A Simple and Scalable Wait-Free Ring Buffer

Andrew Tyler Barrington
University of Central Florida
abarrington@knights.ucf.edu

Steven D. Feldman
University of Central Florida
feldman@knights.ucf.edu

Dr. Damian Dechev
University of Central Florida
dechev@eecs.ucf.edu

Abstract

The ring buffer is a staple data structure in computer science applications. In contrast to linked-list and vector data structures, ring buffers use a constant amount of memory. This property makes it ideal for certain applications such as multimedia, network routing, trading systems, and gaming. In some of these cases, concurrent access is ideal or mandatory. In this paper, we present a new non-blocking ring buffer that provides a wait-free progress guarantee suitable for such applications.

Several non-blocking queue implementations exist in the literature, however, we are not aware of any that provides the wait-free guarantee. The design of efficient wait-free algorithms is challenging, because each thread operating on the data structure must complete its operation in a finite number of steps. This strict guarantee makes it ideal for real-time and mission critical systems, however, the added complexity to achieve this often results in reduced performance. revolutions in data memory (cite memsql)

Circumventing such pitfalls, the presented concurrent ring buffer uses a methodology for diffusing contention, which results in a significant increase in performance. Operating with 64 threads on a 64-core machine, performance results show that our algorithm performs on average X more operations than a coarse-grained locking approach, X more than TBB's concurrent bounded queue, and X more than the cycle queue by Tsigas.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords concurrent, non-blocking, wait-free, queue, ring buffer

1. Introduction

The ubiquitous use of ring buffers in computer applications make them a focal point in computer science research. A ring buffer or cyclical queue is a first-in-first-out queue that stores elements on a fixed length array. In contrast to linked list queues, which do not have a maximum capacity, ring buffers are limited in capacity by the length of this array. This array allows for efficient O(1) operations, cache-aware optimizations, and low memory overhead. In general, the memory utilization of a ring buffer is limited to the cost of the array and two counters, making it desirable for systems with limited memory resources, such as embedded systems.

Ring buffers are found in a wide range of applications such as network routing, multimedia processing, and cloud-based services. Many of these applications depend on ring buffers to pass work from one thread to another. With the rise in many-core architectures, the number of threads executing in a system has greatly increased. As a result, the efficiency of shared data structures, such as ring buffers, have a greater impact on performance than ever before.

In an attempt to achieve efficient and scalable performance, there has been significant research into the development of non-blocking algorithms. These non-blocking algorithms forgo the use of locks, to permit greater scalability and core utilization. Such designs are categorized by the level of progress they guarantee, with wait-free as the highest and most desirable categorization. It provides freedom from deadlock, livelock, and thread starvation. Deadlock is a result of multiple operations waiting on the other to finish, thus blocking all operations involved in the deadlock. Livelock is similar to deadlock except operations yield to each other causing a lack of progress from the threads repeatedly making attempts to allow the other operation to progress. Starvation occurs when an operation waits indefinitely for a resource [4]. In addition to wait-free, non-blocking designs can also be categorized as either lock-free or obstruction-free. In contrast to wait-free, lock-free designs are susceptible to thread starvation and obstruction-free designs are susceptible to both livelock and thread starvation [4].

We are aware of two other non-blocking ring buffers in literatue. Tsigas et al. presented a lock-free approach in which threads compete to apply an operation [**?**], however, this approach suffers from thread congestion and poor scaling. Krizhanovsky presented a non-blocking approach which improves scalability through the use of the *fetch and add* operation [**?**], but unfortunately the design is susceptible to thread starvation and is not thread death safe.

We propose a new wait-free ring buffer algorithm based on the atomic *fetch and add* (*FAA*) operation. A thread performing an enqueue or dequeue operation will perform a *FAA* on the tail or head sequence counter, respectively. The returned sequence identifier (*seqid*) is used to determine the position to enqueue or dequeue an element from the buffer. Specifically, the position is determined by the *seqid* modulo the buffer's length. To address the case where the *seqid* of two different threads refer to the same position, we employ a novel method of determining which thread is assigned the position and which thread must get a new position. To prevent the case where a thread is continually prevented from completing its operation, we integrated a progress assurance scheme [1].

The thread performing a dequeue operation will attempt to replace an element node whose *seqid* matches its assigned *seqid* with a newly allocated empty node. This empty node's *seqid* is equal to the current *seqid* plus the length of the buffer. Our restriction on the type of node a dequeue thread may remove, allows us to derive a

FIFO[1] ordering of elements. Without this restriction it is *theoretically* possible for a thread to enqueue A and B and then dequeue B followed by A, breaking the FIFO property. We explain the specific scenario which may lead to this in Sec. **??**. However, this restriction also introduces the scenario where if a thread never dequeues its node, then other threads with higher sequence numbers must wait. If this scenario is detected, an atomic bitmark is placed on that value. By using this bitmark, we can allow a dequeuer to get a new *seqid* without the risk of an element being enqueued with the old *seqid*.

Enqueueing threads will attempt to replace the current value with an element node, if the current value is an empty node whose *seqid* is equal to the thread's *seqid*. If the current value has a *seqid* less than the thread's *seqid*, the thread will perform a back off routine to provide a delayed thread the opportunity to complete its operation. If the current value has not changed, then depending on the node type (element or empty) and the *seqid* (less than or greater), the thread will either get a new *seqid* or replace the current node with its element node. We explore the various states and their associated actions in Sec. **??**.

Previous research has shown that the use of the *FAA* operation can provide significant increase in performance, when compared to similar designs implemented with the atomic compare-and-swap operation (*CAS*). For example, Feldman et al. [1] compared the performance a *FAA* based vector pushBack operation and a *CAS* based approach and found that the *FAA* design outperforms the *CAS* design by a factor of 2.3. This work includes a naive approach to designing a wait-free ring buffer using a multi-word compare-and-swap algorithm (MCAS) [2]. This allows us to explore the performance difference between our *FAA* based approach to that of a *CAS* based approach. Sec. **??** provides a detailed analysis of the performance difference between these approaches.

We compare the performance of our implementation to that of other known concurrent ring buffers. In this comparison, we explore how different distributions of operations, number of threads, and ring buffer size affect the throughput of each implementation. On average, our design outperforms other designs by % operations per second. Compared to Intel Thread Building Blocks' concurrent bounded queue, we perform % more operations per second. Our results support our hypothesis that our design is scalable, making it optimal for many-core and real-time systems.

We provide the following novel contributions:

- To our knowledge this the first wait-free ring buffer. Other known approaches, are susceptible to hazards such as live-lock and thread starvation.
- Our design presents a unique way of applying sequence numbers and bitmarking to maintain provide the FIFO property. This allows a simple way to mark and correct out-of-sync locations.
- Our design maintains throughput in scenarios of high thread contention. Other known approaches degrade as the thread contention increases, making it ideal for highly parallel environments.

2. Related Works

In this section we describe the implementation of other concurrent multiple producers and multiple consumers ring buffers.

Tsigas [7] presents a lock free ring buffer in which threads compete to update the head and tail locations. This design achieves dead-lock and live-lock freedom, but as we show in Sec. 10, this design scales poorly. We attribute this poor performance to contention placed on head and tail locations. The threads that fail their *CAS* operations are then forced to retry at the next location. Some contention is reduced by lazily updating the head and tail references. This requires an increased amount of read operations for threads to locate the actual index to operate, offsetting some contention.

—

Tsigas [7] presents a lock free ring buffer in which threads compete to update the head and tail locations. Enqueue is performed by determining the tail of the buffer and then enqueuing an element using a *cas* operation. Dequeue is performed by determining the head of the buffer and then dequeueing the current element using a *cas* operation. This design achieves dead-lock and live-lock freedom, however, if a thread is unable to perform a successful *cas* operation, the thread will starve. Unlike other designs, which are able to diffuse contention, the competitive nature of this design leads to increased contention and poor scaling.

Krizhanovsky [6] presents a lock-free and high performance ring buffer. This implementation, relies on the *faa* operation to increment head and tail counters, which assigns the location to perform an operation, thereby reducing thread contention and providing better scaling. Each thread maintains a separate tail and head value of the index it last completed an enqueue or dequeue, respectively. The smallest of all threads' local head and tail values are used to determine the head and tail value at which all threads have completed their operations. These values prevent operations from attempting to enqueue or dequeue at a location the previous operation has not yet completed. Though this a non-blocking design, it is blocking in the case that if the buffer is empty or full, threads performing a dequeue or enqueue will wait until the buffer is not empty or not full, respectively.

The industry standard for many concurrent data structures is provided by Intel Thread Building Blocks (TBB) [**?**]. This library provides a *concurrent bounded queue* which utilizes a fine-grained locking scheme. The algorithm uses an array of *micro queue* to alleviate contention on individual indices. Upon starting an operation, threads are assigned a ticket value which is used to determine sequence of operations in each *micro queue*. If a threads ticket is greater than the expected, it will spin wait until the delayed threads have completed their operations.

3. Restrictions and Limitations

Our approach requires the a sequentially consistent memory model to insure proper ordering of operations, as well as support for the atomic primitives *compare and swap* (*cas*), *fetch and add* (*faa*), *fetch and or* (*fao*), *load*, and *store*. Our design also reserves the least significant bit of a reference for state identification.

The presented implementation omits details related to memory management of short lived objects (i.e. *Helper* and *Node* objects). The tested implementation uses a scheme based on the combination of hazard pointers **??** and reference counting **??** to prevent these objects from being reused prematurely. Without such protection, it could introduce the ABA-problem **??**. For example, a thread could determine that it should replace a reference to a *Node* object with a new *Node* object. However, before calling the *cas* operation, the object was removed and reallocated to another thread. That thread, then placed the object at the same location it was removed from. The first thread would be unaware that the contents of the *Node* has changed, would incorrectly replace it.

For brevity and clarity, the pseudocode omits code related to the unbitmarking of references. If a reference has been determined to hold a bitmark (i.e. *isSkipped* returning true), the next step would be to remove the bitmark from the local copy before dereferencing the object.

The use of the sequence identifiers introduces the *theoretical* danger where the value rolls over, leading to two threads being

[1] First in First Out

assigned the same *seqid*. In the following algorithm discussions, we assume the sequence identifier does not have a maximum value and thus can not roll over. The tested implementation uses a 64 bit signed long, which allows the ring buffer to support a maximum of 2^{63} enqueue operations. In the event a roll over occurs, indicated by a negative result when incrementing a sequence counter, our implementation will creates a new internal ring buffer object where all subsequent elements are enqueued. This is safe for systems using less than 2^{63} threads.

To achieve FIFO behavior of our ring buffer, we make the restriction that an element can only be dequeued by a thread whose *seqid* matches the *seqid* of the *Node* containing the element. To prevent the case where other threads maybe blocked in the event a dequeuer never removes its assigned element, we developed a method by which a thread is able to safely skip a position that holds a value assigned to another thread.

4. Definitions and Structures

Structures:

- RingBuffer:

```
{atomic array[], atomic int head,
atomic int tail}
```

- NullNode

```
{const long seqid}
```

- ElemNode

```
{const long seqid, const Element element}
```

- Helper

```
{const Op *op, const long seqid, Node *old}
```

- EnqueueOp

```
{const Element, atomic Helper *helper}
```

- DequeueOp

```
{atomic Helper *helper}
```

Supporting Functions: Add 1-2 sentence descriptions

- isNull():
- isElement():
- isSkipped():
- setSkipped():
- makeSkipped():
- isHelper():
- getNextTail():
- getNextHead():
- getPosition():
- BackOff():

4.1 Progress Assurance

For a design to be wait-free, a thread must not be continually denied access to a necessary resource. This design employs a progress assurance scheme to prevent thread starvation. Without this progress assurance scheme, the design presented may encounter a rare condition in which threads starve. With larger ring buffer sizes, this possibility can be reduced further.

The progress assurance scheme is designed similar to the announcement table presented by Herlihy [**?**]. Threads will check the table incrementally at the start of every operation and help complete any operation found as presented by Kogan [5]. Our design is inpired by the design presented by Feldman, which is based on these two designs [1]. This design uses an announcement table of *OpRec* which contains a control word indicating an operations state. When the control word is a reference to a descriptor object the operation has been completed, otherwise, threads will continue to help.

Describe it, cite Bible's Annoucement Table, Kogans method, Vector's Descriptor Association

5. Algorithm Overview

In this section, we first present an overview of our approach, and then describe our specific implementation of the enqueue and dequeue operations.

In contrast to designs in which threads compete to finish an operation, our approach diffuses thread contention and reduces forced dependency. We accomplish this through the use of sequence counters to assign order to threads performing enqueue and dequeue operations. The presented implementation stores references to (of) *Node* objects in the ring buffer. As described in Sec. 4, *Node* objects contain a *seqid* member, and if its is an *ElemNode* it also contains an element member. The position at which a *Node* reference is placed, is determined by performing a mod operation on the *seqid* using the length of the ring buffer.

Our implementation solves several dangers that may arise by using this methodology. The first, is the case where an enqueue thread is assigned a position, but an *ElemNode* already exists at that position. The second, is the case where a dequeuer thread is assigned a position, but an *EmptyNode* exists at that position. And a third danger, is the case where a dequeuer thread is assigned a position that holds a value, but its *seqid* is less than the thread's *seqid*. These dangers can arise as a result of poor system scheduling, inopportune context switches, and thread delay. Our implementations include a novel solution that uses the atomic fetch and or (*fao*) operation along with the *seqid* to ensure that these dangers do not affect the correctness of our implementation.

The remainder of this section is structured as follows: We present a detailed explanation of enqueue and dequeue operations. This explanation includes algorithm behavior, linearizability, and wait-freedom descriptions. Next, provide several case examples using Figs. **??** to describe several execution scenarios. After which, we present an informal reasoning on algorithm correctness and linearizability.

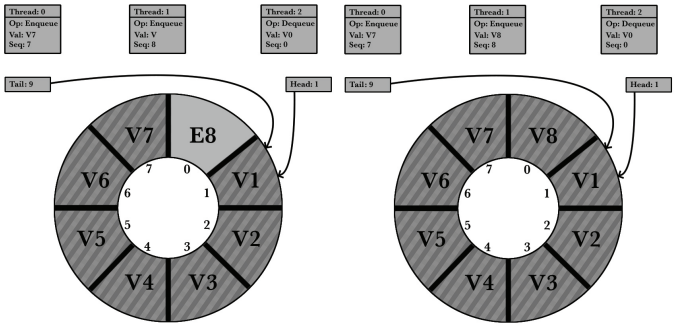

Figure 2. Ideal

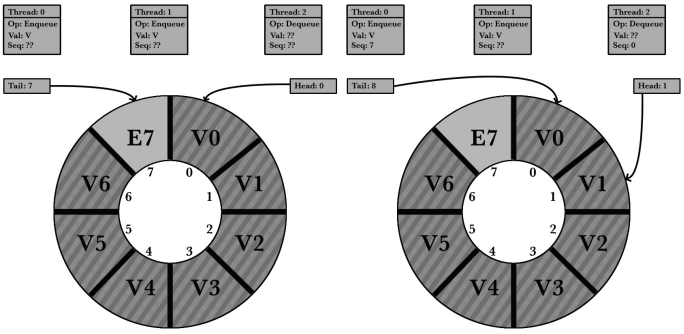

Figure 1. Init

Algorithm 1 Dequeue $(\&result)$

```
 1: TryHelpAnother()
 2: fail_count = 0
 3: while true do
 4:     if is_empty() then
 5:         return false
 6:     end if
 7:     seqid = next_head_seq()
 8:     pos = get_position(seqid)
 9:     new_node = new EmptyNode(seqid + capacity)
10:     while true do
11:         if fail_count++ == MAX_FAILS then
12:             op = new DequeueOp(this)
13:             make_announcement(op)
14:             return op.result(result)
15:         end if
16:         node = buffer[pos].load()
17:         if node.op then
18:             node.op.associate(node, &(buffer[pos]))
19:             continue
20:         else if isSkipped(node) and isEmpty(node) then
21:             if buffer[pos].cas(node, new_node) then
22:                 break
23:             else
24:                 continue
25:             end if
26:         else if seqid > node.seqid then
27:             backoff()
28:             if node == buffer[pos].load() then
29:                 if isEmpty(node) then
30:                     if buffer[pos].cas(node, new_node) then
31:                         break
32:                     end if
33:                 else
34:                     atomic_mark_skip(&buffer[pos])
35:                 end if
36:             end if
37:         else if seqid < node.seqid then
38:             break
39:         else
40:             if isElem(node) then
41:                 if isSkipped(node) then
42:                     new_node = setSkipped(new_node)
43:                 end if
44:                 success = buffer[pos].cas(node, new_node)
45:                 if success then
46:                     *result = node.value
47:                     return true
48:                 end if
49:             else
50:                 backoff()
51:                 if node == buffer[pos].load() then
52:                     if buffer[pos].cas(node, new_node) then
53:                         break
54:                     end if
55:                 end if
56:             end if
57:         end if
58:     end while
59: end while
```

Algorithm 2 Wait-Free Dequeue (op)

```
 1: seqid = get_head_seq() -1
 2: while op.in_progress() do
 3:     if is_empty() then
 4:         return op.try_set_failed()
 5:     end if
 6:     seqid++
 7:     pos = get_position(seqid)
 8:     while op.in_progress() do
 9:         node = buffer[pos].load()
10:         if node.op then
11:             node.op.associate(node, &(buffer[pos]))
12:             continue
13:         else if isSkipped(node) then
14:             if isEmpty(node) then
15:                 if buffer[pos].cas(node, new_node) == false then
16:                     continue
17:                 end if
18:             end if
19:             break
20:         else
21:             if seqid < node.seqid then
22:                 backoff()
23:                 if node == buffer[pos].load() then
24:                     break
25:                 end if
26:             else if sec > node.seqid then
27:                 break
28:             else
29:                 if isElem(node) then
30:                     new_node = new ElemNode(seqid, node.value, op)
31:                     if buffer[pos].cas(node, new_node) then
32:                         op.associate(new_node, &(buffer[pos]))
33:                         return
34:                     end if
35:                 else
36:                     break
37:                 end if
38:             end if
39:         end if
40:     end while
41: end while
```

Algorithm 3 Op::try_set_failed ()

```
1: helper.cas(null, FAIL)
```

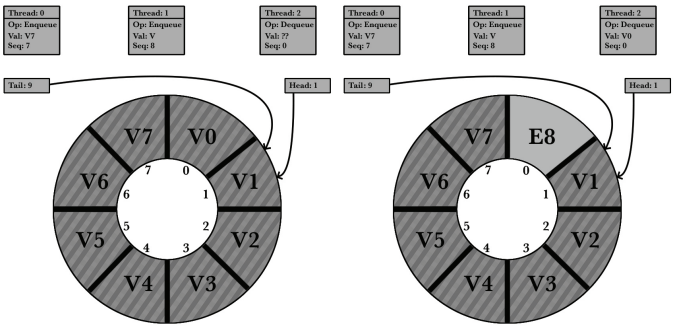

Figure 3. Full

Algorithm 4 DequeueOp::associate $(node, address)$

```
 1: success = helper.cas(null, node)
 2: if success or helper.load() == node then
 3:     new_node = NullNode(node.seqid + capacity)
 4:     if address.cas(node, new_node) == false then
 5:         node = setSkipped(node)
 6:         if address.load() == node then
 7:             new_node = setSkipped(new_node)
 8:             address.cas(node, new_node)
 9:         end if
10:     end if
11: else
12:     node.op.store(null);
13: end if
```

6. Dequeue

The following describes how a dequeue operation is performed. To guide the reader, we reference specific lines in Alg. 1

To *dequeue* an element, a thread will first check if the ring buffer is empty (L. 4), returning false if it is. Otherwise, it will acquire a dequeue sequence number, *seqid*, from the head counter and determine the position, *pos*, to dequeue an element (L. 7- 8). The thread will then prepare an *EmptyNode* to replace the dequeued value (L. **??**). The *seqid* of the *EmptyNode* is set to the assigned *seqid* plus the buffer's capacity. In the common case, the thread will replace an *ElemNode*, whose *seqid* matches its assigned *seqid*, with the prepared *EmptyNode* (L. 44). Uncommon cases, which are often the result of thread delay, are described below.

- The node holds a reference to an operation record (L. 17). This indicates the node was placed as part of another thread's operation. The thread will call the associate function for that operation and upon its return either the node has been replaced or the reference to the operation record has been removed. The node at the current position will then be re-examined.
- The node currently at the position has a *seqid* number less than the assigned or it is an *EmptyNode seqid*(L. 26, L. **??**). In this event, we call the backoff routine to provide an opportunity for a delayed thread to complete its operation. If the current node has not changed the thread will advance the position by either replacing an *EmptyNode* with the prepared *EmptyNode* or performing an atomic bitmark if it is an *ElemNode*. If the node has changed, the position will be re-examined.

  In order to achieve FIFO ordering, threads can only remove an *ElemNode* if it was assigned that node's *seqid*. The atomic bitmark allows a thread to get a new *seqid* without the risk of an *ElemNode* being enqueued with a *seqid* that has been given up.
- The node currently at the position is bitmarked and an *EmptyNode* (L. 20). This state resulted from the previously described, in which a thread bitmarked an *ElemNode*. The thread who was assigned that node's *seqid* must have replaced it with a bitmarked *EmptyNode*. Sec. 9 describes the importance of replacing a bitmarked *ElemNode* with a bitmarked *EmptyNode*. This state is resolved by replacing the bitmarked *EmptyNode* with an unbitmarked *EmptyNode*.
- The node currently at the position has a *seqid* greater than the assigned *seqid* (L. 37). This implies that some thread caused this thread's *seqid* to be skipped and as result this thread needs to get a new *seqid*.

These uncommon cases could force a thread to indefinitely reattempt its operation. However, we employ a progress assurance scheme to prevent this from occurring. Specifically, in the event a threshold is reached (L. 11) the thread will make an announcement and switch to a slow path dequeue operation (Alg. 2). We describe in Sec. **??**, specifically how this announcement scheme is used to ensure an operation is completed in a finite number of steps.

The wait-free dequeue operation functions very similarly to the normal dequeue operation with the following key change.

- The operation ends when some threads calls either *op.try_set_failed* or *op.associate*. Upon return of these functions it is guaranteed that the operation has been completed.
- The thread is not assigned a *seqid* but instead loads the current value of the head counter. This is important to prevent the scenario where a thread is assigned a position after the operation has been completed and as a result no longer needs to dequeue a value.
- The node placed holds a reference to the operation record it was placed on behalf of. This is used to prevent the case where multiple threads complete the same operation. Multiple nodes may reference the same operation record, but the operation record may only refence one of these nodes: the completed operation.
- After a node is placed, the operation's associate function is called. This ensures that if the node was placed incorrectly its reference to the operation record will be removed. If it was placed correctly, then the node will be replaced by an *EmptyNode*.

6.1 Linearizability

In general the linearization point for a successful dequeue operation is the atomic *faa* operation which assigned the *seqid* (L. 7). However, this is not realized until the thread successfully places an *EmptyNode* in place of the *ElemNode* with *seqid* matching the *seqid* assigned (L. 44). If the wait-free path is used then the linearization point for a successful dequeue operation is the successful association of an *ElemNode* and a *DequeueOp* (Alg. 4 L. 1).

The linearization point for a failed dequeue operation is when a thread thread detects that the ring buffer is empty (Alg. 1 L. 4).

If the wait-free path is used then the linearization point for a failed dequeue operation is the successful *cas* that set the operation's *helper* member to the *FAIL* constant (Alg. 3 L. 1).

7. Enqueue

The following describes how a enqueue operation is performed. To guide the reader, we reference specific lines in Alg. 5

To *enqueue* an element, a thread will first check if the ring buffer is full (L. 4), returning false if it is. Otherwise, it will acquire a enqueue sequence number, *seqid*, from the tail counter and determine the position, *pos*, to enqueue an element (L. 7- 8). The thread

Algorithm 5 Enqueue (val)

```
 1: TryHelpAnother()
 2: fail_count = 0
 3: while true do
 4:     if is_full() then
 5:         return false
 6:     end if
 7:     seqid = next_tail_seq()
 8:     pos = get_position(seqid)
 9:     new_node = new ElemNode(seqid, val)
10:     while true do
11:         if fail_count++ == MAX_FAILS then
12:             op = new EnqueueOp(this, val)
13:             make_announcement(op)
14:             return op.result()
15:         end if
16:         node = buffer[pos].load()
17:         if node.op then
18:             node.op.associate(node, &(buffer[pos]))
19:             continue
20:         else if isSkipped(node) then
21:             break
22:         else if node.seqid < seqid then
23:             backoff()
24:             if node != buffer[pos].load() then
25:                 continue
26:             end if
27:         end if
28:         if node.seqid <= seqid and isEmpty(node) then
29:             success = buffer[pos].cas(node, new_node)
30:             if success then
31:                 return true
32:             end if
33:             continue
34:         else if node.seqid > seqid or isElement(node) then
35:             break
36:         end if
37:     end while
38: end while
```

Algorithm 6 Wait-Free Enqueue op

```
 1: seqid = get_tail_seq() -1
 2: while op.in_progress() do
 3:     if is_full() then
 4:         op.try_set_failed()
 5:         return
 6:     end if
 7:     seqid++
 8:     pos = get_position(seqid)
 9:     new_node = new ElemNode(seqid, val, op)
10:     while op.in_progress() do
11:         node = buffer[pos].load()
12:         if node.op then
13:             node.op.associate(node, &(buffer[pos]))
14:             continue
15:         else if isSkipped(node) then
16:             break
17:         end if
18:         if node.seqid < seqid then
19:             backoff()
20:             if node != buffer[pos].load() then
21:                 continue
22:             end if
23:         end if
24:         if node.seqid <= seqid and isEmpty(node) then
25:             if buffer[pos].cas(node, new_node) then
26:                 op.associate(new_node, &buffer[pos]);
27:                 return
28:             end if
29:         else if node.seqid > seqid or isElement(node) then
30:             break
31:         end if
32:     end while
33: end while
```

Algorithm 7 EnqueueOp::associate $(node, address)$

```
 1: success = helper.cas(null, node)
 2: if success or helper.load() == node then
 3:     node.op.store(NULL)
 4: else
 5:     new_node = NullNode(node.seqid)
 6:     if address.cas(node, new_node) == false then
 7:         node = setSkipped(node)
 8:         if address.load() == node then
 9:             new_node = setSkipped(new_node)
10:             address.cas(node, new_node)
11:         end if
12:     end if
13: end if
```

will then prepare an *ElemNode* to hold the element being enqueued (L. 9). The *seqid* of the *ElemNode* is set to the assigned *seqid*. In the common case, the thread will replace an *EmptyNode*, whose *seqid* matches its assigned *seqid*, with the prepared *ElemNode* (L. 29). Uncommon cases, which are often the result of thread delay, are described below.

- The node holds a reference to an operation record (L. 17). This indicates the node was placed as part of another thread's operation. This must be resolved by calling the associate function for that operation. Upon its return either the node has been replaced or the reference to the operation record has been removed. The thread will then re-examine the current position.
- The reference currently at the position has a bitmark (L. 20), indicating it was marked as skipped. This indicates the node at the position needs to be fixed by a dequeue thread. As a result, the enqueue thread will get a new *seqid* and retry its operations.
- The node currently at the position has a *seqid* number less than the assigned the *seqid* (L. 22). In this event,the thread will call the backoff routine to provide time for a delayed thread to complete its operation. If the current node has not changed and it is an *EmptyNode* the thread will attempt to replace it with the prepared *ElemNode*. Otherwise, it will get a new *seqid*.
- The node currently at the position has a *seqid* greater than the assigned *seqid* (L. 34). This implies that some thread caused this thread's *seqid* to be skipped and as result this thread needs to get a new *seqid*.

As described in Sec. 7, these uncommon cases could force a thread to indefinitely reattempt its operation. The following are key differences between the normal enqueue operation and the wait-free enqueue operation.

- The operation ends when some threads calls either *op.try_set_failed* or *op.associate*. Upon return of these functions it is guaranteed that the operation has been completed.
- The thread is not assigned a *seqid* but instead loads the current value of the tail counter. This is important for achieving maximum unskipped buffer indices resulting from cancelled operations.
- The node placed holds a reference to the operation record it was placed on behalf of. This is used to prevent the case where multiple threads complete the same operation. Many nodes may reference the same operation record, but the operation record may only refence one of these nodes: the completed operation.
- After a node is placed, the operation's associate function is called. This ensures that if the node was placed incorrectly then the node will be replaced by an *EmptyNode*. If it was placed correctly, its reference to the operation record will be removed.

8. Wait-Freedom

We show that both the dequeue and enqueue algorithms are wait-free by first examining the loops and their terminating conditions. Both algorithms contain two nested while loops and in general the outer loop executes until the operation has been completed and the inner loop executes until the assigned *seqid* is no longer viable. The *MAX_FAILS* constant is used to place an upper bound on the number of times a thread will execute these loops. If this constant is reached, *make_announcement* is called and be our definition of this function upon its return the operation must be complete. Thus if all functions called by *dequeue* or *enqueue*, then these functions are wait-free.

Except for *TryHelpAnother* and *make_announcement*, the functions called are utility functions that are used to simplify the code explanation. These functions are inherently wait-free because they contain no loops or calls to non-wait-free functions. However, both *TryHelpAnother* and *make_announcement* are capable of calling *wait-free dequeue* or *wait-free enqueue*, thus wait-freedom is determined by the progress guarantee of these two operations.

Both *wait-free dequeue* or *wait-free enqueue* employ the same looping structures which terminate when the operation record (*op*) is no longer in progress. An operation record is in progress as long as its *helper* member is *null*. discuss how its lf and blah blah and if it enough operations occur, all threads will be helping, thus it is wait-free...

9. Correctness

This section presents an informal proof that the ring buffer is correct. It is divided into three parts; the first two we present an informal proof that shows the enqueue and dequeue operations behave as described and the third part shows that the buffer provides the FIFO property when operations are ordered by their linearization points.

9.1 Elements are correctly enqueued

9.2 Elements are correctly dequeued

9.3 First In First Out behavior

10. Results

This section presents a series of experiments that compare the presented ring buffer design to other comparable designs. For each experiment, we present a detailed description, a performance comparison, and an analysis explaining the differences in performance.

10.1 EXPERIMENT NAME

description
- real-world analog
- chart(s)
- analysis

11. Conclusion

Conclusion

References

[1] Feldman. How to be a big noob. *International Journal of Trolls and Nooblets*, pages 1–2, 2014.

[2] S. Feldman, P. LaBorde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, pages 1–25, 2014.

[3] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing*, pages 265–279. Springer, 2002.

[4] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[5] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.

[6] A. Krizhanovsky. Lock-free multi-producer multi-consumer queue on ring buffer. *Linux Journal*, 2013(228):4, 2013.

[7] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143. ACM, 2001.