Fast and Portable Concurrent FIFO Queues With Deterministic Memory Reclamation

Oliver Giersch and Jörg Nolte

Abstract—In this article we present an algorithm for a high performance, unbounded, portable, multi-producer/multi-consumer, lock-free FIFO (first-in first-out) queue. Aside from its competitive performance on current hardware, it is further characterized by its integrated memory reclamation mechanism, which is able to reliably and deterministically de-allocate nodes as soon as the final operation with a reference has concluded, similar to *reference counting*. This differentiates our approach from most other lock-free data structures, which usually require external (generic) memory reclamation or garbage collection mechanisms such as *hazard pointers*. Our deterministic memory reclamation mechanism completely prevents the build up of memory awaiting reclamation and is hence very memory efficient, yet it does not introduce any substantial performance overhead. By utilizing concrete knowledge about the internal structure and access patterns of our queue, we are able to construct and constrain the reclamation mechanism in such a way that keeps the overhead for memory management almost entirely out of the common fast path. The presented algorithm is portable to all modern 64-bit processor architectures, as it only relies on the commonly available and lock-free atomic synchronization primitives *compare-and-swap* and *fetch-and-add*.

Index Terms—Concurrent algorithms, lock-free/non-blocking data structures, memory reclamation, first in-first out (FIFO) queues, shared memory

1 Introduction

It is common wisdom for designing lock-free data structures to separate concerns: Except for a few select examples – usually with specialized access restrictions – practically all lock-free data structures have to contend with the problem of memory reclamation. Since non-intrusive and, in particular, unbounded data structures require dynamically allocated memory, there must also be dynamic de-allocation, once that memory is no longer needed. Due to the concurrent nature of such data structures, it is outright impossible to determine statically when memory can be safely reused by the program. Hence, the assessment when to free memory can only be made at runtime. In programming languages with manual memory management, solving this issue in a lock-free manner is often just as complex and difficult as correctly designing the data structure itself. In contrast, the issue is trivial in automatically managed languages, where a global garbage collector decides when memory can be safely reclaimed based on runtime analysis. Consequently, the separation of these concerns is often justified and also beneficial: It allows the data structure designer to focus on his primary objective, facilitates language agnostic design and keeps use-case restrictions to a minimum, potentially enabling application in zero-allocation environments as well as those with dynamic memory allocation.

• *The authors are with the Brandenburg University of Technology (BTU) Cottbus-Senftenberg, 03046 Cottbus, Germany. E-mail: {oliver.giersch, joerg.nolte}@b-tu.de.*

Manuscript received 26 May 2020; revised 1 June 2021; accepted 2 July 2021. Date of publication 19 July 2021; date of current version 11 Aug. 2021. (Corresponding author: Oliver Giersch.) Recommended for acceptance by J. Lange. Digital Object Identifier no. 10.1109/TPDS.2021.3097901

However, there are notable drawbacks to this approach, as well. For one, reliance on runtime garbage collection is problematic, since most practical general-purpose garbage collectors are not lock-free [1], [2]. For languages with manual memory management, on the other hand, the aforementioned separation of concerns has led to the emergence of numerous generic *memory reclamation* mechanisms. Such mechanisms commonly intend to be applied in a limited manner, e.g., only to the memory used by one specific instance of a data structure and not globally for managing all memory in use by the entire application. Due to their specialized nature and limited application, lock-freedom for these mechanisms is generally attainable in practice [3], [4]. Although most reclamation mechanisms strive to be generically applicable, there are commonly also data structures that are incompatible with certain reclamation mechanisms [3].

A common pattern with dynamic memory reclamation mechanisms is, that memory accesses (either individual or in bulk) have to be *protected* from concurrent reclamation by other threads *before* they occur (with some exceptions [3]). Generally, this requires some form of global announcement to all other threads, indicating that the acting thread is about to access one or more memory objects that *must not* be reclaimed. Such global announcements are fairly costly to realize on modern hardware with multi-level cache hierarchies and expensive coherence protocols, as they require memory fences. Threads attempting to recycle memory, on the other hand, must defer the reclamation of retired memory objects until such a time that *no* other threads are accessing an object anymore. The time in between the record's retirement and its eventual reclamation is often referred to as *grace period* [5]. How the extent of this period is determined is particular to each reclamation mechanism, but it is usually only possible to give an approximate lower bound

for it and, consequently, there is an unavoidable build up of garbage to a certain degree.

In this paper we show, that custom tailoring a memory reclamation mechanism for one specific data structure – in this case a FIFO queue – allows us to achieve better overall performance than comparable approaches using a generic reclamation mechanism such as *hazard pointers* [6] through elision of memory fences *and* to reduce the grace period for retired memory objects to a minimum, thereby preventing a build up of unused memory. We consider this mechanism to be *deterministic* in the same sense as reference counting, i.e., memory is freed at the earliest possible opportunity. This is contrary to, e.g., tracing garbage collectors and most other dynamic lock-free memory reclamation mechanisms, which free memory *lazily* during independent collection cycles. Unlike reference counting, however, our memory reclamation mechanism is highly efficient.

2 Related Work

The first portable, unbounded, multi-producer/multi-consumer and generally applicable lock-free FIFO queue algorithm was presented by Michael and Scott [7]. They give a thorough overview about previous attempts at designing such queues, to which the reader is referred for more information. Their algorithm as well as all previous ones belong to a family of *linked list* queues, i.e., queues that are implemented as traditional linked lists with only one queue element per list node. Such approaches are outclassed by more recent *array-based* approaches in terms of performance by large margins, which is why we chose to not cover them in any greater detail. Array-based approaches also use an outer linked list structure, but each node contains a fixed-size array of slots for storing the individual queue elements.

To our knowledge, the first array-based approach was proposed by Gidenstam *et al.* [8]. Their algorithm is able to achieve substantially better performance than previous ones such as the Michael-Scott queue, but does not utilize the atomic *fetch-and-add* primitive for better scalability around the most hotly contested memory addresses (typically the *head* and *tail* pointers or indices), as many newer algorithms, including our own, do. A different approach is *flat combining* or *batching*, which is based on collecting pending queue operations separately, before a single thread applies (combines) them sequentially [9], [10]. This approach has been shown to exhibit superior scalability properties in comparison to both fine-grained blocking algorithms as well as the lock-free Michael-Scott algorithm. Yet another approach with slightly different semantics are *k-FIFO* queues, which allow up to k elements to be enqueued or dequeued in parallel at the cost of abandoning strict ordering of enqueued elements, as up to $k-1$ elements may be dequeued out-of-order [11].

In terms of competitive lock-free or wait-free FIFO queues we focus our attention on the following four array-based approaches, which, to our knowledge, exhibit the best performance characteristics of all FIFO queues that have so far been proposed in the literature or open source sphere. Morrison and Afek present *LCRQ* [12], which uses a bounded (array-based) reusable *circular ring queue* (CRQ) with FIFO semantics. They use an adapted version of the Michael-Scott algorithm to build a linked list of CRQ nodes, which is the actual LCRQ data structure and ensures that the resulting queue is unbounded. The CRQ algorithm relies on the DCAS (*double word compare-and-swap*) synchronization primitive, which is not available on all CPU architectures, limiting its portability. A recent algorithm by Nikolaev called *(L)SCQ* [13] offers a similar solution to LCRQ but explicitly addresses some of its shortcomings. It uses a *cache remap* mechanism to spread consecutive array accesses around to avoid false sharing without having to over-align every element, thereby drastically reducing the size required for each ring buffer. A similar mechanism is also used in *FFQ* [14], which is another (exclusively) bounded FIFO queue. Unlike the CRQ algorithm, SCQ is able to avoid *live locks* on a single ring, which can not be guaranteed by CRQ, requiring the LCRQ extension in order to reliably avoid such scenarios. Consequently, SCQ can potentially be used as a stand-alone *bounded* FIFO queue as well. The more general variant SCQ2 does also require the DCAS primitive, but they also describe a more specialized SCQ1 variant, which requires further indirection in order to work with only single-word atomic operations. The resulting SCQD algorithm uses two SCQ1 queues for storing available and occupied (integer) indices into a separate array, which stores the actual elements. Another lock-free FIFO queue algorithm called *FAAArray-Queue* has been described by Ramalhete and Correia [15]. Its primary difference to the other algorithms is, that individual array elements are *not* reusable since they are not ring buffers. Effectively, it is a simplified version of LCRQ, which also allows it to function without relying on the DCAS primitive, unlike both LCRQ or SCQ2. Finally, another MPMC FIFO queue algorithm was presented by Yang and Mellor-Crummey [16]. They argue, that their algorithm has *wait-free* rather than lock-free progress guarantees, although Ramalhete has noted certain edge cases, in which this guarantee does not hold due to the queues potentially *unbounded* memory usage [17]. The queue uses its own custom tailored memory reclamation mechanism, which is explicitly not lock-free, since it assigns responsibility for reclaiming nodes (*cleanup*) to a single thread in a mutually exclusive manner, meaning memory could potentially build up unchecked, if the thread assigned to this task were to fail or be significantly delayed.

2.1 Memory Reclamation Mechanisms

Early memory management approaches for lock-free environments had to avoid de-allocating memory or returning memory to the OS at all and instead relied on storing recycled memory in separate *free lists*. Since such lists must also be lock-free and use CAS primitives, they are prone to the ABA problem [6]. One way to prevent this problem is by associating a *tag* value to every pointer swapped out in this manner, but this also requires the DCAS primitive in order to give strong guarantees that the tag value can not overflow [6]. Another early attempt is lock-free reference counting [2], [18], [19], which is quite inefficient in general, however, and has further usefulness limitations on top of that, such as being unable to ever return memory to the operating system. Michael proposed hazard pointers [6], which was one of the first approaches that truly allow memory to be freed while also only requiring single-word atomic instructions. Hazard pointers require (at least) one additional sequentially consistent store for every pointer required to be protected in the general case. Another

category of memory reclamation mechanisms are so called *epoch-based* mechanisms such as EBR and its derivatives [20], [21], [22]. Instead of marking individual memory objects as currently in-use by a thread, these instead mark an entire thread as either *quiescent* or active and restrict reclamation while there are any active threads. This allows reducing the number of fences to only one for batches of one or more data structure operations. However, under contention, EBR-based mechanisms are prone to significantly delaying the reclamation of memory. Therefore, EBR is effectively not lock-free, since a single delayed thread can block all reclamation progress for all other threads. Some mechanisms derived from EBR have been proposed attempting to solve this issue, such as DEBRA+ [21], which uses OS signals to preemptively resolve cases of blocked threads or IBR [22], which restricts reclamation only for bounded intervals of epochs.

Another approach by Ramalhete and Correia called *hazard eras* [23] attempts to combine the small runtime overhead of epoch-based mechanisms with the low memory bound and guaranteed lock-freedom of hazard pointers. Like all reclamation approaches that reduce the number of memory fences required for protecting memory objects from concurrent reclamation, the approach particularly shines when used with data structures, that need to read more than one pointer per operation, such as the list by Michael [24] (which reads at least three pointers in every operation) or the Natarajan-Mital binary search tree [25]. Kang and Jung have proposed PEBR, which is another mechanism following a similar strategy and similar goals as hazard eras [26]. Yet another approach with similarities to both hazard eras and IBR with competitive performance is *Hyaline* proposed by Nikolaev and Ravindran [27].

3 Queue Algorithms

Our own queue algorithm is loosely based on the FAAArrayQueue algorithm by Correia and Ramalhete [15]. It is similarly structured as a linked list of *nodes* containing fixed-size arrays of *slots*, which are not reusable. Individual elements are enqueued in the list's *tail* node and dequeued from the *head* node in FIFO order. Whenever the tail node is full or all elements in the head node have been dequeued, the linked list structure is updated by appending a new node after the old tail or advancing the head node to the old head's successor node. The following paragraph lays out some of the terminology used to describe the processes involved in the queue's algorithm.

All enqueue operations attempt to write elements into a uniquely reserved slot in the current tail node and all dequeue operations attempt to *consume* (read and invalidate) an element from a likewise reserved slot in the current head node. For each slot there can be at most one enqueue and one corresponding dequeue operation. It is possible – although unlikely – for a concurrent dequeue operation to outpace the corresponding enqueue operation and attempt to read from a slot before it is initialized. When this occurs, both operations have to *abandon* the slot and retry. Once all slots have been either written to first and then consumed *or* abandoned, a node is considered to be *drained* and will be unlinked from the list. When a drained node is no longer in neither the head nor the tail position (i.e., has been unlinked) and all operations that may have attempted to advance either pointer from this node have concluded, it can be safely reclaimed and immediately de-allocated.

3.1 Data Structures

In addition to the head and tail pointers, the queue has to manage two additional unsigned integer values: an *enqueue index* and a *dequeue index*. Each index is exclusively incremented by its associated operation using atomic *read-modify-write* instructions. By incrementing the respective index in this manner, the acting thread uniquely reserves the *previous* index for itself, which may be used to address a slot in the respective node's array. Both node pointers are *tagged* with their associated index value, i.e., they store both the address to their respective node as well as the current index value in the same memory word. This allows both values to be updated atomically without resorting to a DCAS, but requires a sufficient number of available bits that are not used to represent the nodes' addresses themselves.[1] As an optimization, the queue also maintains a cached and only lightly contested copy of the current tail pointer's value. This makes it possible to inexpensively check, if a queue is empty in many cases.

Listing 1. Data Type Definitions for Queue, Nodes and Control Blocks

```
type TagPtr = (Node* ptr, u16 idx)

struct Queue:
  Atomic<TagPtr> head
  Atomic<TagPtr> tail
  Atomic<Node*> curr_tail

struct Node:
..[Atomic<void*>; N] slots
..Atomic<Node*>   next
..ControlBlock   ctrl

struct ControlBlock:
..Atomic<(u16, u16)> head_mask
..Atomic<(u16, u16)> tail_mask
..Atomic<u8>    reclaim
```

For all examples in this paper we use an array size of $N = 1024$, which means the index value can be represented with 10 bits. However, the index value is required to be able to grow beyond N to also accommodate operations that are unable to reserve a valid array index and subsequently have to help updating the linked list structure. Having incremented one of the index values is a strict prerequisite for dereferencing a pointer to the head or tail node, since only these operations are accounted for by the reclamation mechanism (similar to *reference counting*). Consequently, we require additional bits in both tagged pointers as a safety margin for preventing tag overflows. We show in Section 5.2 that six additional bits, or a total of $B = 16$ bits, are sufficient

1. The availability of such bits can be guaranteed by, e.g., requiring a specific memory alignment for each allocated node. Alternatively, most current 64-bit architectures (except very recent Intel designs) use 48-bit virtual addresses, meaning the upper 16 bits are always free for storing tags.

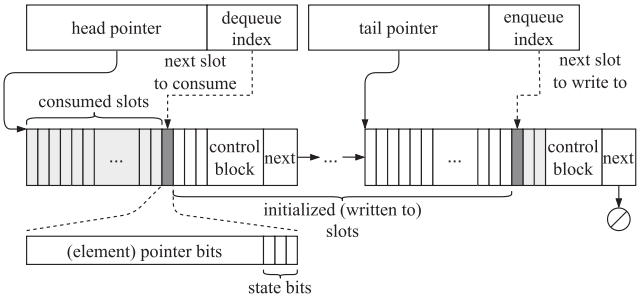


Fig. 1. Data structure and node layout.

to rule out overflows in any practical setting. Additionally, we also reserve and require three bits in each individual slot (and consequently also in each enqueued element pointer) for storing additional state information. Technically, two bits would be sufficient for representing the state but the description of the algorithm can be slightly simplified with three. Listing 1 shows the structure for individual list nodes and their associated control block structure, which is used for memory reclamation purposes. In addition, Fig. 1 shows a visual representation of that same structure.

3.2 Overview

Fundamentally, both enqueue and dequeue operations attempt to exclusively reserve access to a slot in the array of their associated queue node by atomically incrementing the appropriate index value and retrieving the previous value of the index as well as the current node pointer. Threads that retrieve an index $i < N$ gain the *exclusive* right to perform either a write or a consume operation on the corresponding slot. This guarantees that there can only be exactly one of each for any given slot. Operations that retrieve an index $i < N$ are *fast path* operations, which are specifically designed to finish as quickly as possible and only have to contend with memory reclamation in rare edge cases. All other operations have to enter a separate *slow path*, which requires them to first help updating the linked list structure before retrying and entering the fast path in the next attempt.

We use the atomic FAA primitive both for incrementing the index values as well as for performing read (consume) and write operations on the reserved slots. In the former case, using FAA drastically improves the scalability of all operations compared to using a CAS loop, especially on x86-64, where FAA is a single, infallible instruction [28]. In the latter case, the FAA is used to set specific bits in the reserved slot, which is possible since access to each slot is restricted to only one writer and one reader, so the arithmetic *add* is used in place of a bitwise *or*.[2] Note that all operations on slots also modify the slot's state bits, which announces both the operation's completion (in case of a read) and also makes determining the order, in which two operations occurred, possible. Since the index value is stored in the same memory word as its associated node pointer, the FAA operation could potentially affect both values, if too many increments were to occur. An overflow of the reserved tag bits would silently corrupt the pointer and must hence be prevented under any circumstances. In Section 5.2 we prove this to be impossible, except when extraordinarily large numbers of threads are involved.

2. To give an example, the operation $0 + x$ is identical to $0 \vee x$, if the latter is executed only once.

3.3 The Enqueue Procedure

The pseudo-code for the enqueue procedure and its slow path sub-procedure `adv_tail` can be found in Listings 2 and 3. After incrementing the enqueue index, fast path operations proceed to execute a single FAA operation on the reserved slot in the tail node's array. Since every slot is initially zeroed and will only ever be accessed by exactly one enqueue and at most one dequeue operation, this will effectively only set the bits constituting the inserted element, essentially making it a logical *or*. If, however, the enqueueing thread assesses that the `READER` bit was already set *before* the FAA was executed (line E8), the corresponding dequeue operation must have arrived too early. The slot is consequently abandoned and the thread has to retry. Checking for presence of `RESUME` bit in lines E8 and E9 only pertains to the memory reclamation mechanism and is only relevant in rare edge cases, in which an enqueue operation is significantly delayed and lags behind most other operations on the same node. This aspect will be addressed in more detail in Section 4.

Listing 2. Pseudo-Code for Enqueue Procedure (Fast Path) and Element Slot Constants

```
const RESUME = 0b001
const WRITER = 0b010
const READER = 0b100

enum AdvTail: AdvAndInserted, AdvOnly

# method of Queue
  E1 fn enqueue(void* el) -> void:
  E2 loop:
  E3 ..(t,i) = tail.fetch_add(1)
  E4 ..if i < N:
  E5 ....w = (u64) el | WRITER
  E6 ....prev = t->slots[i].fetch_add(w)
  E7 ....if prev <= RESUME: return
  E8 ....if prev == (READER | RESUME):
  E9 ......t->try_reclaim(i+1)
E10 ....continue
E11 ..else:
E12 ....switch (adv_tail(el, t)):
E13 ......case AdvAndInserted: return
E14 ......case AdvOnly: continue
```

The slow path procedure is, in essence, a modified version of the Michael-Scott algorithm [7], with which we assume the reader is sufficiently familiar and refer to for further insight. In summary, the procedure attempts to allocate and append a new node after the previous tail node. First, the current tail's `next` pointer is updated and only then the queue's tail pointer and enqueue index pair. If the first CAS fails, another thread has already appended a new node, in which case the already allocated node must be freed again and the thread has to help with updating the tail pointer. Note, that the thread that succeeds in updating

the previous tail node's next pointer automatically succeeds in inserting its element as well, since the newly allocated node contains that element in its first slot (line T8). Consequently, the second CAS, which updates the queue's tail pointer/index pair, must attempt to set the *enqueue index* to 1 instead of 0. The two bounded CAS loops in lines T11 and T19 are required, because competing FAA operations on the tagged tail pointer (see line E3) could potentially interfere, causing all slow path CAS to fail, otherwise. The loops guarantee, that, eventually, the tail pointer is updated by *some* thread before the slow path procedure concludes. Finally, the cached tail pointer will be updated, as well (see line T23). Before any call to the slow path procedure returns, it must invariably call the inc_enq_cnt function, which serves to count all finished slow path operations and updates the (old) tail node's control block accordingly. When called with a non-zero argument (see lines T14 and T22), it will also set the final number of all slow path operations, that need to be accounted for before the node can be reclaimed. This mechanism is explained in more detail in Section 4.

Listing 3. Pseudo-Code for the adv_tail Sub-Procedure (i.e., the Enqueue Slow Path)

```
# method of Queue
  T1 fn adv_tail(void* el, Node* t) -> AdvTail:
  T2 (res, final) = loop:
  T3 ..curr = tail.load()
  T4 ..if t != curr.ptr:
  T5 ....break (AdvOnly,0)
  T6 ..next = t->next.load()
  T7 ..if next == NULL:
  T8 ....node = alloc_node(el)
  T9 ....if t->next.cas(NULL,node):
T10 ......next = node
T11 ......while !tail.cas(curr, (node,1)):
T12 ........if curr.ptr != t:
T13 ..........break (AdvAndInserted,0)
T14 ......break (AdvAndInserted,curr.idx-N)
T15 ....else:
T16 ......dealloc_node(node)
T17 ......continue
T18 ..else:
T19 ....while !tail.cas(curr, (next,1)):
T20 ......if curr.ptr != t:
T21 ........break (AdvOnly,0)
T22 ....break (AdvOnly,curr.idx-N)
T23 curr_tail.cas(tail,next)
T24 t->inc_enq_cnt(final)
T25 return res
```

3.4 The Dequeue Procedure

For the dequeue algorithm's pseudo-code see Listings 5 and 6. Before incrementing the dequeue index, an initial check must be performed to determine if the queue is empty (Listing 4). This empty check is structured in a way, that avoids reading the potentially highly contested tail/enqueue index pair if possible and instead defers to the cached tail pointer value. Checking the tail is only required, if head and tail point at the same node and both indices have to be compared in order to assess if the queue is in fact empty. If the tail *does* have to be read, the check also helps updating the cached tail, if it is found to be lagging behind. Checking for empty queues helps to avoid unnecessarily incrementing the dequeue index, which is also important for bounding the total number of increments (see Section 5.2). Note, that it is not possible to assess the head node's next pointer before incrementing the dequeue index due to the unique constraints of the reclamation mechanism. After determining the queue to be *currently* not empty (which may change in between the check and the subsequent FAA) and incrementing the index, each operation likewise enters either into the fast path or the slow path.

Listing 4. Pseudo-Code for the Empty Check

```
# method of Queue
  I1 fn is_empty() -> bool
  I2 (head,di) = head.load()
  I3 curr = curr_tail.load()
  I4 if head == curr:
  I5 ..[tail,ei] = tail.load()
  I6 ..if curr != tail:
  I7 ....curr_tail.cas(curr, tail)
  I8 ..if head == tail && (di >= N || ei <= di):
  I9 ....return true
I10 return false
```

Listing 5. Pseudo-Code for the Dequeue Procedure (Fast Path)

```
enum AdvHead: QueueEmpty, Advanced

  # method of Queue
  D1 fn dequeue() -> void*:
  D2 loop:
  D3 ..if is_empty():
  D4 ....return NULL
  D5 ..curr = head.fetch_add(1)
  D6 ..(h,i) = curr
  D7 ..if i < N:
  D8 ....prev = h->slots[i].fetch_add(READER)
  D9 ....if i == N-1:
D10 ......h->try_reclaim(0)
D11 ....if prev & WRITER != 0:
D12 ......if prev & RESUME != 0:
D13 ........h->try_reclaim(i+1)
D14 ......return (void*) (prev & PTR_MASK)
D15 ....continue
D16 ..else:
D17 ....switch (adv_head(curr,h)):
D18 ......case Advanced: continue
D19 ......case QueueEmpty: return NULL
```

The fast path procedure is structurally similar to that of the enqueue algorithm: A dequeueing thread atomically sets the READER bit in the reserved slot and retrieves the previous state using a single FAA operation. Based on the assessment of that state, the operation is either determined to have been successful and the dequeued element is returned, or the slot has to be abandoned. One noteworthy difference lies in lines D9 and D10, which instruct the

dequeue operation on the *last* slot in a node to initiate the reclamation procedure `try_reclaim`. If the `WRITER` bit is indeed set, then the upper bits must contain a valid pointer to an enqueued element that can be returned (see also line E5).

Listing 6. Pseudo-Code for the Dequeue Procedure (Slow Path)

```
# method of Queue
  H1 fn adv_head(TagPtr curr, Node* h) -> AdvHead
  H2 if h == tail.load().ptr:
  H3 ..h->inc_deq_cnt()
  H4 ..return QueueEmpty
  H5 next = h->next.load()
  H6 curr.idx += 1
  H7 while !head.cas(curr, (next,0)):
  H8 ..if curr.ptr != h:
  H9 ....h->inc_deq_cnt()
H10 ....return Advanced
H11 h->inc_deq_cnt(curr.idx-N)
H12 return Advanced
```

The slow path procedure simply attempts to advance the current head pointer to its successor node, if there is one. Otherwise, the queue must be empty and `NULL` is returned. Since the operation for advancing the tail node *always* appends the node first and only *then* updates the tail pointer, a second empty check is required in line H2 to prevent the CAS in line H7 to advance the head pointer *ahead* of the tail pointer. As before, no slow path operation is able to exit the procedure before the head pointer is updated and the dequeue index is reset, with the exception of those operations, which determine the queue to be empty at line H2. This property is likewise crucial for bounding the number of possible operations as discussed in Section 5.2.

4 Memory Management

The mechanism for determining when a drained node can be safely reclaimed is based on the following three observations: A node must only be reclaimed, when all operations that may dereference a pointer to it have concluded. All operations that *can* dereference a node pointer are required by the algorithm to first increment their associated index value. Therefore, the index value serves a second purpose as an *operations* or *reference* counter. Once a node is unlinked from the list and the index value is reset, no further operations can observe the previous value and thus the total number of all operations that *had* observed the previous value and have dereferenced or may yet dereference the pointer, is known. Given these observations, it is evident that all that is required, is for every operation to check, if these conditions are met (i.e., if the current operation is the final one) and reclaim the node if this can be answered in the affirmative.

Since it is unlikely for any fast path operation to meet these criteria and since checking the associated conditions adds undesirable and non-trivial overhead, a different approach is used for fast path operations, which is realized through the `try_reclaim` procedure outlined in Listing 7. The procedure is always initiated by the dequeue operation attempting to consume the *last* slot in the current head node (see lines D9 and D10 in Listing 5). It incrementally checks all slots in the node's array for presence of the `WRITER` and `READER` bits. Once both bits have been set, it is guaranteed that there can be no more pending operations for the respective slot. In the unlikely – yet possible – case that a slot has not yet been visited by an enqueue or a dequeue operation (or both) at the time it is checked, the procedure aborts after atomically setting the `RESUME` bit in the slot (see line R5). Whichever pending operation arrives *last*, detects this bit and *resumes* the procedure from the next slot onward (see lines E8 and D9). This guarantees that all (fast path) enqueue and dequeue operations must have concluded, once all slots have been successfully checked. It is impossible for two threads to execute this procedure concurrently and each slot is only checked exactly once. The final lines R9 and R10 determine, whether the other conditions for reclamation are met and deallocate the node only if this the case. Note that we assume the functions `alloc_node` and `dealloc_node` to be able to dynamically allocate/de-allocate a node in a lock-free manner. This can be realized by implementing these using, e.g., thread-local free lists or, alternatively, with general lock-free memory allocators such as presented in [29] or [30].

Listing 7. Pseudo-Code for the Incremental Checking of All Slots in Order to Determine When All Fast Path Operations Have Concluded

```
const SLOT = 0b001
const DEQ = 0b010
const ENQ = 0b100

const CONSUMED = READER | WRITER

  # method of Node
  R1 fn try_reclaim(u16 start) -> void:
  R2 for i in (start..N):
  R3 ..s = slots[i]
  R4 ..if (s.load() & CONSUMED) != CONSUMED:
  R5 ....prev = s.fetch_add(RESUME) & CONSUMED
  R6 ....if prev != CONSUMED:
  R7 ......return
  R8 flags = ctrl.reclaim.fetch_add(SLOT)
  R9 if flags == (ENQ | DEQ):
R10 ..dealloc_node(this)
```

4.1 Slow Path Reference Counting

Every slow path operation must call either `inc_enq_cnt` or `inc_deq_cnt` immediately before concluding or retrying. These procedures update the `head_mask` or `tail_mask` tuple fields in the respective node's control block structure, using a single FAA each. Listing 8 shows the pseudo-code for `inc_enq_cnt` in an exemplary manner. The corresponding `inc_deq_cnt` function is structurally identical and is therefore omitted here. The first tuple element (i.e., the higher 16 bits) is initially zeroed and *eventually* stores the observed *final* count of finished operations. The lower half stores the *current* count and is incremented by every concluding slow path operation (lines C2 to C4). Whenever the linked list structure is updated, there is exactly one CAS that succeeds in updating the head or tail

pointers and their associated index values (lines T11, T19 and H7). These determine the final count of operations for the previous node and therefore their threads also have to (atomically) set the higher 16 bit (lines C5 to C7). When both halves are observed to be equal (line C9), i.e., when the total count has been set *and* the current count (after being incremented) matches its value, it can be concluded, that all considered operations must have finished and are therefore accounted for. Finally, a similar check as in lines R9 and R10 in Listing 7 is performed to determine, whether all other required conditions are met (i.e., if all three bits in the `reclaim` mask are set). This state can only be observed by the final operation to still hold a pointer to a node. Consequently, the *last one out* proceeds to reclaim the node.

Listing 8. Pseudo-Code for Increasing the Count of Completed Slow Path Enqueue Operations for a Node and Potential Reclamation

```
# method of Node
  C1 fn inc_enq_cnt(u16 final_cnt = 0) -> void:
  C2 if final_cnt == 0:
  C3 ..m = ctrl.tail_mask.fetch_add(1)
  C4 ..final_cnt = m >> 16
  C5 else:
  C6 ..v = 1 + (final_cnt << 16)
  C7 ..m = ctrl.tail_mask.fetch_add(v)
  C8 curr_cnt = 1 + (m & 0xFFFF)
  C9 if curr_cnt == final_cnt:
C10 ..prev = reclaim.fetch_add(ENQ)
C11 ..if prev == (DEQ | SLOTS):
C12 ....dealloc_node(this)
```

4.2 Comparison With Other Reclamation Mechanisms

The primary advantage of hazard pointers is their very low bound on memory usage in comparison to most other reclamation mechanisms, with the exception of lock-free (atomic) reference counting [23]. This advantage comes at the cost of substantially higher performance overhead for certain types of data structures, however (see Section 2.1). On the other hand, many mechanisms with smaller runtime cost, tend to exhibit substantially worse memory usage characteristics (to varying degrees) with increasing numbers of threads [20], [21], [22], [23]. Our own mechanism is not subjected to this same trade-off: It allows highly efficient memory usage (comparable with reference counting), while also requiring fewer or at most as many memory fences as epoch-based approaches or hazard eras. Although these can be specifically calibrated or fine-tuned for certain scenarios or use-cases to use less than one memory fence per data structure operation, this commonly comes at a loss of generality, greater API complexity and/or higher memory usage due to delayed reclamation.

In the following paragraph, we will outline how our mechanism is able to achieve a memory bound equal to or even lower than, e.g., hazard pointers: With the latter, memory reclamation is triggered by specific *reclamation events*. Typically, such opportunities arise, whenever an unlinked object is *retired* and the object is added to a global or thread-local garbage list. Assuming the smallest possible *reclamation interval* (often designated R) of 1, every time an object is retired, a reclamation attempt is made. In the course of such an attempt, *all* hazard pointers must be scanned and cross referenced with every retired object. Any object that is no longer protected by any hazard pointer is de-allocated. Consequently, the final operation to release its hazard pointer to a retired object does *not* generally trigger a reclamation event, unless it is, coincidentally, also the operation retiring the object. Therefore, the object remains in the garbage list at least until the next reclamation attempt, even though it could already be freed. It is principally possible to instead initiate a reclamation attempt whenever a hazard pointer is released, but since *every* operation generally has to release at least one hazard pointer, the resulting performance overhead is typically deemed not worth the trade-off. In our algorithm, on the other hand, every data structure operation is effectively able to trigger a reclamation event at very minor cost in general. Hence, objects that are no longer referenced by any thread are not retired in the usual sense, but instead de-allocated immediately.

Note, that we assign a special role to exactly one thread for each node, i.e., the thread initiating the `try_reclaim` procedure (see line D10 in Listing 5). This is somewhat similar to the reclamation mechanism used by the Yang/Mellor-Crummey [16] algorithm, which hands responsibility for reclaiming all reclaimable nodes to exactly one thread in a mutually exclusive manner and is therefore not lock-free [17]. With our mechanism, however, the thread responsible for initiating `try_reclaim` failing would *not* result in unbounded memory usage and only a single node would not be reclaimed. Furthermore, this special role is not particularly relevant for the reclamation mechanism, since *any* fast or slow path operation failing to conclude in an observable manner would prevent the accessed node from being reclaimed. This is no different from, e.g., hazard pointers or most other mechanisms, however, where a thread failing to release a hazard pointer would also prevent the pointed-to object from being reclaimed.

We can construct a hypothetical scenario, in which exactly one thread fails to conclude its operation on one distinct node each, so that, effectively, no node can ever be successfully reclaimed. Note that failing to reclaim a node does not prevent progress on the data structure itself, as the two are fully decoupled. Evidently, a thread failing at one point will be entirely unable to continue making any further progress. Therefore, once all threads have failed, all progress comes to a halt and no further nodes will be allocated. Hence, the theoretical upper bound on the maximum number of unreclaimed nodes is equal to the number of participating threads, which matches the lowest possible memory bound of hazard pointers, when each thread uses only one hazard pointer.

5 Correctness

Our presented algorithm is both lock-free and linearizable, which we prove in this section. We further prove, that it has strong safety guarantees regarding the impossibility of an overflow of tag bits due to infallible FAA operations for sufficiently large numbers of threads. The number of threads for which these properties can be guaranteed depends only the number of slots in a node's array (N) and the number of reserved tag bits (B). We show that with reasonable values,

this bound is so high as to not present a limitation for any practical applications.

5.1 Linearizability

Informally, *linearizability* requires for every data structure operation to appear to take effect instantaneously to an external observer at some specific point in between being invoked and concluding [31]. The linearizability of concurrent data structures is often shown by identifying the *linearization point* (LP) of each procedure, at which an atomic and externally observable state change occurs in accordance to the high-level sequential specification of the respective data structure [32]. Morrison and Afek have conclusively shown the linearizability of their LCRQ algorithm with regard to a high-level FIFO queue specification. Due to the significant structural similarities of their algorithm with practically all other array-node based FIFO queues including our own, their proof strategy can be easily adopted with minor adjustments, and so we omit a full replication of the proof here.

Like Morrison and Afek, we linearize every fast path enqueue $E_F(v)$ at the FAA incrementing the enqueue index at line E3 (Listing 2) *only if* the subsequent FAA at line E6 is successful, i.e., the corresponding dequeue had not invalidated the slot by the time the FAA was executed. In the slow path, we linearize an enqueue operation $E_S(v)$ by a thread T as soon as it has succeeded in updating the current tail's `next` pointer with the CAS at T9 *and* the queue's `tail` pointer has been updated as well. The first event implies that T had previously succeeded in conclusively and observably installing a new node with v inserted in its first slot. The second event occurs either when T subsequently executes a successful CAS in the loop at line T11 or when another thread succeeds in updating the tail to the node installed by T with the CAS at line T19. In the former case, linearization occurs trivially within T's execution interval of $E_S(v)$. In the latter case, it *must* also occur within $E_S(v)$'s execution interval, as otherwise T itself would succeed with some CAS in the loop at line T11, so by the time T executes the final CAS of that loop, the operation was definitely linearized. The FIFO order of fast path enqueues is guaranteed by the FAA synchronization primitive, so we only have to show, that linearized slow path enqueues linearize in the same order:

Lemma 5.1.1. *When a new tail node is appended to the queue containing element v, the responsible slow path enqueue $E_S(v)$ is linearized (1)* after *the last successful enqueue in the previous tail node and (2)* before *the first successful (fast path) enqueue in the newly appended node.*

Proof. The final successful (linearized) fast path enqueue on any given node will retrieve an enqueue index $i < N$, whereas all slow path enqueues retrieve $i \geq N$, so their FAA must necessarily execute after all successful fast path enqueues. Hence, (1) holds, since all successful fast path enqueues linearize at their FAA of the enqueue index. On the other hand, (2) must hold as well, since fast path enqueues in the next tail node are not possible until *after* the new node is also installed in the queue's tail position by a successful CAS in either line T11 or T19, at which point $E_S(v)$ is linearized. □

For the dequeue procedure, we linearize operations $D() : v$ with $v \neq NULL$ differently from those returning $NULL$. In case of a fast path dequeue, we linearize an operation $D() : NULL$ either at line I3 or at line I5 (Listing 4) at which we identify an empty queue. In the slow path, we linearize an operation returning $NULL$ at line H2, which represents the second opportunity for determining the queue to be empty. It remains to be shown, that dequeue operations which do not return $NULL$ are linearized as well and that a linearization point for $D_i() : v$, which retrieves v from the slot at index i, can always be found at or after the LP of the corresponding $E_i(v)$, thereby adhering to the queue's high level FIFO specification. This is more difficult to show, since a $D_i() : v$ may in fact start and even increment the dequeue index *before* a corresponding $E_i(v)$ and still turn out successful, eventually. For the exact proof, we refer to Morrison and Afek's paper on LCRQ and specifically their Lemma 1. Their proof method of utilizing an infinite auxillary queue, a hypothetical event loop of all sequential events and induction on the number of linearized enqueues can be also applied to our algorithm due to the structural similarities of both queues. Notice, for instance, that the proof for the cited lemma is able to reason about the linearizability of dequeue operations primarily based on the fact, that all operations perform an initial FAA on an index. The full proof is fairly verbose, however, and little would be gained from replicating it here in detail. Hence, we merely provide the following extension adapting it for the specific state transitions of our own algorithm:

Lemma 5.1.2. *Suppose an execution linearizes $E_i(v)$. If there exists a dequeue D, whose FAA on the dequeue index returns i for the same node as $E_i(v)$, then D is $D_i() : v$, i.e., it succeeds and returns v.*

Proof. Since $E_i(v)$ is linearized, its second FAA (line E6) writes v into the slot at index i and does assess no `READER` bit to be present (otherwise it would not linearize). Hence, the FAA by D in line D8 must have occurred *after* the corresponding FAA by $E_i(v)$ and therefore allow D to consume v from the slot and subsequently return it. □

To summarize briefly, the proof by Morrison and Afek shows that every dequeue is linearized as soon as it is invoked (i.e., no earlier), the increment of the index of the corresponding enqueue has occurred and all dequeue operations on previous slots have linearized. Using this proof as a base and our adaptations to it allows us to conclude, that all successful dequeue operations are also linearized and that their linearization order matches the queue's high level FIFO specification.

5.2 Overflow Protection Guarantees

Both enqueue and dequeue operations need to increment their associated index value, which is stored alongside the pointer to the head or tail node, respectively. In order to grant these operations better scalability under contention, the respective indices are incremented using the infallible atomic FAA primitive. By analyzing the possible paths an operation can take and given the two constants N (the array size) and B (the number of available tag bits), it is possible to reason about the maximum extent an index value for any

distinct $(pointer, index)$ pair can take on before being advanced and reset depending on the total number of preemptable threads accessing the same queue. We thereby prove that an overflow of the index bits into the pointer bits is impossible as long as the number of threads concurrently accessing the same queue stays below a precise threshold.

Definition 5.2.1. *Let P be number of all producer threads accessing the same queue and $2^B - 1$ the largest possible integer value that fits into the tag bits of the composed $(tail, index)$ tuple without overflowing.*

Lemma 5.2.1. *The largest value the enqueue index can possibly assume is $P + N$ and it follows, that with any $P \leq 2^B - N - 1$ the enqueue index can never overflow.*

Proof. Assume that the current value of the enqueue index is N, so there must have been N previous increments which resulted in fast path operations, which can unconditionally enqueue another element or retry after concluding (e.g., due to having to abandon the reserved slot). Subsequently, there can hence be at most P further concurrent enqueue attempts (one per producer thread), all of which must necessarily enter the slow path. Within that path there are four sub-paths, none of which permit a thread to exit it and then potentially observe and increment the previous $(tail, index)$ pair again. The early-exit path beginning at line T4 can only be taken, if the tail pointer has already been updated, in which case the enqueue index has also been reset and can not be incremented again. The paths at lines T11 and T19 can likewise only result in an executing thread leaving the procedure if one of the CAS at either line has succeeded in updating the $(tail, index)$ pair. The path at line T15 can never leave the procedure at all and can also be taken at most once per thread. In total, the index value can thus never exceed $P + N$. By equating this hard upper bound with the largest possible index value $(2^B - 1)$, we can conclude that with $P \leq 2^B - N - 1$ distinct producer threads the enqueue index can never overflow. □

The path analysis for the dequeue procedure is somewhat more involved, since its outcome can be influenced by concurrent dequeue as well as enqueue operations, whereas enqueue operations are only affected by other enqueue operations. Additionally, there is also the possibility for a thread to return early from the procedure (and potentially try again an unbounded number of times), before incrementing the dequeue index.

Definition 5.2.2. *Let C be number of all consumer threads accessing the same queue and $2^B - 1$ the largest possible integer value that fits into the tag bits of the composed $(head, index)$ pair without overflowing.*

Lemma 5.2.2. *The largest possible value the dequeue index can possibly assume is $2 \cdot C + N - 1$ and it follows, that with any $C \leq \frac{2^B - N - 1}{2}$ the dequeue index can never overflow.*

Proof. Assume that the current value of the dequeue index is $N - 1$, which means that there must have been $N - 1$ previous increments that resulted in fast path operations, i.e., those that may subsequently attempt to dequeue another element or retry after concluding. Additionally, the queue's head and tail nodes must currently point at the same node and the enqueue index is some value $E \geq N$, i.e., all slots have already been either written to or abandoned. Under any other conditions, the dequeue index can only grow up to $C + N$, same as with the enqueue algorithm.

Further assume that there are exactly C subsequent concurrent dequeue attempts, all of which pass the empty check at line D3 before any of them increments the dequeue index. Otherwise, all follow-on operations would not pass that check and therefore could not increment the index. All C threads necessarily enter the slow path and perform the second empty check at line H2. Assuming that the tail pointer has not yet been updated, all C threads will assess the queue to be empty and return. The value of the dequeue index is now $C + N$, but all C consumers may yet initiate one more dequeue operation directed at the same $(head, index)$ pair. However, none of these may pass the first empty check again and thus can not increment the dequeue index *until* the queue's tail pointer has been updated by some slow path enqueue operation. Since the tail pointer is exclusively updated only *after* a new node has already been appended (see line T9), when any of the C additional operations are permitted to move past the empty check at line D3, it is guaranteed that the current head's `next` pointer is no longer `NULL`, the tail pointer has been updated and hence, none of these threads can ever return from the procedure at the second empty check at line H2 again for the same $(head, index)$ pair.

Consequently, all C threads will remain in the slow path's bounded CAS loop (line H7) until the queue's head pointer is updated and the dequeue index is reset, in which case the previous value can not be incremented any further. Therefore, the dequeue index can not exceed $2 \cdot C + N$. As before, we can conclude that with $C \leq \frac{2^B - N - 1}{2}$ consumer threads the dequeue index can never overflow. □

With the reasonable and sane default values $N = 1024$ and $B = 16$, as in our examples, overflows for either index value are impossible for 64,511 producer and 32,255 consumer threads. This is substantially more than what any practical application would require. Even with more threads than that, the precise circumstances required for an overflow to occur would be highly unlikely and are mostly theoretical in nature. Also, if it should ever become necessary, the threshold values for which these safety guarantees apply can be easily doubled for every additionally reserved tag bit, although finding space for further tag bits may not be as trivial. It is likewise possible to, e.g., reduce the number of tag bits to $B = 11$, which can be guaranteed by allocating each node aligned to a 2048 byte boundary. This would still suffice to guarantee safe access to a single queue instance for up to 1,023 producer and 511 consumer threads, which is still at least 5 times more than the maximum number of threads (96) we used during evaluation.

5.3 Lock-Freedom

For the fast path and in the general case, *all* threads can make progress by simply inserting into or consuming from their reserved array slot. However, this can not be guaranteed, since it is possible for a slot having to be abandoned and both corresponding operations having to retry. This

can, in fact, theoretically result in a temporary live lock situation, in which all threads continuously have to abandon one slot after another. However, this live-lock can last at most for N steps, after which the current node is drained and all dequeue operations have to conclude that the queue is empty, whereas at least one enqueue operation is guaranteed to be able to insert its element in the slow path. Hence, at least one thread is able to make progress in a bounded number of steps. This property is also exhibited by FAAArray [15] and LCRQ [12], but *not* by Nikolaev's SCQ [13] algorithm, which is designed to avoid such live-lock situations outright.

Threads entering the slow path, on the other hand, can not leave it again, before the respective $(pointer, index)$ pair has been updated. Therefore, threads may be required to stay inside the bounded CAS loop paths (lines T11, T19 and H7) in order to maintain this invariant. However, the number of steps required to make progress is bounded by the total number of producer/consumer threads: A thread in either of the bounded CAS loops can only be forced to remain in the loop if another thread interferes by incrementing the index value (lines E3 and D5). Since a thread in the slow path can not leave it and increment the same index value again, however, this results in a "soak-off" effect: The probability for further interfering FAA increments decreases and eventually reaches zero, once all producer or consumer threads have entered the slow path and are "trapped" in it, at which point progress by one of the threads in that path can be guaranteed. This progress will eventually be observed by the other trapped threads, allowing them to proceed as well.

6 Experimental Results

For the evaluation of our queue algorithm and its memory reclamation mechanism we chose a typical benchmark suite measuring the throughput of (a) bursts of consecutive enqueue operations, (b) bursts of consecutive dequeue operations, (c) pairwise alternating enqueue and dequeue operations and (d) through (f) randomized enqueue and dequeue operations with differing weightings towards one or the other. We evaluated an implementation of our own queue (LOO) against implementations of the Michael/Scott [7], Morrison/Afek [12] (LCRQ), Ramalhete/Correia [15] (FAA), Nikolaev [13] (LSCQ2 and LSCQD) and Yang/Mellor-Crummey [16] (YMC) queues. All except the latter use a custom implementation of hazard pointers [6] for safe memory reclamation. Hazard pointers are generally a good fit for array-based queues, since these only need to protect a single pointer per each operation, which rarely changes. For the YMC queue we use the same custom memory reclamation mechanism as it is outlined in the associated paper/reference implementation. Note that benchmarks (a) to (c) are conducted for thread counts ranging from one (single-threaded) to 96, whereas benchmarks (d) to (f) always use multiples of four threads, hence starting at four instead of one. Further note, that benchmark (f) uniquely uses a pre-filled queue with 75 percent of the total elements already present before the benchmark starts, so that the more prevalent dequeue operations never operate on an empty queue.

All benchmarks were conducted on a cluster node with 2 x Intel(R) Cascade Lake Platinum 9242 CPUs, each with 48 physical cores, for a total of 96 cores running under a 3.10 Linux kernel. All code was compiled with the GCC 10 C++ compiler and statically linked. We used *mimalloc* [33] instead of the standard system allocator, since the former is known to perform better for highly concurrent workloads. Note that GNU compilers (gcc and g++) prior to version 10 suffer from a missed optimization opportunity on x86 in the compilation of sequentially consistent stores, which have to be used, e.g., when acquiring hazard pointers, which may distort the results of these benchmarks.[3]

6.1 Throughput

The results of the aforementioned throughput benchmarks can be found in Fig. 2. The measurements for the throughput under single-threaded execution are deliberately truncated in order to provide more readable plots. As contention is introduced, total throughput for all queues drops precipitously at first, which is to be expected. Our own algorithm is able to perform best or among the best in every evaluated scenario. In benchmark (a) it is generally on par with FAA with an increasing margin in favour of LOO from 32 threads and onwards. In benchmark (b) the closest competitor is YMC, which generally performs slightly below our queue. The results of benchmark (c) are similar to (a), the only difference being that LCRQ has a notable advantage at 2 threads, which maybe due its property of allowing array slots to be re-used. In the randomized benchmark (d), which is sometimes also called half-half, LOO outperforms all other queues by a significant margin, except at 32 threads. In the enqueue-heavy randomized benchmark (e), FAA generally outperforms LOO, although this reverses at around 56 threads, whereas in the dequeue-heavy benchmark, our queue is again able to outperform all other queues. It appears, that the benefit of a customized reclamation mechanism over hazard pointers becomes truly apparent under extreme contention, as both our queue and YMC tend to increase their margins over the other queues especially from 48 threads onwards in all scenarios.

6.2 Memory Retention

In order to evaluate the presumed differences in memory retention, we used a similar benchmark suite as for the burst throughput benchmarks. Initially, a fixed number of elements is enqueued, so that at total of 10,000 nodes have to be allocated (both queues can store the same number of elements per node). As a result, the initial state represents the high water mark in terms of total allocated memory. Afterwards, all threads concurrently dequeue elements until the queue is empty. As individual elements are consumed, nodes become drained and unlinked from the linked list structure. Once a node is unlinked, it is conceptually retired and can be de-allocated when there are no longer any threads attempting to access the node. This is ensured by the respective reclamation mechanism at runtime. For every retired node, the current total allocated memory is queried from the allocator immediately *after* it is unlinked.

We only evaluate our queue and FAA for this benchmark, since the focus is on comparing the memory retention characteristics. Additionally, both the basic strategy and

3. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91719

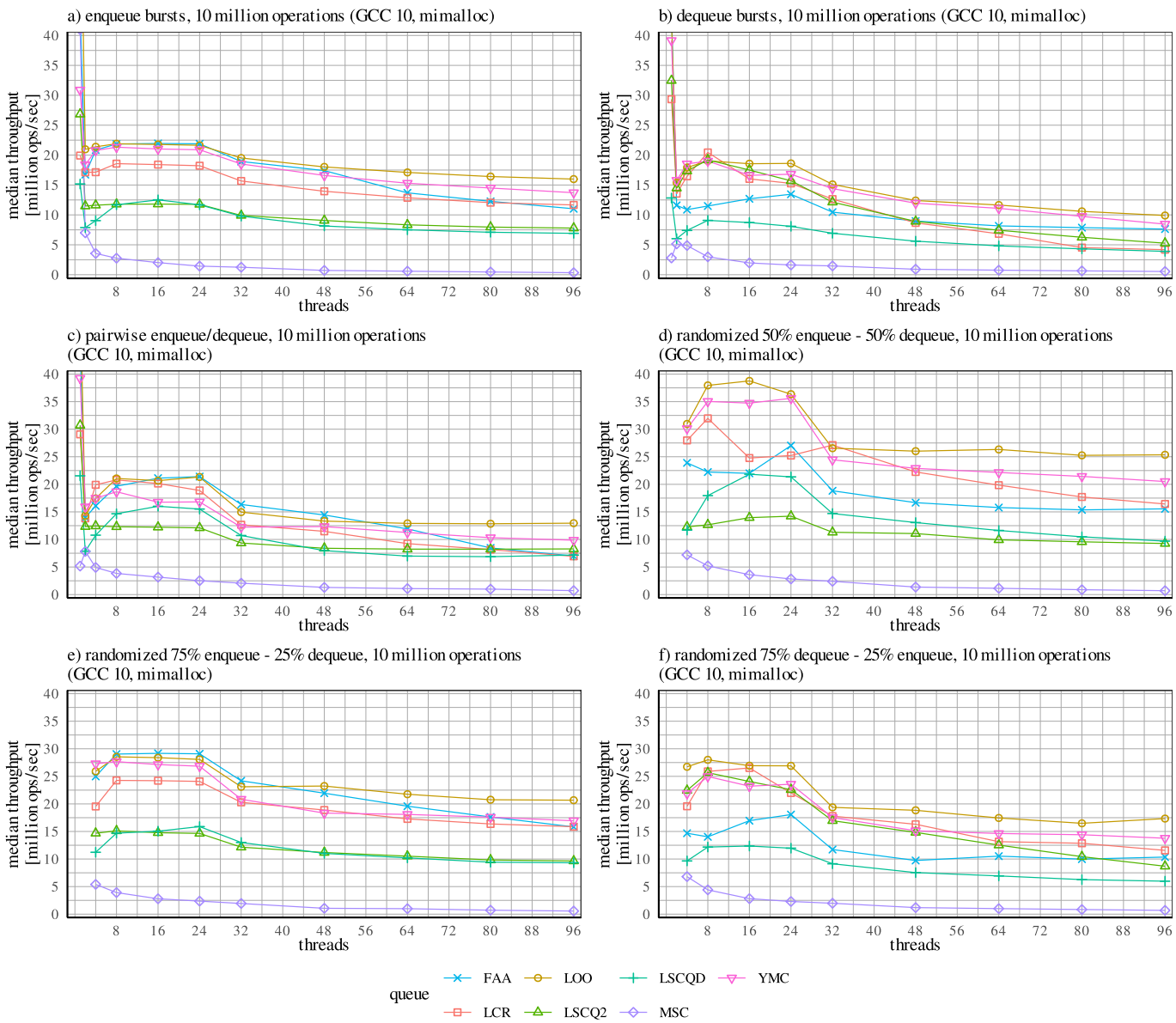
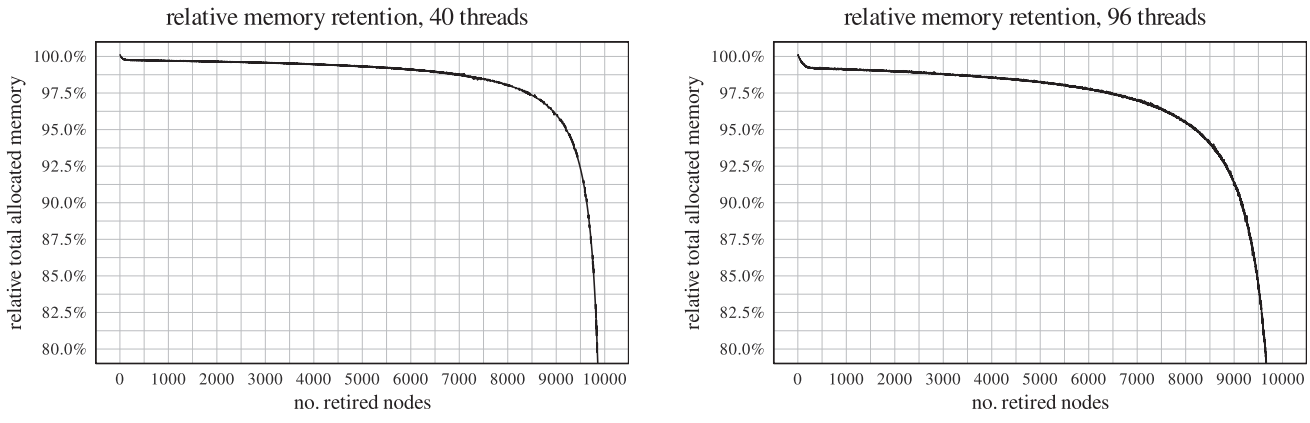


Fig. 2. Results of throughput benchmarks.



(a) Relative memory retention with 40 threads

(b) Relative memory retention with 96 threads

Fig. 3. Relative retention of retired nodes (memory) between LOO and FAA. The portrayed ratios are calculated as mem_{FAA}/mem_{LOO}.

node sizes for the same number of elements are similar for these two queues. The measurements for this benchmark are depicted in Fig. 3. Only two setups, one with 40 threads and one with 96 threads, were evaluated. As in the previous benchmarks, each thread was pinned to one physical CPU and no hyper-threads were used. The plots show the

relative difference in total allocated memory at each step between LOO and FAA. Note that due to the slight node size difference (FAA nodes are slightly smaller), the initial memory usage high water mark for FAA is lower (by ≈ 76 KiB). Overall, the effect of delayed memory de-allocation when using hazard pointers is fairly subtle, but nonetheless noticeable.

The initial rapid decrease in relative retained memory in either plot that occurs during retirement of the first ≈ 200 nodes can most likely be linked to the threads starting their dequeue bursts in slightly staggered fashion instead of in perfect unison. It can be observed that our queue retains less memory at each step and the advantage in memory retention is consistently increasing. The effect becomes more pronounced when more cores are involved. For the most part, this advantage translates to a relatively small difference in allocated memory (between 1 and 3 percent) and only eventually reaches up to 20 percent and finally 40 percent (not depicted) on the last nodes. This advantage can be directly attributed to the fact that our mechanism de-allocates each node at the earliest opportunity, whereas with hazard pointers memory can only freed during specific reclamation attempts.

7 Conclusion

In this paper we have presented an algorithm for an unbounded, lock-free MPMC FIFO queue. By avoiding reliance on a general-purpose memory reclamation mechanism and instead using a specialized one, custom tailored for this specific data structure, and its distinct properties, we are able to achieve performance that is on par with or better than other state-of-the-art queues, while also guaranteeing very low bounds on memory. The mechanism is similar to reference counting, but avoids most of its overhead and other undesirable properties by utilizing operations inherently required by the data structure itself (i.e., increasing the index values) for this purpose, thereby incurring no *additional* overhead. Consequently, we are generally able to avoid all overhead for memory management in fast path operations, which constitute $\approx 99\%$ of all operations under sensible settings. Traditional reclamation mechanisms, on the other hand, commonly require additional memory fences for every data structure operation. The memory reclamation mechanism designed for our queue is able to reliably de-allocate nodes immediately once the last accessing operation concludes. This has a measurable positive effect on memory retention when compared to a specifically fine-tuned and optimized hazard pointers implementation. We see potential for future work in examining and quantifying the actual relative advantage of our reclamation mechanism by benchmarking other queue algorithms using, e.g., fine-tuned hazard eras instead, which would likewise avoid many additional memory fences. Likewise, we see further research potential in examining the impact of different node sizes for the various array-based queues evaluated here.

Our queue has an additional and interesting, although not yet explored, potential special use case for implementing other reclamation mechanisms, e.g., for storing retired objects: These usually require other lock-free data structures for internal management purposes. Since practically all non-trivial lock-free data structures require memory reclamation of some sort, reclamation mechanisms usually resort to supporting, e.g., only fixed and pre-determined numbers of threads, which can be realized using only trivial data structures. It is often plainly impossible to use data structures that themselves have to rely on a reclamation mechanism for this purpose, so our queue could be a potential candidate for such use cases. The same is true for Nikolaev's SCQ algorithm, but only for cases where a bounded queue is sufficient, since the unbounded variant also relies on hazard pointers or, potentially ABA tags, which in turn would require DCAS again. The queue by Yang and Mellor-Crummey could be another candidate for this, since it also uses a custom node reclamation mechanism, although it is not lock-free.

Availability

A reference C++ implementation of the proposed queue algorithm can be found at https://github.com/oliver-giersch/lfqueue-benchmarks alongside implementations of all other evaluated queues and all benchmarking code.

Acknowledgments

This work was supported in part by the Federal Ministry of Education and Research (BMBF) of Germany under Grant 01IS18072 and in part by German Science Foundation (DFG) under Grant DFG NO 625/7-2. The authors would like to extend their special thanks to Ruslan Nikolaev for his helpful remarks on implementing the unbounded variants of SCQ2 and SCQD correctly and efficiently.

References

[1] H. Gao, J. F. Groote, and W. H. Hesselink, "Lock-free parallel and concurrent garbage collection by mark&sweep," *Sci. Comput. Program.*, vol. 64, no. 3, pp. 341–374, Feb. 2007.

[2] A. Gidenstam, P. Tsigas, and H. Sundell, "Practical and efficient lock-free garbage collection based on reference counting," Göteborg Univ., Gothenburg, Sweden, Tech. Rep. TR-2005–04, 2005.

[3] N. Cohen, "Every data structure deserves lock-free memory reclamation," in *Proc. ACM Program. Lang.*, 2018, pp. 143:1–143:24.

[4] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "StackTrack: An automated transactional approach to concurrent memory reclamation," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.

[5] D. Mathieu, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, Feb. 2012.

[6] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th Annu. ACM Symp. Princ. Distrib. Comput.*, 1996, pp. 267–275.

[8] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Int. Conf. Princ. Distrib. Syst.*, 2010, pp. 302–317.

[9] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. 22nd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 355–364.

[10] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank, "BQ: A lock-free queue with batching," in *Proc. 30th Symp. Parallelism Algorithms Archit.*, 2018, pp. 99–109.

[11] C. M. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable, lock-free k-FIFO queues," in *Proc. Parallel Comput. Technol.*, 2013, pp. 208–223.

[12] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 103–112.

[13] R. Nikolaev, "A scalable, portable, and memory-efficient lock-free FIFO queue," in *Proc. 33rd Int. Symp. Distrib. Comput.*, 2019, pp. 28:1–28:16.
[14] S. Arnautov, P. Felber, C. Fetzer, and B. Trach, "FFQ: A fast single-producer/multiple-consumer concurrent FIFO queue," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 907–916.
[15] P. Ramalhete and A. Correia, "FAAArrayQueue - MPMC lock-free queue," 2016. [Online]. Available: http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html
[16] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–13.
[17] P. Ramalhete, "Wait-free bounded vs wait-free unbounded," 2016. [Online]. Available: https://concurrencyfreaks.blogspot.com/2016/09/wait-free-bounded-vs-wait-free-unbounded.html
[18] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr., "Lock-free reference counting," in *Proc. 20th Annu. ACM Symp. Princ. Distrib. Comput.*, 2001, pp. 190–199.
[19] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 8, pp. 1173–1187, Aug. 2009.
[20] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, Univ. Cambridge Comput. Lab., Cambridge, U.K., 2004.
[21] T. A. Brown, "Reclaiming memory for lock-free data structures: There has to be a better way," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2015, pp. 261–270.
[22] H. Wen, J. Izraelevitz, W. Cai, A. H. Beadle, and M. L. Scott, "Interval-based memory reclamation," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 1–13.
[23] P. Ramalhete and A. Correia, "Brief announcement: Hazard eras - non-blocking memory reclamation," in *Proc. 29th ACM Symp. Parallelism Algorithms Archit.*, 2017, pp. 367–369.
[24] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Architectures*, 2002, pp. 73–82.
[25] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 317–328.
[26] J. Kang and J. Jung, "A marriage of pointer- and epoch-based reclamation," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 314–328.
[27] R. Nikolaev and B. Ravindran, "Hyaline: Fast and transparent lock-free memory reclamation," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 419–421.
[28] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 445–456.
[29] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2004, pp. 35–46.
[30] H. Sundell, "Wait-free reference counting and memory management," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp.*, 2005, p. 10.
[31] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[32] T. A. Henzinger, A. Sezgin, and V. Vafeiadis, "Aspect-oriented linearizability proofs," in *Proc. Int. Conf. Concurrency Theory*, 2013, pp. 242–256.
[33] D. Leijen, B. Zorn, and L. Moura, "Mimalloc: Free list sharding in action," Microsoft, Redmond, Was, USA, Tech. Rep. MSR-TR-2019–18, Jun. 2019. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2019/06/mimalloc-tr-v1.pdf

Oliver Giersch received the BSc degree in economics engineering and the MSc degree in business informatics from the Brandenburg University of Technology (BTU), Cottbus, in 2016 and 2019, respectively. Since 2019 he is currently working toward the PhD degree in computer science with the chair for distributed systems and operating systems, BTU. His research interests include concurrent and lock-free algorithms, and memory reclamation.

Jörg Nolte received the MSc (Dipl. Inform.) degree in computer science in 1988 and the PhD (Dr.-Ing.) degree in 1994, both from the Technical University of Berlin. He is currently a professor of computer science with the Brandenburg University of Technology (BTU), Cottbus, Germany, where he holds the chair for distributed systems and operating systems. Prior to that position he was a senior researcher with the Fraunhofer Gesellschaft, Institute for Computer Architecture and Software Technology, Berlin. He was a principal member and eventually the deputy head of PEACE group that developed the operating system for Germany's first massively parallel supercomputer. In the 90s, he was a postdoc fellow and senior researcher with Real World Computing Partnership, Tsukuba Research Center, Tsukuba Science City, Japan. Since then his research interests include scalable, low latency middleware and operating system platforms for clusters and other parallel architectures, including rather strange ones such as, wireless sensor networks. His major research interests include operating systems, middleware and programming languages for parallel, distributed and embedded systems. He is the member of board of special interest group for operating systems of the german GI and is currently the dean of Faculty 1 (mathematics, computer science, physics, electrical engineering and information technology) of the BTU.