share reservations (Hyaline, Hyaline-S) or need their own per-thread (Hyaline-1, Hyaline-1S) reservations. Hyaline-1/-1S are closer to typical schemes (e.g., EBR, IBR, HP, and HE) in the way reservations are managed. In practice, sharing the same reservation is not that useful, especially for wait-free data structures. Typical wait-free algorithms that rely on fast-path-slow-path methodologies already maintain per-thread states, and thus maintaining per-thread reservations does not incur any additional burden. In this paper, we focus only on Hyaline-1/-1S to increase the tractability of the problem of lock- and wait-free reclamation.

Hyaline-1. In Hyaline-1, threads accumulate retired memory blocks, *nodes*, in per-thread lists called *batches*. When a batch exceeds a certain size, it is attached to a two-dimensional grid of linked lists, in which rows represent active threads, and columns represent batches. More specifically, a batch is inserted into the rows of *all* active threads. The batch will not be reclaimed until *all* active threads have "commented on" the batch, stating that they will no longer access any objects in it. This "commenting" is done whenever a data structure operation terminates. Eventually, the batch can be freed once every active thread finishes its operation and comments on the batch. This way,

```
1  struct Node { // Each node takes 3 memory words
2      union {
3          uint64       refc;  // REFS: Refcounter
4          Node*        bnext; // SLOT: Next node
5      };
6      union { // SLOT nodes reuse space after retire
7          uint64       birth; // Node's birth era
8          Reservation* slot;  // Used in Crystalline
9          Node*        next;  // SLOT: After retire
10     };
11     Node*  blink;          // REFS: First SLOT node,
12 };                         // SLOT: Pointer to REFS
13 struct Batch { // Accumulates nodes before retiring
14     Node*  first, refs;    // Init: nullptr, nullptr
15     int    counter;        // Init: 0
16 };
17 thread_local Batch batch; // Per-thread batches
```

Fig. 2. Hyaline's data structures.

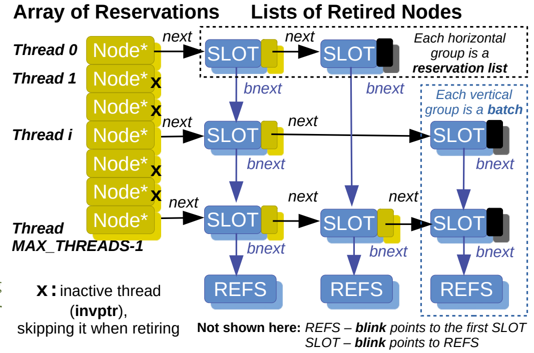


Fig. 3. An example with retired batches.

nodes in a batch are only freed when every thread that could *potentially* be holding a hazardous reference to a node in the batch has relinquished its potential access.

Batches are linked lists, and we think of them as being laid out from top to bottom in a two-dimensional plane. When we say a batch is attached to several rows of the grid, more specifically we mean that *individual* nodes of the batch are added to the linked lists that represent rows of the grid, in a round robin fashion (rather than, say, attaching the head of the batch to each row). Thus, batches (columns) and thread reservation lists (rows) are each linked lists, but the nodes of these lists are woven together to form a grid.

The purpose of batches is twofold: (i) eliminate contention due to frequent `retire()` calls by retiring multiple nodes together, and (ii) reserve space in every node for a separate per-thread linked list pointer (i.e., per each row) when retiring the batch. Due to (ii), the number of nodes in each batch should at least equal to the number of threads, MAX_THREADS. Furthermore, an extra batch node is needed to keep a reference counter, which reflects the number of active threads when the batch is retired. Thus, batches should at least contain MAX_THREADS+1 nodes.

When an active thread completes its operation, it traverses its list (its row) and decrement the corresponding reference counter of every batch. Once a batch's reference counter is zero, the entire batch is reclaimed.

Hyaline-1's [37] API is very similar to that of classical EBR [19] and includes:

- `activate()`, `clear()`: methods used by each thread to enclose a data structure operation; local pointers to shared memory are only valid within each such enclosure. In Figure 1, `activate()` must be inserted before **L8, L16** and `clear()` after **L13, L26**.
- `retire(blk)`: a method which indicates that block `blk` is deleted and will not be accessed subsequently. Already running threads can still access `blk`; when `blk` can be safely freed, it will be deallocated. In Figure 1, `retire()` must be used in lieu of **L24**.

Figure 2 shows a header (`Node`), which must be attached to every allocated data structure object. *Union* definitions are simply for header compaction. For convenience, we categorize retired nodes into two types: REFS and SLOT. No such distinction exists when nodes are initially allocated, but as nodes are retired and attached to a thread-local batch, we start differentiating them. The very first retired node in the batch is denoted as REFS, and its purpose is to keep batch's reference counter. All subsequent nodes are denoted as SLOT, and each of them keeps one list pointer for one row. SLOT nodes are linked together, and each of them has a reference to the REFS node. The REFS

```
1  const uint64 REFC_PROTECT = 1 << 63; // Prevents reclam.
2  struct Reservation {
3    Node* list; // Init: invptr
4  };
5  Reservation rsrv[MAX_THREADS];
6
7  void activate() { // An empty list
8    rsrv[TID].list = nullptr;
9  }
10
11 void clear() {
12   Node* p = SWAP(&rsrv[TID].list, invptr); // p!=invptr
13   traverse(p); // since activate() was called
14 }
15
16 void free_batch(Node* refs) { // RNODE is used later for
17   // Cryst-W, dummy (n=refs->blink) for Hyaline/Cryst-L
18   Node* n = RNODE(refs->blink);
19   do {
20     Node* obj = n;
21     // refc & bnext overlap and are 0
22     // (nullptr) for the last REFS node
23     n = n->bnext;
24     free(obj);
25 } } while (n != nullptr);
26
27 void traverse(Node* next) {
28   while (next != nullptr) {
29     Node* curr = next;
30     next = curr->next;
31     Node* refs = curr->blink;
32     if (FAA(&refs->refc, -1) == 1) free_batch(refs);
33 } }
34 void retire(Node* node) {
35   if (!batch.first) { // the REFS node
36     batch.refs = node;
37     node->refc = REFC_PROTECT;
38   } else { // SLOT: 'blink' points to REFS
39     node->blink = batch.refs;
40     node->bnext = batch.first;
41   }
42   batch.first = node;
43   // Need MAX_THREADS+1 nodes to insert
44   // to MAX_THREADS lists, otherwise exit
45   if (batch.counter++ < MAX_THREADS) return;
46   // Now retire, finalize the batch: 'blink'
47   // of the REFS node points to the 1st SLOT
48   batch.refs->blink = batch.first
49   Node* curr = batch.first;
50   int64 cnt = -REFC_PROTECT;
51   for (int i = 0; i < MAX_THREADS; i++) {
52     while (true) {
53       Node* prev = rsrv[i].list;
54       if (prev == invptr) break;
55       curr->next = prev;
56       if (CAS(&rsrv[i].list, prev, curr)) {
57         cnt++;
58         break;
59     } }
60     curr = curr->bnext;
61   }
62   // Finish retiring: change refc
63   if (FAA(&batch.refs->refc, cnt) == -cnt)
64     free_batch(batch.refs);
65   batch.first = nullptr; batch.counter = 0;
66 }
```

Fig. 4. The Hyaline-1 reclamation scheme.

```
1  struct Reservation { // Add the 'era' field
2    Node* list; // Init: invptr
3    uint64 era; // Init: 0
4  };
5
6  void retire(Node* node) {
7    if (!batch.first) { // the REFS node
8      ...
9    } else { // SLOT nodes
10     // Reuse REFS' birth era to keep the
11     // minimum birth era in the batch
12     if (batch.refs->birth > node->birth)
13       batch.refs->birth = node->birth;
14     ...
15   }
16   uint64 min_birth = batch.refs->birth;
17   ...
18   // After Line 54, Figure 4, add:
19   if (rsrv[i].era < min_birth) break;
20 }
21 uint64 global_era = 1;
22 thread_local int alloc_cnt = 0;
23
24 // Replace malloc() with alloc_node()
25 Node* alloc_node(int size) {
26   if (!(alloc_cnt++ % ALLOC_FREQ)) FAA(&global_era, 1);
27   Node* node = malloc(size);
28   node->birth = global_era;
29   return node;
30 }
31
32 Node* protect(Node** obj) {
33   uint64 prev = rsrv[TID].era;
34   while (true) {
35     Node* node = *obj;
36     uint64 era = global_era;
37     if (prev == era) return node;
38     rsrv[TID].era = era;
39     prev = era;
40 } }
```

Fig. 5. The Hyaline-1S reclamation scheme (showing only changes with respect to Hyaline-1).

node has a reference to the beginning of the list of SLOT nodes. Figure 3 shows the relation of retired batches to reservations.

Figure 4 presents Hyaline-1. We use TID to denote the current thread ID. Each reservation contains its *list* of retired nodes. The original Hyaline-1's paper [37] used a binary reference counter (0 or 1) associated with each reservation's list pointer, which would indicate whether the corresponding thread is active. Figure 4 indicates a "cleared" reservation by using a special `invptr` value for reservation's list pointer, which is similar but without "stealing" one bit. We use the `(void *) -1` value, which is reserved by OSes, e.g., in the `mmap(2)` system call, for errors and never appears in data structures. Any legitimate pointer (including `nullptr`) indicates an "active" reservation. REFC_PROTECT, a constant, in L37 prevents de-allocation until L63 (adjusted in L50).

Hyaline-1S. Since a batch is not reclaimed until *all* active threads decrement its reference count, a crashed or slow thread T can prevent a batch from being freed. And, if all threads continue to attach their batches to T's row, then T can prevent all memory from being reclaimed forever. Hyaline-1S solved this problem by using a hazard era-like approach [43]: T records the last era in which it was active, and each object records the *birth* era in which it was allocated. This way, threads can avoid attaching a batch to T's row if all objects in the batch were allocated after T was last active (since T cannot have access to those objects). Such batches can be reclaimed without waiting for the stalled thread to decrement their reference counts.

Hyaline-1S [37] extends Hyaline-1's API with:

- `protect(ptr):` a method which is used to safely retrieve pointers that are about to be dereferenced by creating a local copy. In Figure 1, we need to wrap `stack_top` with this method in **L19** when assigning it to `node`.
- `alloc_node():` a method which wraps `malloc()` to initialize an object with its *birth* era. In Figure 1, it must be called in lieu of **L8**.

All *valid* pointers (i.e., not marked anyhow for deletion) retrieved between `activate()` and `clear()` can be safely accessed. This semantics, common in other lock-free schemes such as hazard pointers or IBR, is different from that of Hyaline-1 and EBR. Care must be taken when accessing pointers, specifically when traversing logically "deleted" nodes as in Harris' linked-list [21], which needs to be modified to promptly unlink nodes as discussed in [29].

Figure 5 shows Hyaline-1S's changes to Hyaline-1's corresponding methods. Each reservation additionally adds a 64-bit active *era* that was last observed by the respective thread. The eras are assumed to never overflow in practice. When nodes are allocated, `alloc_node()` initializes their birth eras with the global era clock value. When retrieving pointers, threads call `protect()` to update reservation's era value.

`retire()` calculates batch's minimum birth era, which is used subsequently when the batch is retired. Unlike in IBR or HE, the *birth* era field need not survive the `retire()` call. REFS reuses this field to keep the minimum birth era, while SLOT reuses the space to keep a reservation list pointer.

3 LOCK-FREE RECLAMATION

Hyaline-1S's solution is somewhat inefficient: T reserves all eras from the beginning of the execution until the point when it crashes or becomes slow, irrespective of whether it will access any of the objects in those eras. In reality, it might only access objects in one or two eras. Worse yet, this limitation will not guarantee bounded memory usage for an excessively long operation unless the operation is periodically restarted, which is not always feasible. The problem arises when a thread reserves an increasing number of local pointers in an unbounded loop (e.g., one "unlucky" thread is stuck traversing a list because it keeps growing). Though certain data structures can be modified to restart operations [26, 37, 54], other schemes such as HP and HE are lock-free without restarting.

Comparing HP/HE with IBR/Hyaline-1S, observe that the former two bound memory usage due to slight API differences, which result in finer granularity of reservations. This leads us to the

```
1  Reservation rsrv[MAX_THREADS][MAX_IDX];
2
3  void clear() {
4    for (int i = 0; i < MAX_IDX; i++) {
5      Node* p = SWAP(&rsrv[TID][i].list, invptr);
6      if (p != invptr) traverse(p);
7  } }
8
9  void try_retire() { // Attempt to retire
10   uint64 min_birth = batch.refs->birth;
11   Node* last = batch.first;
12   // Check if the number of nodes suffices
13   for (int i = 0; i < MAX_THREADS; i++) {
14     for (int j = 0; j < MAX_IDX; j++) {
15       if (rsrv[i][j].list == invptr) continue;
16       if (rsrv[i][j].era < min_birth) continue;
17       if (last == batch.refs)
18         return; // Ran out of nodes, exit
19       last->slot = &rsrv[i][j];
20       last = last->bnext;
21   } }
22   // Retire if successful
23   Node* curr = batch.first;
24   int64 cnt = -REFC_PROTECT;
25   for (; curr != last; curr = curr->bnext) {
26     Reservation* slot = curr->slot;
27     while (true) { // Do not check 'era' again,
28       Node* prev = slot->list; // it can
29       if (prev == invptr) break; // only grow
30       curr->next = prev;
31       if (CAS(&slot->list, prev, curr)) {
32         cnt++;
33         break;
34   } } }
35   if (FAA(&batch.refs->refc, cnt) == -cnt)
36     free_batch(batch.refs);
37   batch.first = nullptr; batch.counter = 0;
38 }
39 Node* protect(Node** obj, int index) {
40   uint64 prev_era = rsrv[TID][index].era;
41   while (true) {
42     Node* ptr = *obj;
43     uint64 curr_era = global_era;
44     if (prev_era == curr_era) return ptr;
45     prev_era = update_era(curr_era, index);
46 } }
47
48 // Clean up the old list and set a new era
49 uint64 update_era(uint64 curr_era, int index) {
50   if (rsrv[TID][index].list != nullptr) {
51     Node* list = SWAP(&rsrv[TID][index].list, nullptr)
52     if (list != invptr) traverse(list);
53     curr_era = global_era;
54   }
55   rsrv[TID][index].era = curr_era;
56   return curr_era;
57 }
58
59 void retire(Node* node) {
60   if (!batch.first) { // the REFS node
61     batch.refs = node;
62     node->refc = REFC_PROTECT;
63   } else { // SLOT nodes
64     // Reuse the birth era of REFS to retain
65     // the minimum birth era in the batch
66     if (batch.refs->birth > node->birth)
67       batch.refs->birth = node->birth;
68     node->blink = batch.refs; // points to REFS
69     node->bnext = batch.first;
70   }
71   batch.first = node;
72   if (batch.counter++ % RETIRE_FREQ == 0) {
73     // blink of REFS points to the 1st SLOT node
74     batch.refs->blink = RNODE(batch.first);
75     try_retire();
76 } }
```

Fig. 6. Crystalline-L (showing changes with respect to Hyaline-1S), SWAP can be replaced with a CAS loop.

question: would it be possible to adopt HP/HE's API for Hyaline-1S? Surprisingly, not only we can do it, such an algorithm is *faster* and more memory *efficient* than Hyaline, as shown in Section 6.

In Crystalline-L, instead of one row per thread in the grid, there are k rows per thread, where k is the maximum number of eras a data structure operation needs to access at a time (similar to the number of hazard eras or hazard pointers a thread would need to reserve). For example, in a simple linked list, only two nodes might be accessed at a time, so two eras would be reserved at a time, necessitating two rows per thread. This way, a batch whose objects were not accessible in any of the eras reserved by a thread will not be prevented from being reclaimed by that thread.

We redefine the API so that each thread bounds the number of retrieved local pointers. This API is similar to that of HP and has the following subtle differences with Hyaline-1S:

- `protect()`: additionally passes an *index* that is assigned to a specific local pointer. `protect()` no longer has a cumulative effect, each time it resets any previous reservation associated with the specified index.
- `activate()`: **removed** from the API completely, as `protect()` is now non-cumulative.
- `clear()`: resets *all* (rather than just one) local pointer reservations made by `protect()`.

Note that Figure 1 defines only one local pointer in **L19**, for which index 0 is used. With this API, each thread holds at most MAX_IDX local pointers (indices are 0..MAX_IDX-1). For each local pointer, Crystalline-L maintains a separate list and era. Due to the stricter API, even starving threads do not reserve unbounded memory, making the scheme lock-free. Figure 6 shows Crystalline-L's changes to Hyaline-1S. When retiring, a batch must be attached to *all* reservation indices. Therefore, a batch needs to accumulate MAX_THREADS×MAX_IDX+1 rather than MAX_THREADS+1 nodes before it can be retired.

One problem with Hyaline-1S is that a batch must aggregate *all* MAX_THREADS+1 nodes before anything can retired. If this number is too high (aggravated in Crystalline-L), the algorithm becomes suboptimal due to delayed retirement. In practice, the required number of nodes is much lower as each node is appended to the respective list only if the list's era overlaps with batch's minimum birth era. In Crystalline-L, retirement can be tried sooner, irrespective of the number of threads. It implements *dynamic* batches to avoid their excessive growth. On average, batch sizes roughly equal the number of cores as eras for preempted threads are going to be behind. Other schemes (HP/IBR/HE/WFE) amortize their scanning frequency on a similar scale for good performance.

`retire()` periodically calls `try_retire()`, which checks how many reservation lists[7] are to be changed for the batch to be retired and records the location of each such reservation slot. If the number of nodes in the batch suffices, `try_retire()` completes the retirement by appending the nodes to their corresponding slot lists. `try_retire()` may look similar to the method in EBR, IBR, HE, or HP which periodically peruses thread-local lists to check if any of the previously retired nodes are safe to delete. However, its purpose in Crystalline-L is entirely different: it is used for *retirement* rather than *deallocation*. Furthermore, `try_retire()` has the above-mentioned upper bound on nodes and is *guaranteed* to eventually succeed after a *finite* number of retries.

4 WAIT-FREE RECLAMATION

Challenges. Crystalline-L is not wait-free for two key reasons. First, threads try to attach a batch to rows of the grid using an unbounded CAS loop in `try_retire()` (Lines 25-34), and a thread can starve if other threads repeatedly attach their own batches, ahead of this thread. Second, in `protect()`, there is an unbounded loop in which a thread needs to read and validate a pointer successfully, and the thread can be starved if the era continually changes, preventing it from validating successfully, a problem similar to that of HE/HP.

Assumptions. Similar to [43], we assume a 64-bit CPU that supports wait-free FAA to manipulate 64-bit eras. The CPU must also support wait-free SWAP as discussed in Section 2. These requirements are fully met by commonplace x86-64 and ARM64 architectures.

4.1 Crystalline-LW

Crystalline-LW solves the first problem. In some data structures, this is already sufficient to obtain wait-free progress. Specifically, for the second problem, it might be possible to argue that `protect()` succeeds after a bounded number of attempts. As an example, one could imagine a thread fails to validate a pointer because the era has changed, but there is some wait-free helping mechanism in the data structure that guarantees other operations will only change the era a bounded number of times before helping this thread to complete its operation. In such data structures, the elimination of the CAS loop used to attach batches to the grid suffices to ensure wait-free progress.

`try_retire()` in Figure 6, requires a CAS loop (Lines 25-34), which contends with unconditional SWAP in `update_era()` (Line 51) or concurrent `try_retire()`. Crystalline-LW replaces the CAS

[7] We call lists of retired objects "reservation lists" since there are multiple lists in Crystalline-L/-W, one list per a separate reservation.

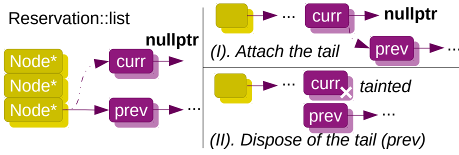


Fig. 7. Crystalline-LW: list tainting.

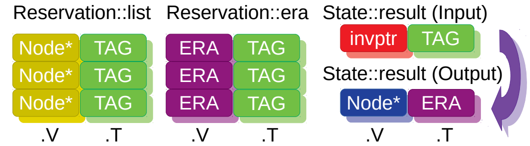


Fig. 8. Crystalline-W: reservations and state.

loop with an unconditional SWAP instruction. Attaching a batch to the grid is a two step process. First, an object *O* in the batch is SWAPped into a row of the grid. This may unlink a list of previously attached batches. If so, the thread attempts to reattach that list of batches to the row, using a single CAS (not a CAS loop) to change the next pointer of the newly attached O. If the CAS succeeds, the list of batches was reattached successfully, and the integrity of the row is restored. However, if it fails, the thread learns something important from the failure. The only way the next pointer of O can be changed is if the thread corresponding to this row finished its current data structure operation, and traversed this row, decrementing the reference counts of all batches in the list, and removing the batches from the list. In this case, rather than reattaching the batches that were unlinked by the SWAP, the correct action is to simply decrement the reference counts of these batches. In essence, the thread that SWAPped out these batches *helps* the thread that corresponds to this row complete its traversal.

We demonstrate the required changes in Figure 7. Prior to SWAPping, we initialize the *next* field of a retired node with `nullptr`. If the retired node still has its next field intact, it simply attaches the previous list as its tail using CAS (Figure 7, part I). It is also possible that a thread associated with the reservation already called `traverse()` for the just retired node. Crystalline-LW's `traverse()` additionally taints the retrieved *next* pointer of all traversed nodes by using SWAP. (Note that a node is *merely* traversed and dereferenced by the thread; only when the batch's reference counter reaches 0 is when the node is deallocated.) `try_retire()` still holds the batch with the retired node, and it finds that the next field is tainted (Figure 7, part II). Thus, `try_retire()` traverses the tail on behalf of the thread that just finished `traverse()`.

Finally, there is a corner case with a cleared reservation (`invptr`). We handle this case in Lines 30-32 of Figure 9 by rolling back to the original state (unless Line 51 of Figure 6 executes concurrently). Figure 9 shows all corresponding changes in `traverse()` and `try_retire()`.

4.2 Crystalline-W

Crystalline-W addresses the second problem irrespective of the underlying data structure mechanisms: `protect()` has an unbounded loop which converges on the era value (Lines 41-46, Figure 6), but `alloc_node()` unconditionally increments the era clock to bound memory usage. Crystalline-W is inspired by the fast-path-slow-path idea and slow-path tags from WFE, but several additional challenges are overcome for asynchronous (Hyaline-like) reclamation to make it wait-free. Unlike WFE, Crystalline-W *must* complete retirement even in the presence of slow-path conflicts. Also, Crystalline-W adopts an HP-like idea to manipulate parent objects in the slow path.

Assumptions. Similar to [35], we need WCAS, which is available in x86-64 and ARM64 architectures. Like WFE [35], Crystalline-W slightly alters `protect()`'s API. We pass an additional parameter, *parent*, which refers to a parent object where the hazardous reference is located (or `nullptr` for topmost locations).

```
1  void traverse(Node* next) {
2    while (next != nullptr) {
3      Node* curr = next;
4      next = SWAP(&curr->next, invptr); // Tainting
5      Node* refs = curr->blink;
6      if (FAA(&refs->refc, -1) == 1) free_batch(refs);
7  } }
8
9  void try_retire() { // A wait-free replacement
10   uint64 min_birth = batch.refs->birth;
11   Node* last = batch.first;
12   // Check if the number of nodes suffices
13   for (int i = 0; i < MAX_THREADS; i++) {
14     for (int j = 0; j < MAX_IDX; j++) {
15       if (rsrv[i][j].list == invptr) continue;
16       if (rsrv[i][j].era < min_birth) continue;
17       if (last == batch.refs)
18         return; // Ran out of nodes, exit
19       last->slot = &rsrv[i][j];
20       last = last->bnext;
21   } }
22   Node* curr = batch.first; // Make the loop wait-
23   int64 cnt = -REFC_PROTECT; // free by tainting
24   for (; curr != last; curr = curr->bnext) {
25     Reservation* slot = curr->slot;
26     if (slot->list == invptr) continue;
27     curr->next = nullptr;
28     Node* prev = SWAP(&slot->list, curr);
29     if (prev != nullptr) {
30       if (prev == invptr) { // Inactive previously
31         if (CAS(&slot->list, curr, invptr))
32           continue; // Try to rollback
33       } else { // Tainted: traverse a chopped tail
34         if (!CAS(&curr->next, nullptr, prev))
35           traverse(prev);
36     } }
37     cnt++;
38   }
39   if (FAA(&batch.refs->refc, cnt) == -cnt)
40     free_batch(batch.refs);
41   batch.first = nullptr; batch.counter = 0;
42 }
```

Fig. 9. Crystalline-LW changes.

Slow Path. We use a fast-path-slow-path idea, wherein `alloc_node()`, instead of unconditionally incrementing the global era, runs a helper method `increment_era()`. The latter calls `help_thread()` for every thread that needs helping, and only then it increments the global era.

For a few iterations, `protect()` attempts to converge on the era clock. After that, a slow path is taken by calling `slow_path()`. Then, `help_thread()` collaborates with `slow_path()` to help converge `protect()`. That idea is similar to that of WFE, but Crystalline-W differs substantially due to its asynchronous reclamation.

Data Formats. The *rsrv* array is modified to contain pairs {.V, .T} by attaching *tags* to both the list and era fields as shown in Figure 8. These tags identify the slow path cycle and are used to prevent spurious updates. For the Crystalline-W methods that are only shown in Figures 6 and 9, the .V component is implied. The *rsrv* array is also extended by two special reservations, i.e., each thread has $MAX_IDX + 2$ indices. The two extra reservations are used internally by `help_thread()`.

Each thread maintains its *state* for the slow path. The *result* field of *state* is used for both input and output. On input, a current slow path cycle is advertised. Output contains a retrieved pointer with its corresponding era. In Figure 8, we show how input and output values must be aligned. To distinguish the two cases unambiguously, `invptr` is placed as a pointer on input. Finally, *slow_counter* counts the number of threads in the slow path. This is used to optimize `increment_era()`.

Retirement Status. `alloc_node()` initializes `blink`. When its value is changed in `retire()`, it will no longer be `nullptr`. `blink` will point to REFS (for SLOT nodes). This indicates that the node is already in the process of retirement. Furthermore, the reference counter in REFS becomes immediately reachable from SLOT. To identify the REFS node itself, we steal one bit from the `blink` field. (See RNODE, which is defined differently for Crystalline-W.)

Implementation. Crystalline-W's high-level changes (with respect to Crystalline-L) are shown in Figure 10. 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 [35] corresponding slow-path

```
1  template <typename type> struct Tag {
2    type   V;    // Value
3    uint64 T;    // Tag, or Era for State::result
4  };
5
6  struct Reservation {
7    Tag<Node*>  list;   // Init: {.V = null, .T = 0}
8    Tag<uint64> era;    // Init: {.V = 0, .T = 0}
9  };
10
11 struct State {
12   Tag<void*> result; // Init: {.V = null, .T = 0}
13   uint64     era;    // Init: 0
14   Node*      parent; // Init: nullptr
15   Node**     obj;    // Init: nullptr
16 };
17
18 Reservation rsrv[MAX_THREADS][MAX_IDX+2];
19 State state[MAX_THREADS][MAX_IDX];
20 Node* parents[MAX_THREADS]; // Init: (all) nullptr
21 int slow_counter = 0;
22 // Help other threads before incrementing the era
23 Node* alloc_node(int size) {
24   if (alloc_cnt++ % ALLOC_FREQ == 0) increment_era();
25   Node* node = malloc(size);
26   node->birth = global_era;
27   node->blink = nullptr; // Retired if != nullptr
28   return node;
29 }
30
31 // Use the fast-path-slow-path method
32 Node* protect(Node** obj, int index, Node* parent) {
33   int tries = MAX_TRIES;
34   uint64 prev_era = rsrv[TID][index].era.V;
35   while (--tries != 0) {
36     Node* ptr = *obj;
37     uint64 curr_era = global_era;
38     if (prev_era == curr_era) return ptr;
39     prev_era = update_era(curr_era, index);
40   }
41   return slow_path(obj, index, parent);
42 }
```

Fig. 10. Crystalline-W (API function changes).

variables. Finally, Crystalline-W defines the *parent* array to keep parent object references and facilitate object handover. 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 10 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 11 shows changes to `try_retire()` and `traverse()`. These methods use list tainting, as previously discussed. Also, unlike WFE [35], we use two slow-path tag transitions (odd and even). This is needed to make the number of iterations finite in some loops (see Lemmas 5.2 and 5.3) by collaborating with `try_retire()` which will skip odd tags. Figure 11 also shows `increment_era()`'s implementation as used by `alloc_node()`.

Crystalline-W's slow-path and helper thread routines are demonstrated in Figure 13. These routines use several utility methods shown in Figure 12. The idea is similar to that of WFE [35], with one major difference: we use the *parent* array to keep parent references to facilitate object handovers, as previously mentioned.

Utility methods in Figure 12 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 [35] 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.

Object Handover. Crystalline-W uses one special reservation for the parent object and the other one for a retrieved object. However, there is a race condition if the parent object is retired by the time the reservation is made in `help_thread()`. Likewise, there is a race condition in the opposite direction when assigning the retrieved object to the thread which is in `slow_path()`. (WFE avoids these issues by scanning the list of retired nodes twice, which is impossible with Crystalline-W.)

```
1  // Redefine RNODE to encode REFS links: steal one bit
2  // to indicate REFS nodes (also applies to other
3  // functions that previously used dummy RNODE)
4  #define IS_RNODE(x) (x & 0x1) // Check if a REFS link
5  #define RNODE(x)    (x ^ 0x1) // Encode or decode REFS
6
7  // Another huge addend for the slow path
8  // (in addition to previously defined REFC_PROTECT)
9  const uint64 REFC_PROTECT_HANDOVER = 1 << 62;
10
11 // Adds a special REFS-terminal node and list tainting
12 void traverse(Node* next) {
13   while (next != nullptr) {
14     Node* curr = next;
15     if (IS_RNODE(curr)) { // REFS-terminal node
16       // It is always the last node, exit
17       Node* refs = RNODE(curr);
18       if (FAA(&refs->refc, -1) == 1) free_batch(refs);
19       break;
20     }
21     next = SWAP(&curr->next, invptr); // Tainting
22     Node* refs = curr->blink;
23     if (FAA(&refs->refc, -1) == 1) free_batch(refs);
24 } }
25
26 // Increments the global era, replaces regular FAA
27 // (need to help other threads first)
28 void increment_era() {
29   if (slow_counter != 0) {
30     for (int i = 0; i < MAX_THREADS; i++) {
31       for (int j = 0; j < MAX_IDX; j++) {
32         if (state[i][j].result.V == invptr)
33           help_thread(i, j);
34   } } }
35   FAA(&global_era, 1);
36 }
37 void try_retire() { // A wait-free replacement
38   uint64 min_birth = batch.refs->birth;
39   Node* last = batch.first;
40   // Also check odd tags to bound slow-path loops
41   for (int i = 0; i < MAX_THREADS; i++) {
42     for (int j = 0; j < MAX_IDX+2; j++) {
43       if (rsrv[i][j].list.V == invptr ||
44           (rsrv[i][j].list.T & 0x1)) continue;
45       if (rsrv[i][j].era.V < min_birth ||
46           (rsrv[i][j].era.T & 0x1)) continue;
47       if (last == batch.refs)
48         return; // Ran out of nodes, exit
49       last->slot = &rsrv[i][j];
50       last = last->bnext;
51   } }
52   Node* curr = batch.first; // Make the loop wait-
53   int64 cnt = -REFC_PROTECT; // free by tainting
54   for (; curr != last; curr = curr->bnext) {
55     Reservation* slot = curr->slot;
56     if (slot->list.V == invptr) continue;
57     curr->next = nullptr;
58     Node* prev = SWAP(&slot->list.V, curr);
59     if (prev != nullptr) {
60       if (prev == invptr) { // Inactive previously
61         if (CAS(&slot->list.V, curr, invptr))
62           continue; // Try to rollback
63       } else { // Tainted: traverse a chopped tail
64         if (!CAS(&curr->next, nullptr, prev))
65           traverse(prev);
66     } }
67     cnt++;
68   }
69   if (FAA(&batch.refs->refc, cnt) == -cnt)
70     free_batch(batch.refs);
71   batch.first = nullptr; batch.counter = 0;
72 }
```

Fig. 11. Crystalline-W's try_retire(), traverse(), and increment_era().

These race conditions, obviously, are only considered for retired objects. To properly hand over a parent object from `slow_path()`, we maintain the *parents* array. When initializing the slow path, *state* passes a pointer to the parent. When `helper_thread()` enters, it initializes its entry in *parents* with this value. Subsequently, when `slow_path()` exits, it calls `handover_parent()`. The latter iterates through *parents* and replaces a parent to `nullptr` with CAS. If successful, the reference counter of the parent is incremented. (REFC_PROTECT_HANDOVER protects from premature deallocation.) The other side subsequently detects the handover and dereferences the object.

In a similar problem, when `help_thread()` passes the retrieved pointer back to `slow_path()`, we access the actual retrieved (already retired) node and increment its batch reference counter. Since we know the exact node, and `protect()` is not yet complete, we clean up the existing reservation list and attach this batch's REFS as a "terminal node."

REFS-Terminal Nodes. The idea behind REFS-terminal nodes is that they can only appear at the very end of the list. The same REFS-terminal node can appear in as many lists as desirable. We steal one bit from the preceding pointer to indicate a REFS-terminal node. When encountering this node, `traverse()` immediately terminates.

```
1  // Hand over the parent object if it is retired
2  void handover_parent(Node* parent) {
3    if (parent && parent->blink != nullptr) {
4      Node* refs = get_refs_node(parent);
5      FAA(&refs->refc, REFC_PROTECT_HANDOVER);
6      int64 cnt = -REFC_PROTECT_HANDOVER;
7      for (int i = 0; i < MAX_THREADS; i++)
8        if (CAS(&parents[i], parent, nullptr)) cnt++;
9      FAA(&refs->refc, cnt);
10 } }
11
12 uint64 get_birth_era(Node* node) { // Get parent's
13   if (node == nullptr) return 0;    // birth era
14   uint64 birth_era = node->birth;
15   Node* link = node->blink;
16   // For already retired SLOT nodes, use REFS value
17   if (link != nullptr && !IS_RNODE(link))
18     birth_era = link->birth;
19   return birth_era;
20 }
21 // Make the tag+1 transition and detach an old list
22 void detach_nodes(int i, int j, int tag) {
23   // A simple era tag transition: tag -> tag+1
24   CAS(&rsrv[i][j].era.T, tag, tag+1);
25   // Detach nodes and increment the list tag
26   do { // Bounded by MAX_THREADS (try_retire checks
27     old = rsrv[i][j].list; // the era tag)
28     if (old.T != tag) break;
29     bool success = WCAS(&rsrv[i][j].list,
30                         old, { nullptr, tag+1 });
31   } while (!success);
32   return success ? old.V : invptr; // Previous value
33 }
34
35 // Get REFS node from any node in a batch
36 Node* get_refs_node(Node* node) {
37   Node* refs = node->blink;
38   if (IS_RNODE(refs)) refs = node; // Itself
39   return refs;
40 }
```

Fig. 12. Crystalline-W's utility functions for the slow path.

5 CORRECTNESS

Crystalline-L/-LW is based on Hyaline-1S, for which correctness is considered in [37]. In the extended version of the paper [39], we present complete Crystalline-W's wait-free arguments. Below we discuss several non-trivial cases that are relevant for the wait-free progress.

Lemma 5.1. `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 11) 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 5.2. `detach_nodes()` *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 12, changes the era tag unless it was already changed by a concurrent thread. For the loop in Lines 26-31, Figure 12 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 13), i.e., after `detach_nodes()` in Line 38 or 81 (Figure 13), 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 11) 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 13.) Only threads that are already in-progress will proceed and potentially contend because of unconditional list pointer updates in Line 58, Figure 11. The number of such threads is bounded by MAX_THREADS, as subsequent `try_retire()` calls will observe that the era tag is odd. □

Lemma 5.3. *The loop in Lines 92-99, Figure 13 is bounded by at most MAX_THREADS iterations.*

Proof. The proof is similar to that of Lemma 5.2. The only major 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 11). □

```
1  Node* slow_path(Node** obj, int idx, Node* parent)
2  { // Getting parent's birth is tricky: use 'birth'
3    // for non-retired nodes, else get the minimum
4    // birth from REFS, see get_birth_era()
5    uint64 parent_birth = get_birth_era(parent);
6    FAA(&slow_counter, 1);
7    state[TID][idx].obj = obj;
8    state[TID][idx].parent = parent;
9    state[TID][idx].era = parent_birth;
10   uint64 tag = rsrv[TID][idx].era.T;
11   state[TID][idx].result = { invptr, tag };
12   uint64 prev_era = rsrv[TID][idx].era.V;
13   do { // Bounded by MAX_THREADS
14     Node* list, * ptr = *obj;
15     uint64 curr_era = global_era;
16     if (curr_era == prev_era &&
17         WCAS(&state[TID][idx].result,
18              { invptr, tag }, { nullptr, 0 })) {
19       rsrv[TID][idx].era.T = tag+2;
20       rsrv[TID][idx].list.T = tag+2;
21       FAA(&slow_counter, -1);
22       return ptr; // DONE
23     }
24     // Dereference previous nodes & update an era
25     if (rsrv[TID][idx].list.V != nullptr) {
26       list = SWAP(&rsrv[TID][idx].list.V,
27                   nullptr);
28       if (rsrv[TID][idx].list.T != tag)
29         goto produced; // Result was just produced
30       if (list != invptr) traverse(list);
31       curr_era = global_era;
32     }
33     // WCAS fails only when the result is produced
34     WCAS(&rsrv[TID][idx].era,
35          { prev_era, tag }, { curr_era, tag });
36     prev_era = curr_era;
37   } while (state[TID][idx].result.V == invptr);
38   list = detach_nodes(TID, idx, tag) //tag+1 state
39 produced:
40   ptr = state[TID][idx].result.V;
41   uint64 era = state[TID][idx].result.T;
42   rsrv[TID][idx].era.V = era;
43   rsrv[TID][idx].era.T = tag+2;
44   rsrv[TID][idx].list.T = tag+2;
45   // Check if the obtained node is already retired
46   if (ptr && ptr->blink != nullptr) {
47     Node* refs = get_refs_node(ptr);
48     FAA(&refs->refc, 1);
49     if (list != invptr) traverse(list);
50     list = SWAP(&rsrv[TID][idx].list.V,
51            RNODE(refs)); // Put a REFS-terminal node
52   }
53   FAA(&slow_counter, -1);
54   // Traverse the previously detached list
55   if (list != invptr) traverse(list);
56   // Hand over the parent to all helper threads
57   handover_parent(parent);
58   return ptr; // DONE
59 }
```

```
60  void help_thread(int i, int j) {
61    Tag<void*> result = state[i][j].result;
62    if (result.V != invptr) return;
63    uint64 era = state[i][j].era;
64    Node* parent = state[i][j].parent;
65    if (parent != nullptr) {
66      rsrv[TID][MAX_IDX].list.V = nullptr;
67      rsrv[TID][MAX_IDX].era.V = era;
68      parents[TID] = parent; // Advertise for a handover
69    }
70    Node** obj = state[i][j].obj;
71    uint64 tag = rsrv[i][j].era.T;
72    if (tag != result.T) goto changed;
73    uint64 curr_era = global_era;
74    do { // Bounded by MAX_THREADS
75      prev_era = update_era(curr_era, MAX_IDX+1);
76      Node* ptr = *obj;
77      uint64 curr_era = global_era;
78      if (prev_era == curr_era) {
79        if (WCAS(&state[i][j].result, // Published the
80              result, { ptr, curr_era })) { // result
81          Node* list = detach_nodes(i, j, tag);
82          if (list != invptr) traverse(list);
83          do { // Set the new era, <= 2 iterations
84            old = rsrv[TID][idx].era;
85            if (old.T != tag+1) break;
86          } while (!WCAS(&rsrv[TID][idx].era,
87                         old, { curr_era, tag+2 }));
88          // If the obtained node is already retired
89          if (ptr && ptr->blink != nullptr) {
90            Node* refs = get_refs_node(ptr);
91            FAA(&refs->refc, 1);
92            do { // Bounded by MAX_THREADS
93              old = rsrv[TID][idx].list;
94              if (old.T != tag+1) break;
95              ok = WCAS(&rsrv[TID][idx].list,
96                        old, { RNODE(refs), tag+2 });
97              if (ok && old.V != invptr) traverse(old.V)
98              if (ok) goto done;
99            } while (!ok);
100           FAA(&refs->refc, -1); // Already inserted
101         } else { // A simple tag transition
102           CAS(&rsrv[TID][idx].list.V, tag+1, tag+2);
103       } }
104       break;
105     }
106   } while (state[i][j].result == result);
107 done:
108   Node* lst=SWAP(&rsrv[TID][MAX_IDX+1].list.V,invptr)
109   traverse(lst);
110 changed: // If handover occurs, dereference the parent
111   if (parent != nullptr) {
112     if (SWAP(&parents[TID], nullptr) != parent) {
113       Node* refs = get_refs_node(parent);
114       if (FAA(&refs->refc, -1) == 1) free_batch(refs);
115     }
116     Node* lst=SWAP(&rsrv[TID][MAX_IDX].list.V,invptr)
117     traverse(lst);
118 } }
```

Fig. 13. Crystalline-W's slow-path methods.

6 EVALUATION

We evaluate all SMR schemes for up to 192 threads on 4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores in total), 256 GB RAM. (HyperThreading is OFF for reliable measurements.) We use clang 11.0 with `-O3`. (clang performs marginally better than gcc 9.2.1 for *all* SMR schemes due to its better C++11 atomics optimizations.) Similar to [35, 54], we use jemalloc [17] due to its better performance. We have also considered mimalloc [28] but did not find any significant benefit for the workload presented in this section. Unfortunately, production-ready memory allocators which are both (fully) wait-free and high-performant do not seem to exist at the moment. Thus, our overall progress claims exclude the memory allocator progress properties.

We implemented Crystalline-L/-LW/-W schemes in C++11 and integrated them into the benchmark from [35, 54]. We compared our Crystalline schemes against well-known or closely related baselines (Figure 14). We do not present Crystalline-LW since its results closely match that of Crystalline-L. We also skip: (i) classical reference counting [16, 20, 30, 53] since it needs a different (intrusive) API and is already known to be slower than the evaluated approaches; (ii) OS-based approaches, e.g., DEBRA+ [9], NBR [49] as they are inevitably blocking; (iii) PEBR [26] and lock-free garbage collectors [10, 14] due to API differences and lack of performance benefits [37] compared to Hyaline, and consequently – Crystalline; (iv) DRC [5], a scheme that enhances usability but is generally slower than other presented schemes (DRC also lacks wait-freedom); (v) VBR [47], an approach with optimistic writes, due to substantial API differences which require data structure changes, e.g., inserting roll-back instructions (VBR also lacks wait-freedom). Also, despite great performance, NBR and VBR lack a uniform set of API operations and thus forfeit easy integration [48], which puts them at odds with all other presented schemes.

We have extended the existing benchmark to support skip lists. We based our skip-list implementation on the approaches described in [19, 25]. The approach in [25] avoids lazy skip-list traversals and is thus more suitable for the HP-like interface (HP, IBR, HE, Hyaline-1S, Crystalline). However, the algorithm appeared to have an issue in one corner case with overlapping insertion and removal operations, which we have fixed. The approach in [19] did not have this issue but was only considering lazy traversals for the EBR-like interface. Furthermore, additional adaptations were needed for manual reclamation. Since skip-list nodes effectively maintain multiple sublists (for faster traversals), we use a reference counter in each node to indicate how many sublists the node is still attached to. The reference counter is only changed during deletion from each of the sublists. In our tests, we have used nodes with up to six levels (i.e., that reside in up to six sublists).

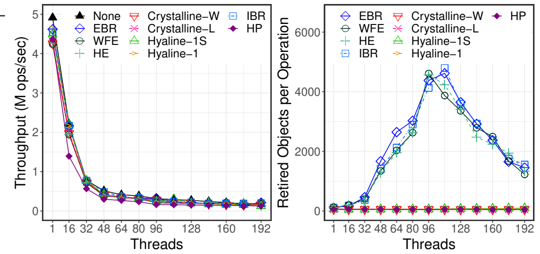
The original benchmark was suboptimal for HP, HE, and WFE due to excessive cache misses when scanning lists of retired nodes. As in [37], we take per-thread snapshots of hazard pointers (or eras) before scanning the list. This improves performance greatly, especially for HP. This optimization already existed for IBR. EBR, Hyaline-1/-1S, and Crystalline-L/-LW/-W do not need it.

As in [37], we modified how memory objects are retired in the existing schemes. The original benchmark used an indirection when retiring by allocating thread-local list nodes (they store pointers to retired objects), which creates circular allocator dependency. This is problematic because typical memory allocators are blocking. When using this indirection, `retire()` becomes also blocking, effectively forfeiting all SMR's lock-/wait-free guarantees. Furthermore, a wait-free `malloc` implementation may itself require memory reclamation, making it a chicken-and-egg situation. We note that our change did not significantly impact the results for the existing schemes.

Inability to allocate memory in `retire()` without affecting progress properties is unfortunate for other reasons. It could have opened a way for additional optimizations in some algorithms. For example, HE and WFE could have leveraged batched reclamation using the minimum birth era of the batch and deduplicated snapshots to improve performance. Although static memory

None	no reclamation (leaking memory)
Hyaline-1	(non-robust) Hyaline-1 [37]
Hyaline-1S	(robust) Hyaline-1S [37]
HP	the hazard pointers scheme [29]
HE	the hazard eras scheme [43]
IBR	2GEIBR (interval-based) [54]
WFE	the wait-free eras scheme [35]
EBR	epoch-based reclamation

Fig. 14. Evaluated reclamation schemes.



(a) Throughput (b) Retired objs

Fig. 15. Wait-free CRTurnQueue.

provisioning for the same purpose is still feasible, it is not very practical. In HE and WFE, the worst-case (as opposed to average-case) number of unreclaimed objects is much higher than in HP. A lot of memory would have been wasted to accommodate the worst-case scenario and skew memory consumption results unfavorably for HE and WFE.

In the benchmark, data structures implement abstract key-value interfaces such as `insert()`-and-`delete()` or `get()`-and-`put()`. For each data point, the benchmark prefills the data structure with 50,000 elements and runs 10 seconds. Each thread then randomly chooses the corresponding abstract operation. The key for each operation is randomly chosen from the range (0, 100,000). We run the experiment 5 times and report the average. Both Crystalline-W and WFE set the fast path threshold (MAX_TRIES) to 16.

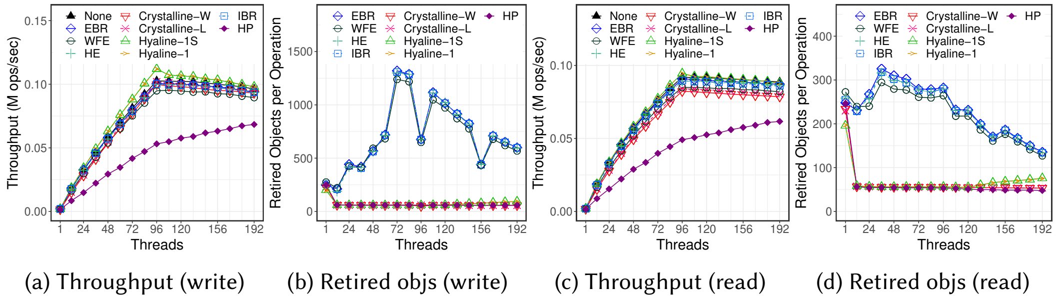
The default parameters in [35, 54] are suboptimal for our system even for existing schemes, so we adjust them for a fair comparison. The benchmark's `epochf=110` and `emptyf=120` appear to be optimal for *all* existing schemes as they attain the best possible throughput with good memory efficiency. These parameters are also optimal for all Crystalline schemes. Thus, all schemes are tested identically. (Note that `emptyf` is used as RETIRE_FREQ for `try_retire()` in Crystalline.)

We focus on common performance metrics including throughput and memory efficiency. Wait-free data structures are typically more difficult to implement and involve extra variables in their evaluation. Thus, we primarily focus on lock-free data structures: (i) a hash map [29] and sorted list [21, 29] already implemented in the original benchmark; (ii) a skip list that we have implemented. This choice of data structures allows to test all reclamation schemes under different conditions (short operations in hash tables, long traversal operations in linked lists, and long but expedited operations in skip lists). We also evaluate CRTurnQueue [45], which has been previously evaluated with WFE [35]. CRTurnQueue is a memory-bounded wait-free queue, unlike other queues such as WFQUEUE [55], which has potentially unbounded memory usage [42]. CRTurnQueue is faster [35, 45] than Kogan-Petrank's memory-bounded wait-free queue [27].

Global retire lists are mandatory features of all Hyaline and Crystalline algorithms by their design and do not impose any extra overheads. In contrast, EBR, IBR, HE, WFE, and HP use thread-local lists. While global lists can provide better workload balancing and also be *potentially* implemented in these algorithms, they would not come naturally: they require additional memory barriers and atomic operations, which will likely degrade overall memory reclamation performance. Thus, we evaluate all schemes based on their original design.

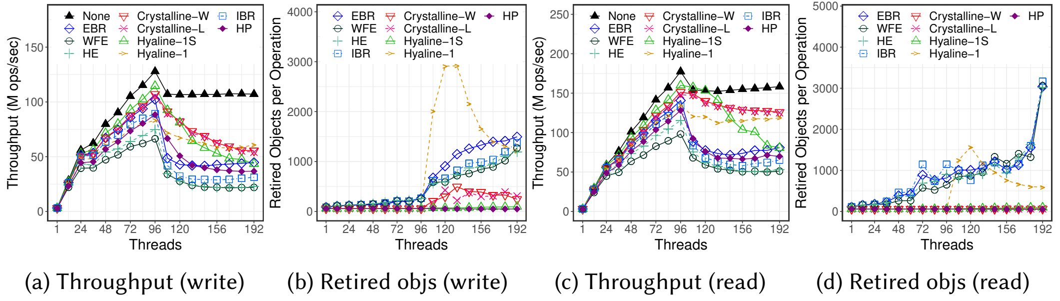
Finally, the original benchmark is not designed to properly count objects in global retire lists, but we are using a modified method that works with both global and local retire lists from [37].

We first ran a write-dominated workload (50% of `insert()` and 50% of `delete()` operations). This workload stresses SMR algorithms greatly. We found that Crystalline-L/-LW/-W generally



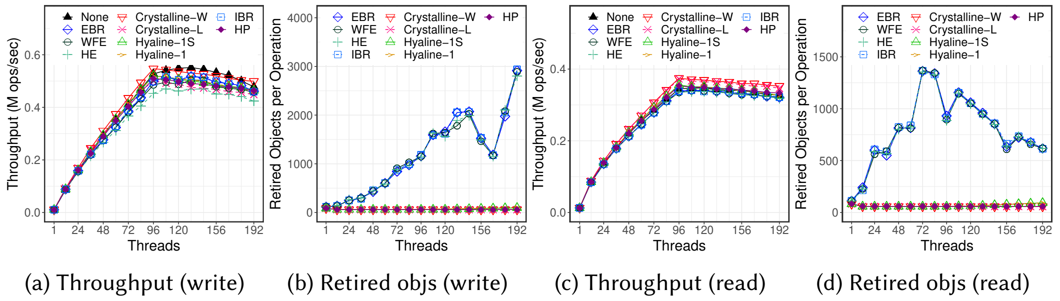
(a) Throughput (write) (b) Retired objs (write) (c) Throughput (read) (d) Retired objs (read)

Fig. 16. Lock-free LinkedList. (Crystalline-L and Crystalline-LW are close, showing Crystalline-L only.)



(a) Throughput (write) (b) Retired objs (write) (c) Throughput (read) (d) Retired objs (read)

Fig. 17. Lock-free HashMap. (Crystalline-L and Crystalline-LW are close, showing Crystalline-L only.)



(a) Throughput (write) (b) Retired objs (write) (c) Throughput (read) (d) Retired objs (read)

Fig. 18. Lock-free SkipList. (Crystalline-L and Crystalline-LW are close, showing Crystalline-L only.)

outperform other SMR schemes in both throughput and memory efficiency. Hyaline-1/-1S are often on par but Hyaline's memory efficiency can sometimes be worse than that of Crystalline-L/-LW/-W due to larger granularity and lack of dynamic batches present in more advanced Crystalline schemes. Though Crystalline's theoretical memory bound is not necessarily better than that of HE and WFE, its practical efficiency is better than that of HE/WFE and is often comparable to HP's.

Figure 15 shows CRTurnQueue results. The write-intensive workload is typical for queues and guarantees that queues will not grow indefinitely. Queues generally do not scale well, and throughput is almost identical (Figure 15a) for all schemes. Crystalline and Hyaline schemes show exceptional memory efficiency, which is on par with HP (Figure 15b).

For the sorted list, HP has the worst throughput (Figure 16a), and WFE exhibits a consistent slight overhead which is already discussed in [35]. Other schemes, including Crystalline, achieve

better throughput. Hyaline-1/1S are slightly faster than other schemes, including Crystalline: linked lists are dominated by traversal operations. They heavily stress the `protect()` operation, which is not needed in Hyaline-1 and EBR (or cheaper in Hyaline-1S) but crucial for achieving strong progress properties in Crystalline, HP, and HE. All Hyaline and Crystalline variants are very memory efficient (Figure 16b).

For the hash map, Crystalline-L/-LW/-W achieve very high throughput (Figure 17a), which is especially evident for oversubscribed scenarios, where the gap with other algorithms is as large as 2x. HP's throughput is worse than that of Crystalline-L/-LW/-W. Hyaline-1's throughput is worse than that of Crystalline, which can be explained by a larger granularity of reservations. WFE has the worst throughput. Memory efficiency (Figure 17b) of Crystalline-L/-LW/-W is mostly superior to all algorithms except HP. Hyaline-1's efficiency greatly reduces after oversubscription. Crystalline-L/-LW/-W and Hyaline-1/-1S are more memory efficient before hitting the oversubscribed situation. In the oversubscribed situation, however, the throughput also remains higher than that of other schemes (including Hyaline-1S) which implies that the total number of allocated objects was higher in the first place. As the throughput reduces to that of EBR (192 threads), the number of unreclaimed objects evens out.

Figures 18a and 18b present results for the skip list, which benefits from reclaiming memory fast (Crystalline, Hyaline, HP). Sometimes, reusing memory is more beneficial than allocating fresh memory (hence the leaky *None* baseline is not the best one) since the latter involves expensive operations (updating page tables, expanding the heap, etc). In skip lists, objects are typically larger and the average number of unreclaimed objects is higher than in linked lists, which should explain the observed difference. Overall, Crystalline exhibits superior throughput and memory efficiency.

The benefits of the Crystalline schemes are even stronger in a read-dominated workload (90% of `get()` and 10% of `put()` operations). This is partially due to a better balancing of the reclamation workload across all threads. As in the write-dominated workload, the hash map (Figures 17c, 17d) achieves the highest throughput, especially in oversubscribed scenarios. At the same time, Crystalline-L/-LW/-W achieve exceptional memory efficiency which is on par with hazard pointers. (Hyaline-1, EBR, IBR, HE, and WFE are visibly less memory efficient.) We only observed an overhead in the linked list shown in Figure 16c. This overhead is similar to that of WFE [35] and is due to a higher register spillover when `protect()` gets inlined into the code. In linked lists, many nodes need to be traversed, increasing the frequency of `protect()` calls, especially in read-dominated workloads. However, `protect()` does not update eras that frequently, and a better optimization strategy would be to avoid a premature register spillover, e.g., via customized assembly code. Although Crystalline is worse than Hyaline in this test (Hyaline's `protect()` is cheaper), it is still more memory efficient than most other schemes, including Hyaline and WFE. Finally, for the skip list (Figure 18c and 18d), Crystalline-L/-LW/-W, again, show superior throughput to other schemes due to their better memory efficiency and faster reclamation.

Overall, the overhead of Crystalline-W (vs. Crystalline-L/-LW) is negligible. All these schemes mostly outperform existing schemes when considering *both* throughput and memory efficiency, which make them appealing for many lock- and wait-free data structures. Moreover, Crystalline-W visibly outperforms WFE, the only fully wait-free scheme that previously existed, even in non-oversubscribed scenarios. Crystalline-W is also substantially more memory efficient than WFE.

Snapshots. We ignore snapshot overheads, but for HP/IBR/HE/WFE snapshots can create more memory inefficiency than the schemes themselves if the number of threads is high. For 144 threads and 16 local pointers in HP and HE, snapshots additionally require 2.5 MB, which is significant. (For example, if the number of unreclaimed objects is as high as 2000 and the object size is 64 bytes, we only use 128 KB.) This makes Crystalline even more memory efficient.

7 RELATED WORK

Blocking Techniques. EBR [19] is a common scheme, where threads explicitly make reservations. At the start of an operation, EBR records the global epoch value. At the end, EBR resets the reservation. Quiescent-based reclamation (QSBR) [22] makes this automatic as threads go through a "quiescent" state. Stamp-it [41] can bound reclamation overheads. These techniques do not bound memory usage, i.e., are blocking when memory is exhausted. Hyaline-1 [34, 37] implements a similar API.

Memory Usage. To bound memory usage and improve usability, several techniques were developed that exploit OS support. As mentioned in [10], these approaches are not strictly non-blocking, because typical OS primitives such as *signals* use locks internally (e.g., in Linux). ThreadScan [4] is one such mechanism which uses signals. Once the signal is received, every thread scans its own stack and registers to report its working-set to the reclaiming thread. The reclaiming thread uses the collected information to determine unreachable objects that can be reclaimed. Forkscan [3] is a ThreadScan extension that reduces the interruption time to only working threads. DEBRA+ [9] and NBR [49] are other examples that use signals. QSense [7] relies on the OS scheduler behavior. (Thus, it is hard to guarantee non-blocking behavior in general.) It mixes QSBR [22] with HP [29].

Lock-Free Techniques That Require Restarting. IBR [54] and PEBR [26] improve upon EBR and are not dependent on OS environments. IBR only defends against threads that are stalled indefinitely. Starving threads may still reserve an unbounded number of blocks. Thus, IBR's authors advise to restart operations that are unable to progress. Although restarting is trivial for simple data structures such as linked lists, it is more problematic for complex data structures. In the same vein, PEBR's authors demonstrated their scheme while assuming restarting. PEBR inherently requires restarting to retain simple EBR-like semantics [1]. Moreover, PEBR's API does not put any explicit bound on how many blocks each thread can reserve. PEBR's authors only compare against EBR, and PEBR's performance appears to be only 85-90% of EBR's, i.e., worse than IBR's and only marginally better than HP's. Due to potentially unbounded memory usage and restarting, IBR and PEBR are not lock-free in general. Hyaline-1S [37] has an IBR-like API and similar progress guarantees.

General Lock-Free Techniques. A number of lock-free approaches that bound memory usage were proposed over the years. Traditional reference counting [16, 20, 30, 53] is fine-grained but has high overheads, especially in read-dominated workloads. HP [29] and pass-the-buck [23, 24] are also very precise as they track each object individually. However, these techniques still have high overheads due to their extensive use of memory barriers for each pointer retrieval. Some approaches [8] aim to reduce overheads of HP, but they are only suitable for specific data structures such as linked lists. Other techniques [11, 12] do not have this limitation, but require data structures to be represented in a normalized form [51]. This, however, can be burdensome. FreeAccess [10] removes this burden and uses a garbage collector. However, it does not transparently handle SWAP, which coincidentally is a prerequisite for making our Crystalline-LW/-W algorithms wait-free. OrcGC [14] is another lock-free garbage collector with great performance, but it can be slower in some tests than HP. VBR [47], an approach with optimistic writes and great performance, requires data structure adaptations. Similar optimistic lock-free techniques [31] can be used for specific applications, e.g., memory allocators. DRC [5], a scheme that enhances usability, can be slower than other lock-free schemes. Hazard eras (HE) [43] is a lock-free scheme with bounded memory usage, which is inspired by HP but uses epochs to expedite the algorithm. Our Crystalline-L scheme also provides lock-free progress guarantees but is often faster and more efficient.

Wait-Free Techniques. HP and HE support restartable wait-free data structures [45]. In fact, HP's original paper [29] claimed HP to be "wait-free". Crystalline-LW provides identical wait-freedom

guarantees if restarting is allowed. The above schemes are not fully wait-free, which makes it difficult to use them in *every* wait-free data structure without modification. Wait-free SMR has recently received increasing attention. OneFile [46] implements Software Transactional Memory (STM) with wait-free reclamation. CX [13] implements a universal construct which converts the sequential specification of a data structure into a wait-free implementation. Although OneFile and CX enable the implementation of many wait-free data structures, customized, hand-crafted data structures can often better utilize parallelism and achieve higher overall performance. To that end, a wait-free SMR approach, WFE [35] was proposed. Crystalline-W goes beyond WFE by achieving high memory efficiency and performance over a broader range of conditions. Crystalline can also adopt a recent auto SMR approach [6] that already extends Hyaline and other schemes.

ERA Theorem. It is proven [48] that at most 2 out of 3 properties (robustness, easy integration, wide applicability) can be achieved. Crystalline achieves the first two properties.

Lazy List Traversals. Given that Crystalline, like many other existing schemes such as IBR, HP, HE, WFE, and Hyaline-1S, does not achieve the wide applicability property, care must be taken to handle traversals in linked lists, skip lists, and similar data structures. Whereas non-robust schemes, such as EBR and Hyaline-1, can work with the original lock-free linked list [21], many robust schemes (Crystalline, IBR, HP, HE, WFE, and Hyaline-1S) require a modification [29] that timely retires deleted list nodes and thus avoids read-only lazy traversals across logically deleted nodes. A similar approach is used with skip lists: a skip list [25] timely retires nodes, whereas a skip list [19] uses lazy traversals instead. One consequence of non-lazy traversals is that a search operation may need to be restarted from the very beginning whenever, due to overlapping operations, the data structure state diverges substantially and cannot be recovered locally. However, this restart is fundamentally unavoidable in the algorithm: it is already required in insertion and deletion operations irrespective of lazy traversals and the reclamation scheme used. It should be differentiated from restarts due to the reclamation scheme itself, which are avoidable with both WFE and Crystalline-W.

Wait-Free Restarts. Unavoidable restarts have to be bounded to guarantee wait-freedom. Timnat-Petrank's method [50, 51] can be used for many data structures, including linked lists. In fact, [50] explicitly mentions HP as a solution for memory management in their wait-free linked list, which makes it possible to use the same approach with Crystalline. Crystalline provides a compatible API but otherwise even stronger progress guarantees than HP. In [50], the traversal operation can be restarted when the list changes in a bounded manner. That makes wait-free traversals feasible.

8 CONCLUSION

Crystalline is a family of memory-bounded SMR schemes. Crystalline-W's uniquely distinguishing aspect is that it incorporates all desirable properties of prior schemes including wait-freedom, asynchronous reclamation, and balanced reclamation workload in the *same* algorithm. The only existing wait-free scheme, WFE, lacks the two latter properties. Unsurprisingly, Crystalline-W outperforms WFE in almost all test cases. Crystalline-W is based on a simpler wait-free algorithm, Crystalline-LW. Crystalline-LW, in turn, is based on Crystalline-L, which is an improved version of Hyaline-1S that additionally guarantees bounded memory usage even in the presence of starving threads. Crystalline-L introduces *dynamic batches* which resolve one major inconvenience with Hyaline-1S and also improve overall performance in certain cases.

All Crystalline schemes exhibit very high throughput and great memory efficiency, which is especially evident in read-dominated workloads. Crystalline's performance is occasionally superior to that of Hyaline-1S, which proves the benefits of dynamic batches and a more fine-grained API. Finally, Crystalline-W is a great alternative when used with *multiple* non-blocking data structures.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped greatly improve the paper. We especially thank the reviewers for asking challenging questions and suggesting alternative approaches to retire memory objects in existing SMR schemes.

A preliminary version of Crystalline previously appeared as a brief announcement at DISC '21 [36]. An extended and up-to-date version of this paper is available at https://arxiv.org/abs/2108.02763.

This work is supported in part by the startup fund (Pennsylvania State University) as well as ONR under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-21-1-2523, and AFOSR under grant FA9550-16-1-0371 (Virginia Tech).

ARTIFACT AVAILABILITY

The benchmark and data to support this paper are available on Zenodo [40]. Our artifact consists of: (1) a Linux VM image which can be deployed using VirtualBox; (2) source code for the benchmark, all evaluated data structures, and all evaluated reclamation schemes; (3) benchmark scripts to run tests and generate charts. We provide all relevant instructions which describe how to reproduce the results presented in this paper. Crystalline's latest source code and benchmark are also available via GitHub: https://github.com/rusnikola/wfsmr/.

REFERENCES

[1] 2020. Private communication with PEBR's authors.

[2] 2024. Crossbeam: epoch-based memory reclamation. https://docs.rs/crossbeam/latest/crossbeam/epoch/index.html.

[3] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the 12th European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 483–498. https://doi.org/10.1145/3064176.3064214

[4] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/2755573.2755600

[5] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, New York, NY, USA, 526–541. https://doi.org/10.1145/3453483.3454060

[6] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2022. Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI '22)*. 61–75. https://doi.org/10.1145/3519939.3523730

[7] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16)*. ACM, 349–359. https://doi.org/10.1145/2935764.2935790

[8] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Montreal, Quebec, Canada) *(SPAA '13)*. ACM, 33–42. https://doi.org/10.1145/2486159.2486184

[9] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastian, Spain) *(PODC '15)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

[10] Nachshon Cohen. 2018. Every Data Structure Deserves Lock-free Memory Reclamation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 143 (Oct. 2018), 24 pages. https://doi.org/10.1145/3276513

[11] Nachshon Cohen and Erez Petrank. 2015. Automatic Memory Reclamation for Lock-free Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, 260–279. https://doi.org/10.1145/2814270.2814298

[12] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/2755573.2755579

[13] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2020. A Wait-Free Universal Construction for Large Objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. ACM, New York, NY, USA, 102–116. https://doi.org/10.1145/3332466.3374523

[14] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, 205–218. https://doi.org/10.1145/3437801.3441596

[15] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's Law for Tail Latency. *Commun. ACM* 61, 8 (jul 2018), 65–72. https://doi.org/10.1145/3232559

[16] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (01 Dec 2002), 255–271. https://doi.org/10.1007/s00446-002-0079-z

[17] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*. https://www.bsdcan.org/2006/papers/jemalloc.pdf

[18] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[19] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. Univ. of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[20] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. 2009. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems* 20, 8 (Aug 2009), 1173–1187. https://doi.org/10.1109/TPDS.2008.167

[21] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 300–314. https://doi.org/10.1007/3-540-45414-4_21

[22] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270 – 1285. https://doi.org/10.1016/j.jpdc.2007.04.010

[23] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (May 2005), 146–196. https://doi.org/10.1145/1062247.1062249

[24] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 339–353. https://doi.org/10.1007/3-540-36108-1_23

[25] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] Jeehoon Kang and Jaehwang Jung. 2020. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. ACM, New York, NY, USA, 314–328. https://doi.org/10.1145/3385412.3385978

[27] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/1941553.1941585

[28] Daan Leijen, Benjamin G. Zorn, and Leonardo Mendonça de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Asian Symposium on Programming Languages and Systems (APLAS '19)*. https://doi.org/10.1007/978-3-030-34175-6_13

[29] Maged M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[30] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report. University of Rochester, CS. https://www.cs.rochester.edu/u/scott/papers/1995_TR599.pdf

[31] Pedro Moreno and Ricardo Rocha. 2023. Releasing Memory with Optimistic Access: A Hybrid Approach to Memory Reclamation and Allocation in Lock-Free Programs. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. ACM, 177–186. https://doi.org/10.1145/3558481.3591089

[32] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[33] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proceedings of the 33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 28:1–28:16. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[34] Ruslan Nikolaev and Binoy Ravindran. 2019. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto, ON, Canada) *(PODC '19)*. ACM, New York, NY, USA, 419–421. https://doi.org/10.1145/3293611.3331575

[35] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. ACM,

New York, NY, USA, 130–143. https://doi.org/10.1145/3332466.3374540

[36] Ruslan Nikolaev and Binoy Ravindran. 2021. Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation. In *Proceedings of the 35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 60:1–60:4. https://doi.org/10.4230/LIPIcs.DISC.2021.60

[37] Ruslan Nikolaev and Binoy Ravindran. 2021. 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)*. ACM, New York, NY, USA, 987–1002. https://doi.org/10.1145/3453483.3454090

[38] Ruslan Nikolaev and Binoy Ravindran. 2022. wCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA) *(SPAA '22)*. ACM, New York, NY, USA, 307–319. https://doi.org/10.1145/3490148.3538572

[39] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation (an extended arXiv version of this paper). https://arxiv.org/abs/2108.02763

[40] Ruslan Nikolaev and Binoy Ravindran. 2024. Artifact for PLDI'24. https://doi.org/10.5281/zenodo.10775789

[41] Manuel Pöter and Jesper Larsson Träff. 2018. Brief Announcement: Stamp-it, a More Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. ACM, 355–358. https://doi.org/10.1145/3210377.3210661

[42] Pedro Ramalhete and Andreia Correia. 2016. A Wait-Free Queue with Wait-Free Memory Reclamation (Full Version). https://github.com/pramalhe/ConcurrencyFreaks/raw/master/papers/crturnqueue-2016.pdf

[43] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. ACM, New York, NY, USA, 367–369. https://doi.org/10.1145/3087556.3087588

[44] Pedro Ramalhete and Andreia Correia. 2017. Hazard Eras - Non-Blocking Memory Reclamation (Full Version). https://github.com/pramalhe/ConcurrencyFreaks/raw/master/papers/hazarderas-2017.pdf

[45] Pedro Ramalhete and Andreia Correia. 2017. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. ACM, New York, NY, USA, 453–454. https://doi.org/10.1145/3018743.3019022

[46] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 151–163. https://doi.org/10.1109/DSN.2019.00028

[47] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference) (LIPIcs, Vol. 209)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:18. https://doi.org/10.4230/LIPIcs.DISC.2021.35

[48] Gali Sheffi and Erez Petrank. 2023. The ERA Theorem for Safe Memory Reclamation. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing* (Orlando, FL, USA) *(PODC '23)*. ACM, New York, NY, USA, 102–112. https://doi.org/10.1145/3583668.3594564

[49] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, 175–190. https://doi.org/10.1145/3437801.3441625

[50] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2012. Wait-Free Linked-Lists. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS 2012)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 330–344. https://doi.org/10.1007/978-3-642-35476-2_23

[51] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2555243.2555261

[52] R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* Technical Report RJ 5118. IBM Almaden R. Center.

[53] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada) *(PODC '95)*. ACM, New York, NY, USA, 214–222. https://doi.org/10.1145/224964.224988

[54] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3178487.3178488

[55] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168

Received 2023-11-16; accepted 2024-03-31