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

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

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

ABSTRACT

The ring buffer is a staple data structure used in many algorithms and applications. It is highly desirable in high-demand use cases such as multimedia, network routing, and trading systems. This work presents a new ring buffer design that is, to the best of our knowledge, the only array-based first-in-first-out ring buffer to provide wait-freedom. Wait-freedom guarantees that each thread completes its operation within a finite number of steps. This property is desirable for real-time and mission critical systems.

This work is an extension and refinement of our earlier work. We have improved and expanded algorithm descriptions and pseudo-code, and we have performed all new performance evaluations.

In contrast to other concurrent ring buffer designs, our implementation includes new methodology to prevent thread starvation and livelock from occurring.

CCS Concepts

•Theory of computation → Shared memory algorithms;

Keywords

concurrent, parallel, non-blocking, wait-free, ring buffer

1. INTRODUCTION

A ring buffer, or cyclical queue, is a first-in-first-out(FIFO) queue that stores a finite number of elements, usually on a fixed-length array. This design enables operations to complete in O(1) complexity and allows for cache-aware optimizations. Additionally, its low memory utilization makes it highly desirable for systems with limited memory.

The rise in many-core architecture has resulted in an increase in applications that are parallelizing their workloads (e.g., cloud-based services). Often buffers are used to pass work from one thread to another, and increasingly these buffers have become the source of bottlenecks.

Existing research has forgone the use of coarse-grained lock, and it has achieved greater scalability and core utilization by using fine-grained and/or non-blocking designs. Non-blocking algorithms are a class of algorithms that focus on providing guarantees of progress, which imply freedom from different concurrency dangers. The strongest form of non-blocking algorithms are wait-free, which guarantees that each operation completes at a finite number of steps. Wait-free algorithms are immune to deadlock, livelock, and thread starvation [6].

Deadlock occurs when there is a cyclic dependency between one or more operations. This dependency causes them to wait for one another to finish, thus blocking all operations involved. In livelock, operations are continually yielding to one another instead of making progress with their own operations. Thread starvation occurs when an operation waits indefinitely for a resource or is continually denied a resource.

Weaker forms include lock-free and obstruction-free, which are only free from a subset of those dangers. Lock-free algorithms guarantee that some thread is always making progress in its operation. Obstruction-free guarantees that if all threads are suspended, there exists a thread that if resumed and executed in isolation, then it can complete its operation.

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 their operation [11]. Section 7 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* (*faa*) operation [8], but unfortunately the design is susceptible to thread starvation and is not thread death safe.

To address these limitations we propose a new wait-free ring buffer algorithm based on the atomic *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. This *seqid* is also stored within the value type that is placed on the buffer. 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 in which a thread is continually prevented from completing its operation, we integrated a progress assurance scheme [2].

Previous research has shown that the use of the *faa* operation can provide significant increase in performance when compared to similar designs implemented using the atomic compare-and-swap (*cas*). For example, Feldman et al. [3] compared the performance of a *faa*-based vector push back 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 wait-free ring buffer design that uses a multiword-compare-and-swap (mcas) [1] to enqueue and dequeue values. This allows us to compare the performance difference between our *faa*-based approach to that of a *cas*-based approach. Section 7 provides a detailed analysis of the performance difference between these approaches.

This work presents the following novel contributions:

- This is the first array-based wait-free ring buffer. Other known approaches are susceptible to hazards such as livelock and thread starvation.
- Our design presents a new way of applying sequence numbers and bitmarking to maintain the FIFO property. This allows for 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 our implementation more suitable for highly parallel environments.

2. ALGORITHM DESIGN

This section describes how threads perform the *enqueue* and *dequeue* operations. These operations were implemented using the open-source *Tervel* wait-free algorithm framework [2]. This framework provides memory management constructs, inter-thread helping techniques, and a progress assurance scheme. Progress assurance enables a thread to detect when it has been delayed by other threads and recruit those threads to aid in its operations.

Figure 1 presents the ring buffer class, its public member functions, and its private member variables.

Table 1 provides an overview of several utility functions used in the presented algorithms. Their full implementations have been omitted for brevity, but their names are representative of their behavior and we clarify their behavior in text. A full and annotated implementation is available in the latest release of the *Tervel* library [2].

The rest of this section is organized as follows: Section 2.1 presents the restrictions and limitations of our design. Section 2.2 presents an overview of our approach and the ring buffer class object. Sections 2.3 and 2.4 present high-level descriptions of the enqueue and dequeue opetations. Sections 2.5 and 2.6 expand upon the high-level descriptions using implementation examples.

```
template<typename T>
Class RingBuffer
    //member functions
    bool isFull();
    bool isEmpty();
    bool enqueue(T v);
    bool dequeue(T &v);
    //member variables
    const int64_t capacity_;
    std::atomic<int64_t> head_ 0;
    std::atomic<int64_t> tail_ 0;
    std::unique_ptr<std::atomic<uintptr_t>[]> array_;
end Class
```

Figure 1: Ring Buffer Class.

- valueType: a pointer to a type T object
- emptyType: a pointer sized integer indicating that a position does not hold a value.
- nextHead(): returns buffer.tail_.fetch_add()
- nextTail(): returns buffer.head_.fetch_add()
- getPos(int v): returns v%buffer.capacity_
- backoff(int pos, T& val): performs a user-defined backoff procedure, then loads the value held at 'pos' and returns whether or not it equals val
- DelayMarkValue(T *val): returns 'val' with a delay mark
- getInfo(T& val, int64_t& val_seqid, bool& val_isValueType, bool& val_isDelayedMarked): based on 'val', this function assigns values to the other passed variables.

Table 1: Utility Functions

2.1 Design Restrictions

Our approach requires support for the following atomic primitives: *Compare-and-Swap* (*CAS*), *Fetch and Add* (*FAA*), *Load*, and *Store*. These atomic primitives are supported by the majority of modern hardware architectures and provided in C++11 [7].

It also reserves the three least significant bits of a value for encoding state and type information.

Another constraint is that elements stored within the buffer each contain an integer for storing a sequence identifier. This sequence identifier is used to ensure that the buffer provides a first-in-first-out ordering of elements. In the presented implementation, a pointer to the data is stored in the buffer instead of the data itself. Storing the data on the buffer is possible using techniques described in [4].

2.2 Design Overview

The design of our ring buffer diffuses contention by assigning sequence numbers (seqid) to each thread to use to complete its operation. The seqid is assigned by performing a *faa* on the head counter for the *dequeue* operation and on the tail counter for the *enqueue* operation.

After receiving a seqid, a thread examines the value held at a position determined by the seqid and acts based on the value's seqid, type, and whether or not it is marked as delayed. If the least significant bit is set to zero, its type is a pointer to a *valueType*, otherwise its type is an *emptyType*. A *valueType* is a pointer to an object containing a value that has been enqueued, and an *emptyType* indicates a value is not presently enqueued at the position. The second least significant bit indicates whether or not the value is marked as delayed. The seqid of an *emptyType* is represented by the unreserved bits of the value, and the seqid of a *valueType* is stored within the object. The *getInfo* function is used to extract this information from the value.

We make the restriction that a *valueType* can only be removed by a dequeuer that is assigned a seqid that is equal to that *valueType*'s seqid.

Due to the nature of the ring buffer, the actions or inactions of other threads may cause a thread to livelock by indefinitely reattempting its operation. We prevent this scenario from occurring by using the progress assurance scheme described in [2].

2.3 Enqueue Overview

A thread performing enqueue loads the value at the specified position and acts based on the information returned by the *getInfo* operation. Ideally, this value will be an *emptyType* that has a seqid that matches the thread's seqid and is not marked as delayed. In this case, the thread attempts to replace the value with a *valueType* that contains the thread's seqid. If the thread successfully replaces the value it returns true, otherwise the thread re-loads the value at the specified position and acts based on that value.

If the current value's seqid is less than the thread's seqid, but still an *emptyType* and not marked as delayed, the thread will perform a back-off routine to allow a delayed thread to complete its operation, and if the value has not changed, the thread will attempt to replace it.

If the current value is marked as delayed or it is a *valueType*, it will also cause the thread to perform a back-off routine, but in this case, if the value has not changed, the thread will get a new seqid and restart its operation. Similarly, if the current value's seqid is greater than the thread's seqid, this will also cause the thread to get a new seqid and start over.

If the thread successfully replaced a value, it will return, otherwise it will retry part or all of its operation.

2.4 Dequeue Overview

A thread performing dequeue loads the value at the specified position and acts based on the information returned by the *getInfo* operation. In the ideal case, this value will be a *valueType* whose seqid matches the thread's seqid. In this case the thread will attempt to replace it with an *emptyType* containing the thread's seqid plus the capacity of the buffer. If the current value was marked as delayed, the replaced value will also be marked as delayed. If the replacement was not successful, it implies that it failed because the current value became marked as delayed. The thread will re-attempt its replacement and this time it is guaranteed to be successful because we make the restriction that only the thread assigned the seqid of a value may remove that value. Since no other threads will attempt to remove it and it is already marked as delayed, the value can not be changed by another thread.

If the current value's seqid is greater than the thread's seqid, this means no value was enqueued with this seqid and the thread needs to get a new seqid and start over.

In the event of thread delay, the current value's seqid could be less than the thread's seqid or the current value maybe an *emptyType*. In this case the back-off routine is used to give the delayed thread a chance to complete its operation. If it has not made progress upon the routine's return then one of the following steps is taken to ensure progress of other threads.

- If the value is an *emptyType* with a delay mark, it is replaced by an *emptyType* that has a seqid larger than the current tail counter.
- If the value is an *emptyType*, it is replaced by the same *emptyType* described in the ideal case at the start of this section.
- If the value is a *valueType*, then it is marked as delayed.

The specific reasoning for this logic is explored in the following two sections. In short, it is designed to allow threads to skip positions without losing the first-in-first-out property of the buffer.

2.5 Enqueue Implementation

Figure 2 presents the enqueue operation as implemented in the Tervel library.

Lines 2 to 4 and 25 to 28 are part of the progress assurance scheme used to provide wait-freedom. To prevent a thread from perpetually delaying another thread, we include a call to the *check_for_announcement* function, which may cause the calling thread to help one other thread. The *Limit* class is used to detect when a thread has been delayed, calling *notDelayed* increments a counter by the passed value, and it returns false when a user-specified limit has been reached. In the event a thread determines it has been delayed, the logic starting at line 25 enables the thread to recruit other threads to complete its operation. This code is further discussed in Section 4.

After getting a seqid and position to operate on, a thread loads the value currently stored at position(Line 10) and then acts based on that value. The *readValue* function may return false in the event it is unsafe to dereference the current value(Section 4). It extracts the necessary information form the value by calling the *getInfo* function(Line 13).

First, it checks to see if its seqid has been skipped(Line 14) and if it has, the thread gets a new seqid. Then it checks to see if the value is marked as delayed and if so the thread performs a back-off routine(Line 17). If the back-off was successful, the value at the specified position has changed and a

```
1: function BOOL ENQUEUE(T v)
2:     check_for_announcement();
3:     Limit progAssur;
4:     while progAssur.notDelayed(0) do
5:         if isFull() then return false;
6:         int64_t seqid = nextTail();
7:         uint64_t pos = getPos(seqid);
8:         uintptr_t val;
9:         while progAssur.notDelayed(1) do
10:            if !readValue(pos, val)) then continue;
11:            int64_t val_seqid;
12:            bool val_isValueType, val_isDelayedMarked;
13:            getInfo(val, val_seqid, val_isValueType, val_isDelayedMarked);
14:            if val_seqid > seqid then
15:                break;
16:            if val_isDelayedMarked then
17:                if backoff(pos, val) then continue;
18:                else break;
19:            else if !val_isValueType then
20:                if val_seqid < seqid && backoff(pos, val) then continue;
21:                uintptr_t new_value = ValueType(value, seqid);
22:                if array_[pos].cas(val, new_value) then
23:                    return true;
24:            else if !backoff(pos, val) then break;
25:    EnqueueOp *op = new EnqueueOp(this, value);
26:    make_announcement(op);
27:    bool res = op->result();
28:    op->safe_delete();
29:    return res;
```

Figure 2: Enqueue Operation

```
1: function BOOL DEQUEUE(T &v)
2:     check_for_announcement();
3:     Limit progAssur;
4:     while progAssur.isDelayed(0) do
5:         if isEmpty() then return false;
6:         int64_t seqid = nextHead();
7:         uint64_t pos = getPos(seqid);
8:         uintptr_t val, new_value = EmptyType(nextSeqId(seqid));
9:         while progAssur.isDelayed(1) do
10:            if !readValue(pos, val) then continue;
11:            int64_t val_seqid;
12:            bool val_isValueType, val_isDelayedMarked;
13:            getInfo(val, val_seqid, val_isValueType, val_isDelayedMarked);
14:            if val_seqid > seqid then break;
15:            if val_isValueType then
16:                if val_seqid == seqid then
17:                    if val_isDelayedMarked then
18:                        new_value = DelayMarkValue(new_value);
19:                    value = getValueType(val);
20:                    if !array_[pos].cas(val, new_value) then
21:                        new_value = DelayMarkValue(new_value);
22:                        array_[pos].cas(val, new_value);
23:                    return true;
24:                else
25:                    if !backoff(pos, val) then
26:                        if val_isDelayedMarked then
27:                            break;
28:                        else atomic_delay_mark(pos);
29:            else
30:                if val_isDelayedMarked then
31:                    int64_t cur_head = getHead();
32:                    int64_t temp_pos = getPos(cur_head);
33:                    cur_head += 2*capacity_;
34:                    cur_head += pos - temp_pos;
35:                    array_[pos].cas(val, EmptyType(cur_head));
36:                else if !backoff(pos, val) && array_[pos].cas(val, new_value) then break;
37:    DequeueOp *op = new DequeueOp(this);
38:    make_announcement(op);
39:    bool res = op->result(value);
40:    op->safe_delete();
41:    return res;
```

Figure 3: Dequeue Operation

thread will reprocess it, otherwise the thread will get a new seqid. It is safe to get a new seqid as we allow for both enqueue and dequeue operations to replace *emptyType* objects that have seqid less than expected. A dequeuer assigned a skipped seqid would simply get a new seqid.

The next check determines if the value is an *emptyType* or *valueType*(Line 19). If it is an *emptyType* with a matching seqid, the thread attempts to replace it(Line 22). If it is an *emptyType* with a seqid less than the thread's seqid, the thread performs a back-off routine before attempting to replace the value. By performing a back-off routine, it provides a slow executing thread with the opportunity to complete its operation. If the thread was successful at replacing the value it will return true, otherwise, it will get the new current value.

If the value is a *valueType*, the thread will perform a back-off routine, which returns whether or not the value at passed position has changed. If it has not changed, the thread will get a new seqid, otherwise the thread acts based on that value.

2.6 Dequeue Implementation

Figure 3 presents the dequeue operation as implemented in the Tervel library.

After getting a seqid, the thread determines the position to operate on and loads the value currently stored at that position(Line 10). The *readValue* function may return false in the event it is unsafe dereference the current value(Section 4). The thread will then extract the necessary information form the value(Line 13) and act accordingly.

First, it ensures that its seqid has not been skipped(Line 14) and if it has, the thread will get a new seqid. Next it checks if it is a *valueType* and if so compares its seqid to the current value's seqid. If the seqid matches, then the thread replaces that value with an *emptyType* containing the next seqid for that position(Line 20 or 22). If the current value was marked as delayed, the value replacing it will also have that mark. Section 3 discusses the importance of this step.

If the value is a *valueType* whose seqid is less than the thread's seqid, the back-off routine will be used to give a delayed thread a chance to complete its operation. If the back-off routine was unsuccessful, the thread will get a new seqid if the value is marked as delayed. Otherwise, the thread will place a delay mark on the value and then re-process the value(Lines 25- 28). This reprocessing is important to handle the case where the value changes just before placing the delay mark.

If the current value is an *emptyType* that is marked as delayed, the thread will attempt to replace it with an *emptyType* that has a seqid that has not been assigned(Lines 30-35). If the *emptyType* is not marked as delayed, the thread will perform the back-off routine and if the value has not changed it will try to skip the position(Line 36).

3. CORRECTNESS

In exploring the correctness of the *enqueue* and *dequeue* operations, we want to ensure that the effects of an operation occur exactly once, that the return result of the operation accurately describes how the buffer was modified, and that the operations are linearizable to one another. To support this, we show that a valid sequential history can be constructed from any valid concurrent history(*Linearizability*).

When constructing this valid sequential history, concurrent operations are ordered by their linearization points. The enqueue operation's linearization point occurs when the cas at Line 22 returns true. The dequeue operation's linearization point occurs when the cas at Line 20 or 22 returns true.

Showing that the effects of an operation occur once and that they return the correct value can be proven by examining the two algorithms. The enqueue operation returns false when the buffer is full and similarly the dequeue operation returns false when the buffer is empty. They return true after successfully replacing a value in the buffer with the effect of their operation. In this regard they can not return incorrectly.

To show that each concurrent history corresponds to a valid sequential history we have to ensure that elements are removed in a first-in-first-out order. We first show that this holds for buffers of infinite length and then we show it also holds for buffers of finite length.

For an infinite length buffer, concurrent enqueues are ordered by a monotonically increasing counter and concurrent dequeue operations are ordered by a separate but also monotonically increasing counter. We use the seqid returned from these two counters to impose a FIFO ordering on the operations. This is accomplished by initializing the positions on the buffer with *emptyType* values and upon enqueue replacing the value with a *valueType* containing the same seqid. By the nature of two monotonically increasing counters, elements must be removed in FIFO order.

If the capacity of the buffer is finite, positions on the buffer must be reused, which allows for two or more threads to attempt to apply their operations at the same position. Further, the algorithm must also correctly handle the case where an enqueuer is delayed and has not written its value before a dequeuer attempts to remove that value.

We are able to prevent these scenarios from leading to incorrect behavior by leveraging the seqid to enforce first-in-first-out ordering of actions.

From Figures 2 and 3 we extracted the following valid changes to a value at a position on the ring buffer's internal array.

- Any value may become delay marked.
- A thread will only attempt to replace a *valueType* that does not have a delay mark with an *emptyType* with a seqid equal to the seqid of the *valueType* plus the capacity of the buffer.
- A *valueType* that has a delay mark can only be replaced by an *emptyType* with a delay mark and a seqid equal to the seqid of the *valueType* plus the capacity of the buffer.
- A thread will only attempt to replace an *emptyType* that does not have a delay mark with either an *emptyType* with a higher seqid or a *valueType* with a seqid greater than or equal to its seqid.
- A thread will only attempt to replace an *emptyType* that has a delay mark with an *emptyType* with a seqid greater than the value of the current tail counter.

Using these valid transitions, we derived the following rules, that ensure FIFO behavior of the buffer.

- A value with a lower seqid can not replace a value with a higher seqid.
- An element is enqueued by replacing an *emptyType* whose seqid is less than or equal to the elements seqid.
- An element can only be removed from the buffer by a thread performing a dequeue operation that has been assigned a seqid that matches the seqid stored within the element.
- A dequeue thread can only get a new seqid if it can guarantee that an element has not and will not be enqueued with that seqid.

Together these valid transitions and rules show that our algorithms behave as expected and operate correctly. Our implementation contains numerous state checks designed to detect incorrect behavior and our test handler contains logic to detect if the values are dequeued multiple times or out of order. In the numerous tests and experiments performed by our implementation, we were unable to detect any errors or anomalousness behavior that was not a result of correctable implementation error.

4. PROGRESS ASSURANCE

This section describes how we integrated *Tervel's* progress assurance scheme [2] into our design to guard against possible scenarios of livelock and thread starvation. We used the following *Tervel* constructs explicitly within the aforementioned algorithm:

- Limit: This object encapsulates logic related to detecting when a thread has been delayed beyond a number of attempts defined by the user.
- check_for_announcement: Every *checkDelay*[1] number of operations, this function tries to help one thread, if that thread has made an announcement. After making $checkDelay * numThreads$ calls to this function, a thread will have attempted to help every other thread.
- make_announcement: This function is used to recruit threads to help the calling thread complete an operation. After making an announcement, only $checkDelay * numThreads^2$ more operations may complete before it is guaranteed for the operation described in the announcement to be completed.

To enable a thread to safely recruit other threads to help complete its operation, we implemented specialized versions of the enqueue and dequeue operations encapsulated within *EnqueueOp* and *DequeueOp* class objects. The following two sections present a brief description of these classes and the differences between them and the enqueue and dequeue operations presented in Section 2. Full implementation details are available in the documentation of Tervel [10].

4.1 Overview of Operation Record

The ring buffer's operation record uses Tervel's association model to enable a thread to temporarily place a reference to a helper descriptor object (*Helper*) at a position on the buffer. If the *Helper* object is associated with the operation record, it is replaced replaced by the result of the operation. Otherwise, it is replaced by the value that the *Helper* had replaced.

An operation record is associated with a *Helper* object if that operation record's atomic reference was changed from *nullptr* to the address of the *Helper* object. By design once this reference has been set, it can not be changed again.

If the *readValue* function loads a reference to a *Helper* object, the thread executing the function will attempt to acquire hazard pointer protection[2] on that object, which causes the object's *on_watch* function to be called. We implemented the *Helper's on_watch* function to cause the *Helper* object to be replaced by its logical value and return false. This prevents the needed to add additional logic to earlier algorithms. As a result, when the *readValue* function loads a *Helper* value, it will fail to acquire memory protection and return false.

[1]A user defined constant

[2]Hazard pointer protection is a memory protection technique used to prevent objects from being freed by one thread whilst being used by another. If memory protection was not used, then the algorithm would be forced waste memory or risk entering a corrupted state. A thread acquires hazard pointer protection on an object by writing the address of an object to a shared table and then examining the address the reference to the object was loaded from. If the value of the address has changed, the thread will often re-attempt part of its operation. If it has not changed, the thread has successfully acquired memory protection on the object [5].

Tervel's operation record classes require the implementation of the function *help_complete* which is called by threads recruited to help complete an operation. Our implementation of this function mirrors the implementation of the enqueue and dequeue operations, with a few key changes. First, the looping structure terminates when the operation record becomes associated with a *Helper* object. In the event the buffer is full (for enqueue) or empty (for dequeue), the operation record will become associated with a constant value indicating such.

The following sections present summaries of the design of these operation records. For complete implementation details with in-line comments, please see the latest release of the Tervel library.

4.2 Enqueue Operation Record

The enqueue operation record's *help_complete* function begins by setting its seqid to the current value of the tail counter, and then examining the value stored at the position indicated by that value. If the value is not a non-delay marked *emptyType* with a *seqid* less than or equal to its seqid, the thread will increment its seqid and try again. Otherwise, it will attempt to place a reference to a *Helper* object, and if unsuccessful it will re-examine the value at the position.

After successfully placing the reference to a *Helper* object, the *Helper's finish* function is called to remove it and replace it with its logical value. This function is also called by the *Helper's on_watch* function, which always returns false because the *finish* function removes the reference to the helper object. This design pattern removes the need for other operations to include logic to resolve *Helper* objects.

After removing the *Helper* object, the last step is to repeatedly try to update the tail counter until it is greater than or equal to seqid used to enqueue the element. This ensures that *isEmpty* does not report falsely. The loop used to implement this will terminate after a finite number of iterations, since each failed attempt implies the counter has increased in value and has gotten closer to the seqid.

To ensure that the enqueued element has the correct seqid, before beginning the operation the element's seqid is set to a negative value. Then, before the *Helper* object is removed a *cas* operation is used to change it to the seqid stored within the *Helper* object.

4.3 Dequeue Operation Record

This dequeue operation permits threads to dequeue a value without modifying the head counter. A thread assigned the seqid of a value that was dequeued in this manner will act as if the seqid was never used to enqueue a value. The dequeue operation record's *help_complete* function begins by setting its seqid to the current value of the head counter and then examining the value stored at the position indicated by that value.

If the value's seqid is greater than thread's seqid, the thread will increment its seqid and examine the next position.

If the value is an *emptyType* and delay marked, it will be handled as described in Section 2.6. If it is not delay marked and it has a seqid less than the thread's seqid, the thread will attempt to change the position to an *emptyType* with the next seqid. If the thread is successful, it will increment its seqid and examine the next position, otherwise it will re-examine the position. This ensures that values will not be enqueued after we passed this position, which may affect the FIFO property.

If the value is a *valueType* with a seqid less than or equal to the thread's seqid and greater than or equal to a minimum seqid specified in the operation record, the thread will use a *Helper* object to attempt to dequeue the value, re-examining the position if it is unsuccessful. The minimum value is important to ensure the FIFO property, otherwise a value assigned to a thread that has been delayed for an extended period maybe returned. If the seqid was not within the proper range, the thread will increment its seqid and examine the next position.

After successfully placing the reference to a *Helper* object, the *Helper's finish* function is called to remove it and replace it with its logical value. This function is also called by the *Helper's on_watch* function, which always returns false because the *finish* function removes the reference to the *Helper* object. This design pattern prevents other algorithms from perceiving the *Helper* object.

After removing the *Helper* object, the last step is to repeatedly try to update the head counter until its value is greater than or equal to the seqid used to dequeue the element. This ensures that *isFull* does not report falsely. The loop used to implement this will terminate after a finite number of iterations, since each failed attempt implies the counter has increased in value and has gotten closer to the seqid.

5. WAIT-FREEDOM

To show that the *enqueue* and *dequeue* operations are wait-free we examine the function calls and the looping structures used in constructing them.

Each function called by these operations, except for the two *Tervel* operations, *check_for_announcement* and *make_announcement*, are simple utility functions that contain no loops or calls to other functions. The looping structures used in both algorithms terminate when the call to *notDelayed* returns false. The *Tervel* library specifies that this function returns false when the a counter in the *progAssur* reaches 0. This counter, which is initially set equal to a compile-time constant (*delayLimit*), is decremented by an amount specified in the call to *notDelayed*. As a result these loops terminate after *delayLimit* iterations of the inner loop, which is called at least once for each iteration of the outer loop.

If *check_for_announcement* and *make_announcement* are wait-free, then the *Enqueue* and *Dequeue* operations are also wait-free since it can be shown that they terminate after a finite number of instructions. Both of these functions may call the *help_complete* function of an operation record class. Because of this, if each operation record's *help_complete* function is lock-free then these operation's are also wait-free.

The reason the requirement is lock-free stems from how *Tervel* recruits threads to help complete delayed operations. For example, if an operation is delayed long enough, then all threads will eventually try to complete it, and no new operations can be created until it is complete. In this case, the lock-free property ensures that one of the thread's will complete the operation, causing all threads to return.

In Section 4, we discuss the design of the operation records used to help complete a delayed thread's enqueue or dequeue operation. The *help_complete* function in the two objects is a modified version of the algorithms presented in Figures 2 and 3. Specifically, we changed the terminating condition of the while loops to terminate when the operation record becomes associated with a helper object. Since these are lock-free functions and there is a finite number of new operations before all threads are executing the *help_complete* function of a particular operation record, the *help_complete* function of the *EnqueueOp* and *DequeueOp* classes must be wait-free.

6. RELATED WORKS

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

6.1 Lock-free Ring Buffer by Tsigas et al.

Tsigas et al. [11] presents 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 enqueueing 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 capable of diffusing contention, the competitive nature of this design leads to increased contention and poor scaling.

6.2 Krizhanovsky Ring Buffer

Krizhanovsky [8] describes a ring buffer that he claims to be lock-free. It relies on the *fetch-and-add* operation to increment head and tail counters, which determine the index to perform an operation.

Each thread maintains a pair of local variables, local_head and local_tail, to store the indexes where the thread last completed a enqueue or dequeue operation. The smallest of all the threads' local_head and local_tail values is 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.

Though the author claims it to be lock-free, if a thread fails to update its local variables, it will prevent other threads form making progress, even if the buffer is neither full or empty. As a result, this design is actually obstruction-free because it can enter a state where all threads are waiting on a single thread to perform an action.

6.3 Intel Thread Building Blocks

Intel Thread Building Blocks (TBB) [9] 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.

6.4 MCAS Ring Buffer

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 software-based Multi-Word Compare-and-Swap (MCAS) [1]. MCAS is an algorithm used to conditionally replaces the value at each address in a set of address. An MCAS operation is successful and subsequently performs this replacement if the value at each address matches an expected value for that address. To provide correctness this must appear to happen atomically, such that overlapping operations cannot read a new value at one address and then read an old value at another address.

In our MCAS-based ring buffer design, we use MCAS to move a head and tail markers along the buffer's internal array. The act of moving a marker requires the completion of six hardware compare-and-swap operations. Though this implementation is simple to implement and reason about, the number of compare-and-swap operations necessary to complete a single operation results in poor permanence and scalability.

The specific procedure to enqueue an element is as follows: after identifying the position holding the head marker, a thread enqueues an element by performing a successful MCAS operation that replaces the head marker with the value being enqueued and replaces an empty value at the next position with the head marker. If the next position holds the tail marker, it indicates that the buffer is full.

Similarly, a value is dequeued by a thread using 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 is empty.

6.5 Comparison

In contrast to the design presented by Tsigas et al., where threads compete to update a memory location, our design diffuses thread congestion by incrementing the head or tail value and using the pre-incremented value to determine where to apply an operation. By reducing the amount of threads attempting a *cas* operation at a given buffer location, our approach is able to reduce the amount of failed *cas* operations. This is similar to the approach presented by Krizhanovsky, however, in contrast Krizhanovsky's design, we have devised methodology to prevent threads from becoming livelocked or starved.

7. RESULTS

This section presents a series of experiments that compare the presented ring buffer design (WaitFree) to the alternative designs described in Section 6. Tests include performance results of TBB's concurrent bounded queue (TBB), Krizhanovsky's ring buffer (Krizhanovsky), Tsigas et al.'s cycle queue (Tsigas), and the MCAS-based design (MCAS). For each experiment we present the configuration of the test, the collected results, and performance analysis.

7.1 Testing Hardware

Tests were conducted on a 64-core ThinkMate RAX QS5-4410 server running the Ubuntu 12.04 LTS operating system. It is a NUMA architecture with 4 AMD Opteron 6272 CPUs, which contain 16 cores per chip with clockrate 2.1 GHz and 314 GB of shared memory. Executables were compiled with GNU's C++ compiler, g++-4.8.1 with level three optimizations and *NDEBUG* set. The algorithm implementations used for testing where the latest known versions from their respective authors.

7.2 Experimental Methodology

Our analysis examines how the number of threads and the type of operation executed by each thread impacts the total number of operations completed.

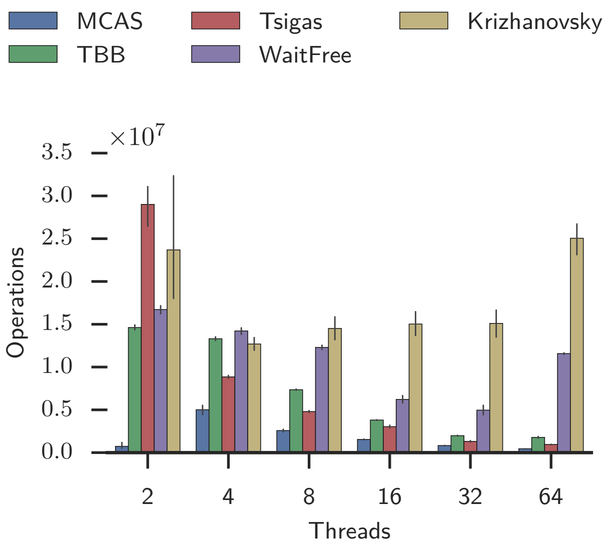
We use the *Tervel* library's synthetic application tester to simulate various use cases and configurations of the ring buffer implementations. The testing procedure consists of a main thread, which configures the testing environment and spawns the worker threads. For these tests, the queue was constructed with a fixed capacity of $32,768$ elements and then initialized by performing $16,384$ enqueues. The main thread signals the worker threads to begin execution, sleeps for 5 seconds, and then signals the end of the execution. Information gathered during the test is outputted to a log file.

The following graphs depict the total number of operations of all threads completed while the main thread was asleep. Each test configuration was executed twenty times, the standard deviation of these results is included within the graphs.

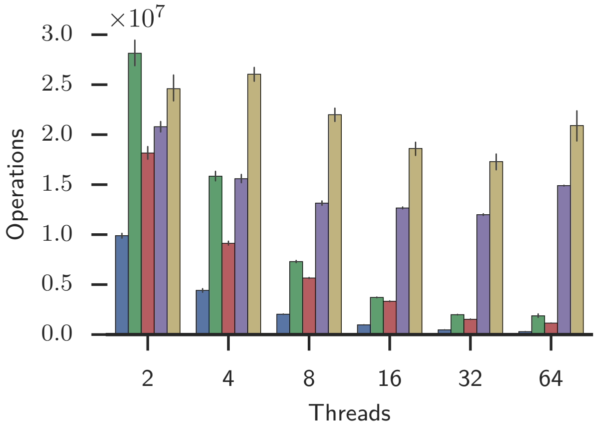
7.3 Even Enqueue and Dequeues

We first discuss experiments where the number of enqueue and dequeue operations are relatively equal. This can be accomplished in three different ways, each of which results in slightly different performance. Performance of a data structure, in this case, is the number of successful operations applied to the ring buffer during the test, unsuccessful operations occur when attempting to enqueue on a full buffer or dequeue on an empty buffer.

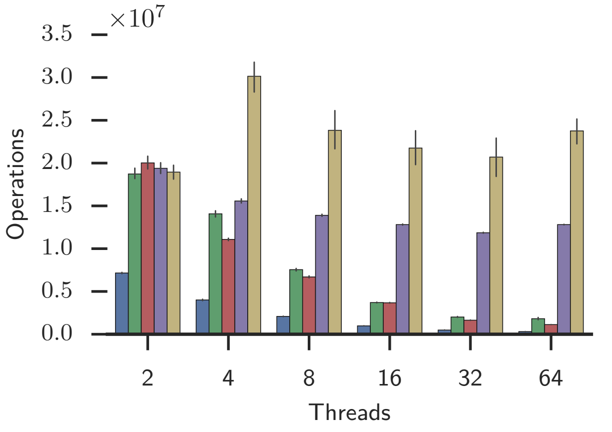
Figure 4a shows the performance of the first case, where half the threads perform enqueue and the other half perform dequeue. In this graph, the MCAS-based buffer (*MCAS*) performs significantly worse than the other designs at all thread levels. The presented wait-free ring buffer (*WaitFree*) performs better than the all but Tsigas et al.'s non-blocking buffer (*Tsigas*) at two threads and at 64 threads it outperforms all except for Krizhanovsky's buffer (*Krizhanovsky*).



(a) 50% of Threads Enqueue and 50% Dequeue



(b) Threads Enqueue then Dequeue



(c) Threads pick randomly, with even chance

Figure 4: Equal Enqueue and Dequeue

Developers using the Krizhanovsky buffer accept the risk that under certain conditions threads executing operations on the buffer may become blocked indefinably. The logic we include in our design to prevent this, however, it is the likely reason for this performance difference.

Figure 4b shows the performance in the case where threads alternate between enqueue and dequeue operations. In this figure, Krizhanovsky, and TBB perform significantly better at low thread levels, while the WaitFree version performs slightly worse. However, at high thread levels, the performance differences are similar to the previous figure.

Figure 4c shows the performance of the last case, where threads select randomly between the two operations, with each option having equal probability. The cost of calculating the random value creates slightly more work between calls to buffer operations. At low thread levels, each design (except for MCAS) performs equal well, however, at higher thread levels, the Krizhanovsky and WaitFree buffers still perform the best.

On average, at 64 threads our design performs 3825.84% more operations than the MCAS, 1117.52% more than Tsigas, 631.52% more than TBB, but 43.07% less than the Krizhanovsky buffer. Overall even enqueue and dequeue test scenarios, our design performs 1329.61% more operations than the MCAS, 339.51% more than Tsigas, 209.90% more than TBB, but 34.47% less than the Krizhanovsky buffer.

7.4 Higher Enqueue Rate

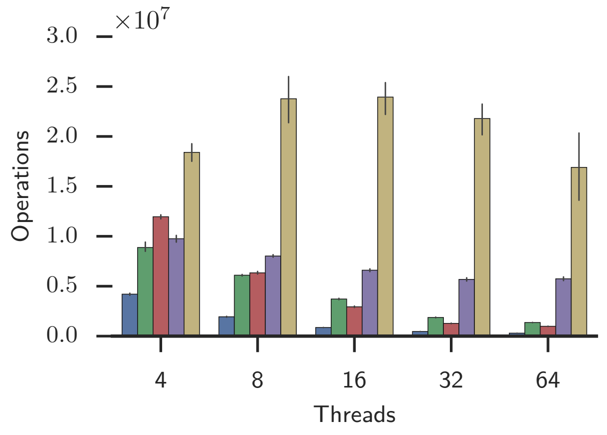
Figure 5 shows the performance of the algorithms when 75% of the threads perform enqueue and 25% of the threads perform dequeue. Figure 5a graphs the number of successful queue operations, Figure 5b graphs the number of failed queue operations, and Figure 5c graphs the total number of queue operations. Failed queue operations are the result of enqueue operations returning that the queue is full. Unlike other designs, threads executing on *Krizhanovsky* buffer blocks until they are able to apply their operations, as such they do not return failure.

If we only presented the total number of operations completed by each threads, then our results would not accurately reflect the work being completed. For example, *TBB* performs more operations than either *Krizhanovsky* or *WaitFree*, but nearly all of them are failed operations. Similarly about half of the *WaitFree* operations are failed. Having a significant number of failed operations indicates that the way the data structure is being used is not ideal. In this example, there is a three to one ratio of enqueue to dequeue operations and because of this, the queue becomes full rather quickly.

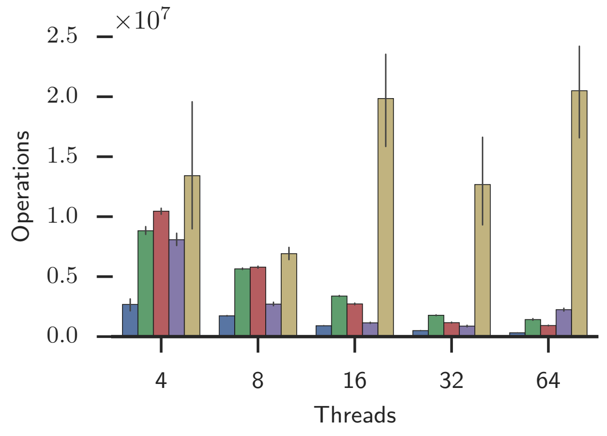
On average, at 64 threads our design performs 1835.45% more successful operations than the MCAS, 483.34% more than Tsigas, 321.58% more than TBB, and 66.02% less than the Krizhanovsky buffer. Overall test in this scenario, our design performs 813.27% more successful operations than the MCAS, 192.28% more than Tsigas, 128.95% more than TBB, and 64.12% less than the Krizhanovsky buffer.

7.5 Higher Dequeue Rate

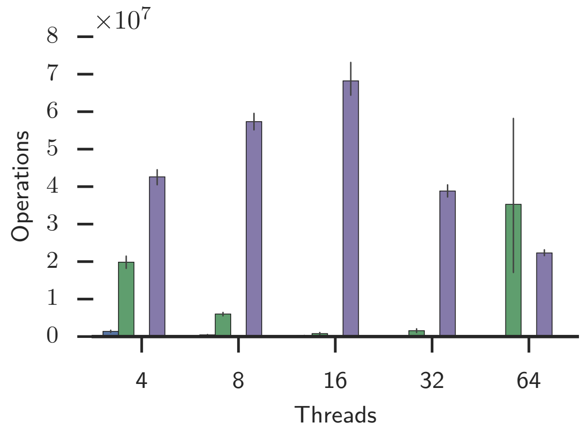
🟦 MCAS
🟩 TBB
🟥 Tsigas
🟪 WaitFree
🟨 Krizhanovsky



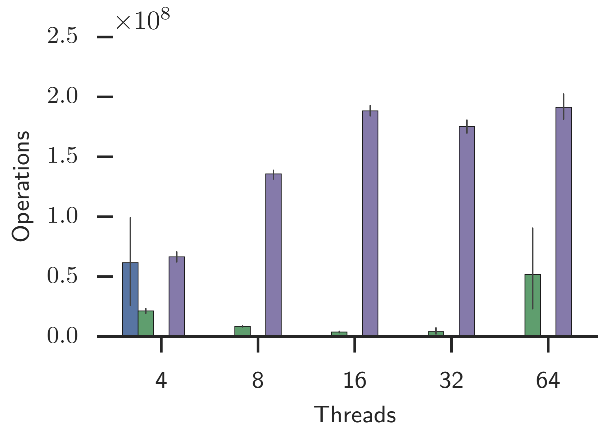
(a) Successful Operations



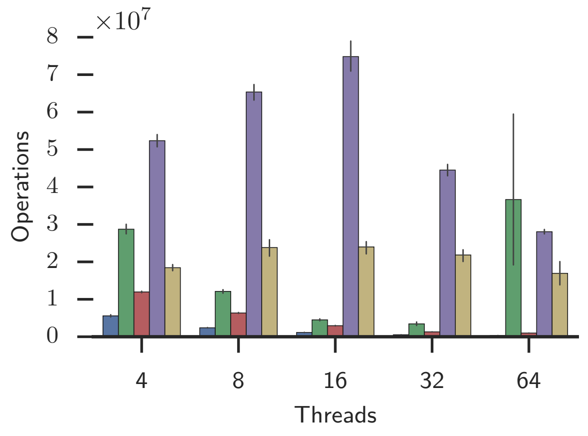
(a) Successful Operations



(b) Failed Operations

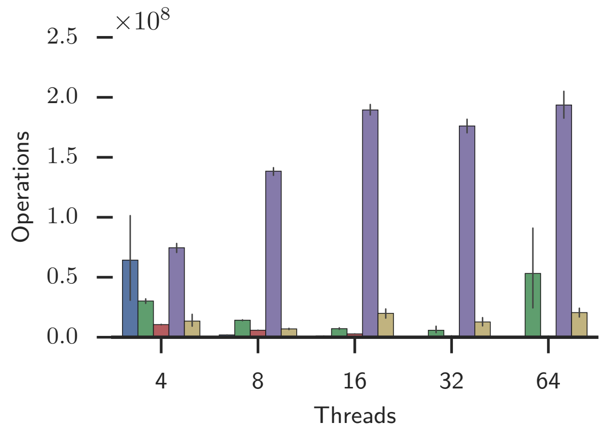


(b) Failed Operations



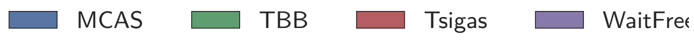
(c) Total Operations

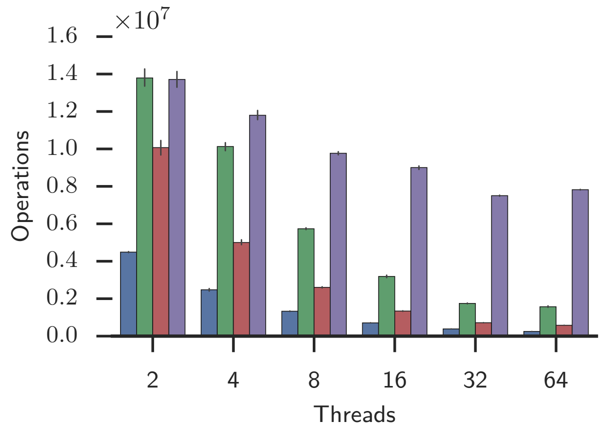
Figure 5: Use Case: High Enqueue



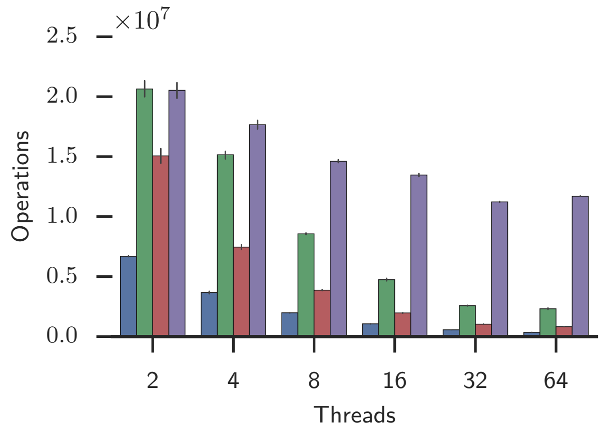
(c) Total Operations

Figure 6: Use Case: High Dequeue

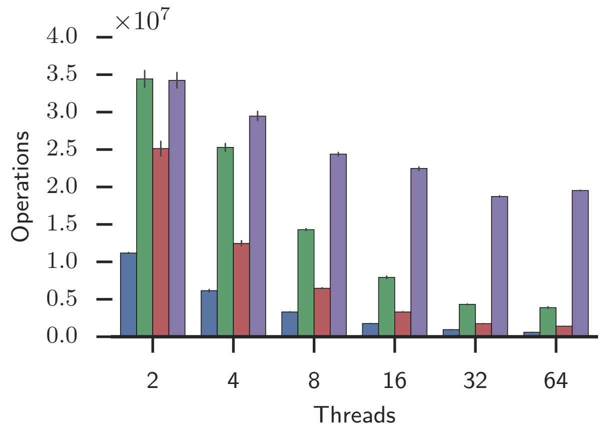




(a) **Successful Operations**

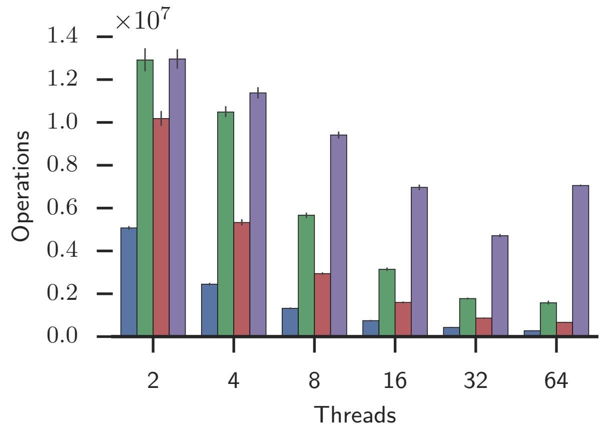


(b) **Failed Operations**

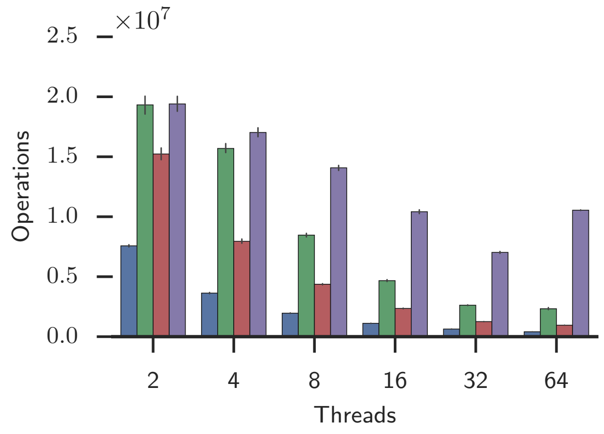


(c) **Total Operations**

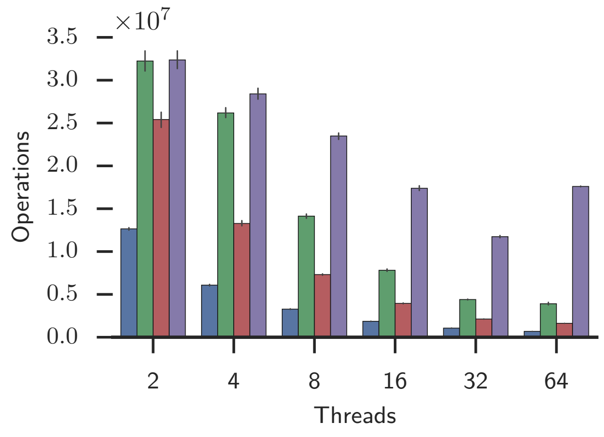
Figure 7: Use Case: 80% Enqueue, 20% Dequeue



(a) **Successful Operations**



(b) **Failed Operations**



(c) **Total Operations**

Figure 8: Use Case: 20% Enqueue, 80% Dequeue

Figure 6 shows the performance of the algorithms when 75% of threads perform dequeue and 25% of threads perform enqueue. Like in the previous section, Figure 6a graphs the number of successful queue operations, Figure 6b graphs the number of failed queue operations, and Figure 6c graphs the total number of queue operations. Failed queue operations are the result of dequeue operations returning that the queue is empty. As stated earlier, threads executing in the Krizhanovsky buffer block until they are able to apply their operation, as such they do not return failure.

In this example, the amount of failed operations is much higher than in the previous example. This indicates that the dequeue operation takes less time to complete then the enqueue operation, which causes the queue to become empty faster than the queue in the previous example to become full.

On average, at 64 threads our design performs 605.11% more successful operations than the MCAS, 143.28% more than Tsigas, 58.18% more than TBB, but performs 89.03% less than the Krizhanovsky buffer. Overall test in this scenario, our design performs 192.85% more operations than the MCAS, however it performs on average 3.05% less than Tsigas, 23.81% less than TBB, and 75.39% less than the Krizhanovsky buffer.

7.6 Further Analysis

We further analyze the performance by examining the performance of the data structures when each thread is capable of performing both enqueue and dequeue operations. Figure 7 presents the results from tests that have threads select enqueue with a 80% chance and dequeue with a 20% chance and Figure 8 presents results when there is a 20% chance of enqueue and 80% chance of dequeue. Section 7.3 presents the results when there is an even ratio of enqueue and dequeue operations. The Krizhanovsky buffer was not included in these results as it would become deadlocked when the buffer became empty and all threads are trying to dequeue or when the buffer became full and all threads are trying to enqueue.

Comparing these graphs to one another and to Figure 4c, we see that performance behavior between the different use cases is very similar. On average, at 64 threads our design performs 2746.50% more successful operations than MCAS, 372.47% more than Tsigas, and 1105.065% more than TBB. Overall test in this scenario, our design performs 943.05% more successful operations than the MCAS, 378.02% more than Tsigas, and 122.78% more than TBB.

8. CONCLUSION

This paper presents a wait-free ring-buffer suitable for parallel modification by many threads. We defined the linearization point of each operation and used it to argue that the design is linearizable and correct. We have demonstrated that our buffer provides a first-in-first-out ordering on elements and that its API is expressive enough for use in a concurrent environment.

The presented design introduces methodology to relieve thread contention by combining atomic operations, sequence counters, 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 livelock and thread starvation. We maintain the FIFO ordering of the ring buffer through the use of these sequence counters. Our design is supported by a strategic bitmarking scheme to correct scenarios that would otherwise invalidate the FIFO property. Lastly, we integrated a progress assurance scheme to guarantee that each thread completes its operation in a finite number of steps, thus achieving wait-freedom.

Our performance analysis showed that our design performs competitively in a variety of uses cases. Though it does not provide significant overall performance improvements, it provides stronger safety properties then other known buffers. Because of this, we believe that our design is practical for a variety of use cases that require strict safety properties and guarantees of progress.

9. ACKNOWLEDGMENT

This work is funded by National Science Foundation grants NSF ACI-1440530 and NSF CCF-1218100.

10. REFERENCES

[1] 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.

[2] S. Feldman, P. LaBorde, and D. Dechev. Tervel: A unification of descriptor-based techniques for non-blocking programming. In *Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV)*, Samos, Greece, July 2015.

[3] S. Feldman, C. Valera-Leon, and D. Dechev. An efficient wait-free vector. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[4] 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.

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

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

[7] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages: C++*. American National Standards Institute, September 2011.

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

[9] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[10] C. S. E. Scalable and S. S. Lab. Tervel: A framework for implementing wait-free algorithms, 2015.

[11] 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.

ABOUT THE AUTHORS:

Steven Feldman received his MS degrees in computer science from the University of Central Florida, in 2013. His research interests include concurrency, inter-thread helping techniques, and progress conditions, which has led to the development of several wait-free algorithms.

Damian Dechev is an assistant professor in the EECS Department of the University of Central Florida and the founder of the Computer Software Engineering-Scalable and Secure Systems Lab, UCF. He specializes in the design of scalable multiprocessor algorithms and has applied them to real-time embedded space systems at NASA JPL and HPCdata-intensive applications at Sandia National Labs. His research has been supported by grants from the US National Science Foundation(NSF), Sandia National Laboratories, and the Department of Energy.