SAND2015-2903C

A Scalable Multi-Producer Multi-Consumer Wait-Free Ring Buffer

Steven Feldman
University of Central Florida
Carlos Valeraleon
University of Central Florida
Damian Dechev
University of Central Florida and Sandia National Laboratories

ABSTRACT

The Ring Buffer is a staple data structure in computer science. They are excellent for high demand applications such as multimedia, network routing, and trading systems. We present a new ring buffer providing the wait-free progress guarantee suitable for such applications. We are not aware of any other array-based design providing this guarantee that every thread must complete its operation in a finite number of steps. Thus it is desirable for real-time and mission critical systems. However, the added complexity to achieve this guarantee often results in reduced performance.

To prevent such pitfalls, our ring buffer introduces a methodology for diffusing contention resulting in increased performance and scalabilty. On average, the design experiences 100% more operations than a coarse-grained locking approach, 15% more than TBB's concurrent bounded queue, 10% more than the lock-free approach presented by Krizhanovsky, and 140% more than the cycle queue by Tsigas.

Keywords

concurrent·parallel·non-blocking·wait-free·queue·ring buffer

1. INTRODUCTION

A ring buffer or cyclical queue is a First In, First Out (FIFO) queue that stores elements on a fixed-length array.This allows for efficient O(1) operations, cache-aware optimizations, and low memory overhead. Because ring buffers are limited to only the array and two counters they are desirable for systems with limited memory. Many applications (e.g. cloud-based services) depend on ring buffers to pass work from one thread to another. The rise in many-core architecture has resulted in increased performance from shared data structures such as ring buffers. Existing research has forgone the use of locks and permitted greater scalability and core utilization for such designs. Such non-blocking designs are categorized by the level of progress they guarantee with wait-freedom being the most desirable categorization. Such a guarantee provides freedom from deadlock, livelock, and thread starvation. Lock-free and obstruction-free designs are not safe from all of these pitfalls [3].

We are aware of two other non-blocking ring buffers in literature. Tsigas et al. presented a lock-free approach in which threads compete to apply an operation [7]. Section 10 shows 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 [5] but unfortunately, the design is susceptible to thread starvation and is not thread death safe.

In the presented performance comparison, on average, our design outperforms best available alternative designs by at least 10%. Compared to the Intel Thread Building Blocks' (TBB) concurrent_bounded_queue, it performs 15% more operations. These results support the hypothesis that the design is optimal for many-core systems, as seen in Section 10.

We provide the following novel contributions. This is the first array-based wait-free ring buffer. Other known approaches are susceptible to hazards such as live-lock and thread starvation. Our design presents a new way of applying sequence numbers and bitmarking to maintain the FIFO property. Our design maintains throughput in scenarios of high thread contention. Other known approaches degrade as the thread contention increases, making our implementation more suitable for highly parallel environments.

2. RELATED WORKS

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

Tsigas et al. [7] present a lock-free ring buffer in which threads compete to update the head and tail locations. An enqueue is performed by determining the tail of the buffer and then enqueuing an element using a *CAS* operation. A 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; 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 [5] presents a lock-free and high performance ring buffer. This implementation relies on the *FAA* operation to increment head and tail counters, which determine the index to perform an operation. This reduces thread contention and provides better scaling. Each thread maintains

a separate tail and head value of the last completed 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 an attempt to enqueue or dequeue at a location where a previously invoked thread has yet to complete it's operation.

Intel Thread Building Blocks (TBB) [6], an industry standard for many concurrent data structures provides a *concurrent bounded queue* which utilizes a fine-grained locking scheme. The algorithm uses an array of micro queues to alleviate contention on individual indices. Upon starting an operation, threads are assigned a ticket value which is used to determine the sequence of operations in each *micro queue*. If a thread's ticket is not expected, it will wait until the delayed threads have completed their operations.

Since there are no other known wait-free ring buffers to compare the performance of our approach against, we implemented one using a wait-free Multi-Word Compare-and-Swap (MCAS) [1]. The design uses MCAS to move a head and tail marker around the buffer. Specifically, after identifying the position holding the head marker, a thread enqueues an element by using MCAS to replace the head marker with the value being enqueued and replace an empty value at the next position with the head marker. If the next position holds the tail marker, this indicates the buffer to be full. Similarly, to dequeue a value, a thread uses MCAS to replace the tail marker with an empty value and to replace the value at the next position with the tail marker. If the tail marker is followed by the head marker, this indicates the buffer to be empty.

In contrast to the design presented by Tsigas, our buffer does not lazily update head and tail values but instead increments the values for every operation. By reducing the amount of threads attempting to *CAS* at a given buffer location, our approach is able to reduce the amount of failed *CAS* operations. We achieve this with the help of using *FAA* to increment head and tail values similar to Krizhanovsky. However, unlike this design we avoid the danger of live-lock by using these values as sequence numbers and implementing new strategies to maintain order.

3. RESTRICTIONS AND LIMITATIONS

Our approach requires support for the following atomic primitives: *Compare-and-Swap* (*CAS*), *Fetch and Add* (*FAA*), *Load*, and *Store*. Our design also reserves the least significant bit of a reference for designating a state.

The presented implementation omits details related to memory management of short-lived 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 brevity and clarity, the pseudocode omits code related to the bitmarking of references. If a reference has been determined to hold a bitmark, the next step would be to remove the bitmark from the local copy before dereferencing the object.

4. ALGORITHM OVERVIEW

In this section, we first define the algorithms supporting functions, then present an overview of our approach, and finally describe our specific implementation of the enqueue and dequeue operations. The specific implementation of the following supporting algorithms are omitted for brevity and we instead provide their definitions:

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 ordering of elements. A thread performing an enqueue or dequeue operation will perform an *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. This *seqid* is also stored in a *Node* for placement in the buffer.

Op.try_set_failed()	attempts to CAS an operation record helper to a FAIL constant
is_EmptyNode(Node)	returns whether the Node is an EmptyNode
is_ElemNode(Node)	returns whether the Node is an ElemNode
is_skipped(Node)	returns whether the Node is marked skipped
set_skipped(Node)	atomically marks the Node as skipped
next_head_seq()	increments the head seq value and returns the previous value
next_tail_seq()	increments the tail seq value and returns the previous value
get_head_seq()	returns the current head seq
get_tail_seq()	returns the current tail seq
get_position(seq)	returns the buffer position for a seq value (seq modulo buffer capacity)
backoff()	suspends a thread for some interval of time to reduce contention

Figure 1: Supporting Functions

A *Node* in the ring buffer may be one of two types. If the *Node* is an *ElemNode* it contains an element member and the *seqid*. An *EmptyNode* simply contains the *seqid* and represents an empty buffer location. Thus we initialize our buffer by storing an *EmptyNode* at each location with the *seqid* equal to that index.

Our implementation solves several dangers that may arise by using this methodology.

- An enqueue thread is assigned a position that holds an *ElemNode*.
- A dequeue thread is assigned a position that holds an *EmptyNode*.
- A dequeue thread is assigned a position that holds an *ElemNode*, but its *seqid* is less than the thread's *seqid*.

These dangers can arise as a result of inopportune context switches and/or thread delay. Our implementation includes a novel solution that uses reference bitmarking along with the *seqid* to ensure that these dangers do not affect the correctness of our implementation.

5. PROGRESS ASSURANCE

For a design to be wait-free, a thread must be able to complete its operation in a finite amount of time. If a given

thread cannot complete its operation, there is not system-wide progress. Without the progress assurance scheme, our algorithm may encounter a rare condition in which threads starve. With larger ring buffer sizes this possibility can be further reduced. This is because the larger capacity increases the number of operations between a given enqueue or a given dequeue making it less likely for threads to compete for the same buffer index.

The progress assurance scheme is designed in a similar fashion to Herlihy's announcement table [2]. Threads check the table incrementally at the start of every operation and help complete any operation found (as presented by Kogan [4]). This design uses an announcement table of operation records, which contains a control word indicating an operation's state. A specific operation record exists for both types of operations performed on the buffer: *EnqueueOp* and *DequeueOp*. The control word for each operation record is a reference to a *Node* object. In addition to the control word, the *EnqueueOp* must contain the element to enqueue. When the control word is no longer null, the operation has been completed.

6. DEQUEUE

To *dequeue* an element a thread first checks 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. 9). 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 a delayed thread's operation. Upon its return from the associate function the node will be replaced or the reference to the operation record will be removed. The node at the current position is then re-examined.

- The node at the current position is an *EmptyNode* or has a *seqid* number smaller than the *seqid* assigned to the thread. (L. 9, 26). In this event, a backoff routine is used to provide an opportunity for the delayed thread to complete its operation. If the current node has not changed, the thread will attempt to advance the position by either replacing the current *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 had been 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 is an *EmptyNode* (L. 20). This state resulted from the previously described situation, in which a thread bitmarked an *ElemNode*. The thread assigned that node's *seqid* must have replaced it with a bitmarked *EmptyNode*. Section 9 describes the importance of replacing

Algorithm 1 Dequeue (&*result*)

```
 1: try_help_another()
 2: fails = 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:   n_node = EmptyNode(seqid + capacity)
10:   while true do
11:     if fails++ == MAX_FAILS then
12:       op = DequeueOp()
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 is_skipped(node) and is_EmptyNode(node) then
21:       if buffer[pos].cas(node, n_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 is_EmptyNode(node) then
30:           if buffer[pos].cas(node, n_node) then
31:             break
32:           end if
33:         else
34:           set_skipped(&buffer[pos])
35:         end if
36:       end if
37:     else if seqid < node.seqid then
38:       break
39:     else
40:       if is_ElemNode(node) then
41:         if is_skipped(node) then
42:           n_node = set_skipped(n_node)
43:         end if
44:         success = buffer[pos].cas(node,n_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, n_node) then
53:             break
54:           end if
55:         end if
56:       end if
57:     end if
58:   end while
59: end while
```

Algorithm 2 DequeueOp::associate (*node*, *address*)

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

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*.

If no additional mechanism is employed, these uncommon cases could force a thread to indefinitely reattempt its operation. In the event the user-specified threshold of allowed attempts is reached (L. 11), the thread will post an announcement and switch to a slow path dequeue operation (Algorithm 3). We describe in Section 8 specifically how this announcement scheme is used to ensure an operation is completed in a finite number of steps.

Algorithm 3 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 is_skipped(node) then
14:       if is_EmptyNode(node) then
15:         if !buffer[pos].cas(node, n_node) 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 is_ElemNode(node) then
30:           n_node = ElemNode(seqid, node.value, op)
31:           if buffer[pos].cas(node, n_node) then
32:             op.associate(n_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
```

The wait-free dequeue operation is similar to the regular dequeue operation with the following differences.

- The operation ends when some thread calls *op.try_set_failed* or *op.associate*. Upon return from either function the operation will be 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 being executed. 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 reference a single node.

- 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, the node will be replaced by an *EmptyNode*.

6.1 Linearizability and Correctness

6.1.1 Linearizability

In the general case, the linearization point for a successful dequeue operation is the atomic *FAA* operation that assigns 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* (Algorithm 2 L. 1).

The linearization point for a failed dequeue operation is when a thread detects that the ring buffer is empty (Algorithm 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.

6.1.2 Correctness

To ensure elements are dequeued in a correct order, only the thread assigned the *seqid* matching the *ElemNode* in the buffer may return that stored element. If a dequeue operation fetches a head *seqid* greater than the current, the position is marked for correction but the *ElemNode* is not removed. This behavior ensures delayed operations linearize before subsequent operations at the same index. Even though the delayed operation may return after an operation with a greater *seqid*, the former linearizes first because of its lesser *seqid*.

Since dequeues may also modify locations holding an *EmptyNode*, these operations must also be correct. However a dequeue will only replace an *EmptyNode* with another *EmptyNode* having a greater *seqid*. As other dequeue operations may also only dequeue *ElemNodes* with matching *seqid*, increasing the *seqid* of the current *EmptyNode* does not affect their linearization ordering. Additionally, enqueue ordering will not be invalidated as the enqueue sharing the *seqid* of the previous *EmptyNode* does not linearize until its *EmptyNode* is placed. This enqueue will then reset its linearization point when it finds a new *seqid* or when the helping scheme associates its *EmptyNode* and *EnqueueOp*.

7. ENQUEUE

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 an enqueue sequence number (*seqid*) from the tail counter and determine the position (*pos*) to enqueue an element (L. 7- 8). The thread will then prepare an *ElemNode* with the assigned *seqid* that holds the element being enqueued (L. 9). In the common case, the thread will replace

Algorithm 4 Enqueue (*val*)

```
 1: try_help_another()
 2: fails = 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:   n_node = ElemNode(seqid, val)
10:   while true do
11:     if fails++ == MAX_FAILS then
12:       op = EnqueueOp(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 is_skipped(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 is_EmptyNode(node) then
29:       success = buffer[pos].cas(node, n_node)
30:       if success then
31:         return true
32:       end if
33:       continue
34:     else if node.seqid > seqid or is_ElemNode(node) then
35:       break
36:     end if
37:   end while
38: end while
```

Algorithm 5 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:   n_node = 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 is_skipped(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 is_EmptyNode(node) then
25:       if buffer[pos].cas(node, n_node) then
26:         op.associate(n_node, &buffer[pos]);
27:         return
28:       end if
29:     else if node.seqid > seqid or is_ElemNode(node) then
30:       break
31:     end if
32:   end while
33: end while
```

Algorithm 6 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:   n_node = EmptyNode(node.seqid)
6:   if !address.cas(node, n_node) then
7:     node = set_skipped(node)
8:     if address.load() == node then
9:       n_node = set_skipped(n_node)
10:      address.cas(node, n_node)
11:    end if
12:  end if
13: end if
```

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, showing that the node needs to be fixed by a dequeue thread. As a result, the enqueue thread will get a new *seqid* and retry.
- The node currently at the position has a *seqid* number less than the assigned *seqid* (L. 22). In this event, the thread will call the backoff routine to provide time for the 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 current node at the enqueue location 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 Section 6 unless the wait-free path is used, these uncommon cases could force a thread to indefinitely reattempt its operation. The following are key differences between the regular enqueue operation and the wait-free enqueue operation:

- The operation ends when some thread calls *op.try_set_failed* or *op.associate*.
- The thread is not assigned a *seqid* but instead loads the current value of the tail *seqid* counter. This is important for achieving maximum unskipped buffer indices resulting from canceled operations.
- The node placed holds a reference to the operation record being executed. 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 reference a single node.

- 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.

7.1 Linearizability and Correctness

7.1.1 Linearizability

In the general case, enqueue operations linearize at the atomic *FAA* assignment of the tail *seqid* (L. 7). The linearization point is observed by other threads when the *ElemNode* is placed in the buffer by the enqueueing thread (L. 29). When relying on the wait-free path, operations linearize upon successful association of an *ElemNode* and *EnqueueOp* (Algorithm 6 L. 1).

If the buffer is full, the enqueue operation linearizes at the detection of a full buffer (Algorithm 4 L. 4). When the wait-free path is used, a failed operation linearizes upon setting the operation's *helper* to *FAIL*.

7.1.2 Correctness

Elements are placed according to the *seqid* modulo *capacity* only if the current element is an *EmptyNode* with *seqid* less than or equal to the new. This ensures that delayed enqueue operations cannot overwrite new data but instead find a new enqueue location. Additionally, the *Node* type check designates that an *ElemNode* cannot replace another *ElemNode*. These constraints ensure enqueue operations do not overwrite other enqueued elements and are placed in a correct order with respect to parallel enqueues.

8. WAIT-FREE PROGRESS

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. By our definition of this function, the operation must be complete upon its return. Thus, if all supplementary functions called by *dequeue* or *enqueue* are wait-free, 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*.

To determine the number of operations that must completed before a given thread is successfully helped, we examine the nature of the progress assurance scheme. After exceeding the MAX_FAIL constant an operation will be helped. Within $NUM\_THREADS$ operations, some arbitrary thread will have incremented its helping index to our current operation in need of help. Accordingly, after $NUM\_THREADS^2$ operations all threads must be helping the operation. With all threads helping, one thread is guaranteed to succeed its *CAS* to associate and complete the operation. Therefore, each operation is wait-free with a maximum number of attempts $MAX\_FAIL+NUM\_THREADS^2$.

9. FIRST IN FIRST OUT BEHAVIOR

Applying the constraints expressed in Sections 6.1 and 7.1 we are able to derive a correctness about the ring buffer's FIFO property. To achieve this, dequeue operations only replace an *EmptyNode* with a matching *seqid* of the thread performing the operation. This constraint is important to maintain FIFO ordering of the buffer. We have also applied a bitmarking strategy to ensure the correct ordering implied by the *seqid* values.

Without this bitmarking strategy, the following scenario presented would break FIFO behavior. Assume a buffer exists of arbitrary size N, filled to capacity. Thread T1 is assigned a dequeue operation with *seqid* 0. Before T1 is able to deqeueue the element at index 0, the thread is suspended while other threads successfully dequeue elements with *seqids* 1 through (N-1) (i.e. the remaining elements). Another thread then attempts to enqueue, incrementing the tail value and allowing T2 to examine a non-empty buffer in order to fetch *seqid* N. Rather than removing the element also seen by T1, T2 will bitmark the *ElemNode* for correction by the thread that guarantees FIFO (i.e. T1). However, if this constraint is not in place and T2 were to remove this *ElemNode*, the operation linearized after the dequeues with *seqids* 1 through (N-1). This would break the FIFO property because as described in Section 7.1, the enqueue operation for this element linearized first. However, Section 6.1 states that the element's dequeue operation must linearize before the elements with greater seqid. When T1 with *seqid* 0 removes the element, the linearization occurs before the subsequent dequeues for *seqids* 1 through (N-1). This provides FIFO ordering amongst concurrent operations. If the *ElemNode* is bitmarked, T1 will place a bitmarked *EmptyNode* to synchronize the position with the buffer.

10. RESULTS

This section presents a series of experiments that compare the presented ring buffer design to the best available alternative designs. Tests include performance results of a coarse locking design (Locking), TBB's concurrent_bounded_queue (TBB), the Tsigas cycle_queue (Tsigas), Krizhanovsky's ring buffer (Linux), and the MCAS derived approach (MCAS). For each experiment we present the configuration of the test, the collected results, and analysis of that performance.

10.1 Testing Hardware

Tests were conducted on a 64-core ThinkMate RAX QS5-4410 server running the Ubuntu 12.04 LTS operating system. The system has 4 AMD Opteron 6272 CPUs, which contain 16 cores per chip with clockrate 2.1 GHz. It is a NUMA architecture with 314 GB of shared memory. Executables were compiled with GNU's C++ compiler, g++ version 4.7 with the -O3 flag enabled.

10.2 Experimental Methodology

Our performance analysis is based on measuring the throughput of each buffer when a set of threads perform enqueue and dequeue operations. Our testing methodology consists of a main thread that initializes the buffer and spawns a set of worker threads before starting each test. The capacity of each buffer was of size $2^{24} = 16777216$ and contents were pre-filled as necessary to ensure that no buffer completely fills or empties its contents during the test. During the test, threads choose operations according to the specific enqueue and dequeue rate varied across tests. For the duration of the test the main thread will sleep for 3 seconds, notifying worker threads when the test is finished. The presented results are the average of 5 executions.

10.3 100% Distributions

In the following tests, enqueue and dequeue operations were tested separately at a 100% enqueue or dequeue rate to examine the throughput of said operations. The chosen capacity prevented any buffer from becoming full or empty during execution and each buffer was filled to capacity prior to the dequeue test. Figures 2 and 3 show that our approach provides an average of at least 25% more enqueue throughput than other locking and non-blocking approaches but falls 8% behind that of TBB and 18% the Linux buffer in regards to dequeues. However, our dequeue algorithm scales more effectively than all other designs excluding the Linux buffer, with improvement over TBB at least 68% with 64 threads. Enqueue throughput provided at least 150% improvement at 64 threads. We credit this scalability to the manner in which the buffer assigns enqueue and dequeue locations, reducing the amount of failed *CAS* operations and permitting less cache invalidations.

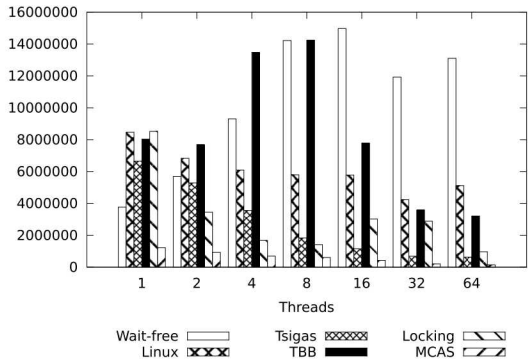

Figure 2: 100% Enqueue

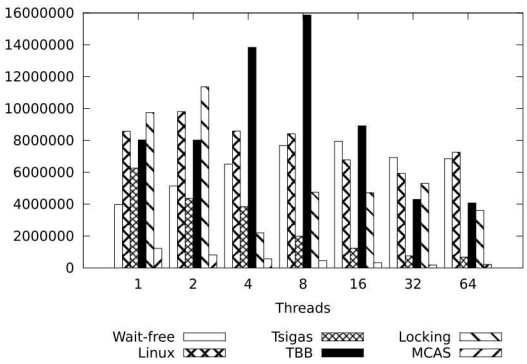

Figure 3: 100% Dequeue

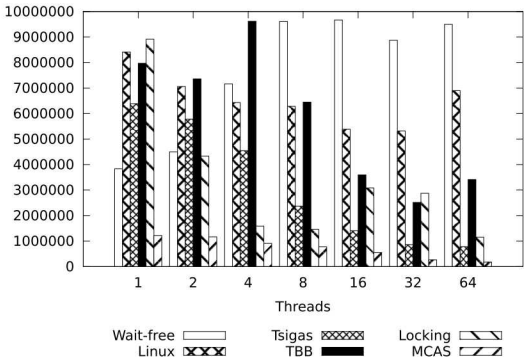

Figure 4: 80% Enqueue, 20% Dequeue

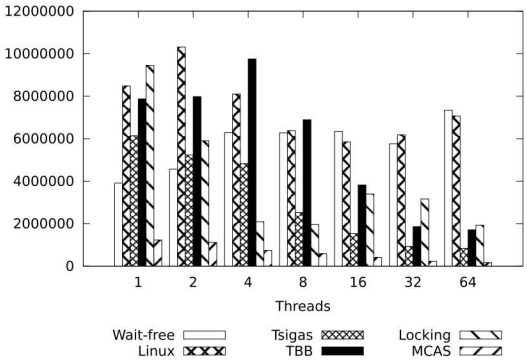

Figure 5: 20% Enqueue, 80% Dequeue

10.4 80%/20% Distributions

This scenario was chosen to examine the performance when each buffer experiences a greater rate of one operation type. Figures 4 and 5 show results for execution at 80% of one operation and 20% of the other in which each buffer was initially filled to 50% capacity.

At a higher enqueue rate, our buffer outperforms the alternative designs by an average of 16% or greater. This improvement is drastically increased as thread count increases, exhibiting the designs scalability.

On average, when the dequeue rate is 80% our design outperformed the Tsigas, Locking, and MCAS approcaches each by at least 45% excluding TBB in which improvement was 2%. The average throughput fell behind the Linux Buffer by 18% mostly due to the design performing more operations for the lower thread counts. However, the improvement of the design's dequeue rate is drastically increased as thread count increases, averaging at least 280% improvement with 64 threads excluding the Linux Buffer in which improvement was 4%.

Contention is further reduced in these tests due to the separation of operations on the head and the tail. The combination of the results provided in these tests and those in Section 10.3, show that the algorithm is efficient at performing enqueue operations.

10.5 50%/50% Distributions

This test provides a metric of performance when each buffer experiences an equivalent rate of enqueue and dequeue operations. Contents of the buffer were filled to 50% capacity. Our design outperforms the other ring buffers with an average improvement of 46% or greater excluding the Linux Buffer in which improvement was 1%.

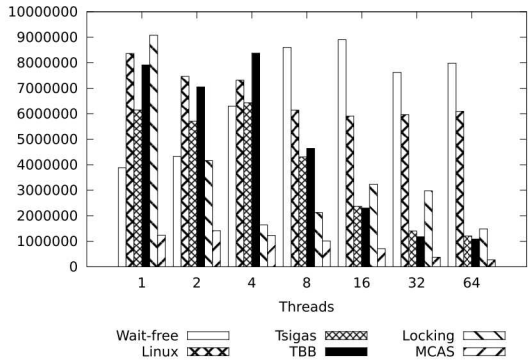

Figure 6: 50% Enqueue, 50% Dequeue

11. CONCLUSION

This paper presents the *first* wait-free ring-buffer suitable for parallel modification by many threads. This design is shown to be linearizable for all operations regardless of complexity. Using defined linearization points, we have demonstrated that the buffer stores and removes elements in a first in, first out manner.

The presented design introduces a method to relieve contention using a combination of atomic operations, sequence values, and strategic bitmarking. Though our approach is not the first design to use sequence counters, it is the only design that is able to do so without the risk of live-lock and thread starvation. We maintain the first-in-first-out ordering of the ring buffer by assigning sequence values used as indices. Supplementing this is an introduction of strategic bitmarking to mark locations for correction which would otherwise invalidate this ordering. Additionally, the use of a progress assurance scheme guarantees that each thread completes its operation in a finite number of steps. Using these methods, we streamline thread operations with minimal hindrance to concurrent operations.

In a majority of tested scenarios, our design performs operations faster than the best alternative approach. Additionally, the ring buffer is capable of supporting greater amounts of parallelism than any design to which it was compared except in some cases the lock-free Linux Buffer. On average, our buffer provides 15% improvement over the industry standard, TBB. Under contention, the design is capable of 300% performance increase over TBB. When compared to other non-blocking designs, our wait-free algorithm provides an average of at least 10% improvement.

12. ACKNOWLEDGEMENT

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

13. REFERENCES

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

[2] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[3] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

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

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

[6] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[7] Philippas Tsigas and Yi 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.