Lock-free Fill-in Queue

Basem Assiri
Jazan University
Jazan city, 45142, Saudi Arabia
babumussmar@jazanu.edu.sa

***Abstract*—One of the fundamental data structure that is commonly used in parallel and concurrent systems is first-in-first-out *queue*. Many studies propose lock-based concurrent queue algorithms to satisfy correctness property. However, the use of locks results in delays, contention, deadlock and some other issues. Instead, lock-free algorithms are introduced to overcome such issues and improve performance. Some lock-free algorithms use an atomic operation compare-and-swap (CAS) while some others replace it with fetch-and-add (FAA), to improve the performance. From implementation perspective, some queue algorithms use array data structure to ease the enqueuing and dequeuing processes but the size of queue is static. On the other hand, many queue algorithms use linked data structure where the size of queue is dynamic but the enqueuing and dequeuing processes are complicated. In this paper, we introduce new algorithms for concurrent queue that merge the ideas of array and linked data structure to get the advantages of both. Actually, our queue consists of circular linked list with k empty (dummy) nodes. Therefore, in normal case it works like array. The enqueue process places the data in one of the empty nodes (at $tail$ position), while the dequeue process deletes the data from a non-empty node (at $head$ position), and no need to maintain the linked list queue. However, our queue is dynamic such that we change the queue size by either creating new node and connect it to the linked-list, or deleting some exist nodes. Our algorithm eases the enqueue and dequeue processes, and reduces the use of CAS operations. Therefore, it improves the performance comparing to existing queue algorithms that use CAS, and almost matches the performances of the algorithms that use FAA.**

Keywords: Concurrent Queue; Fill-in Queue; Array Queue; Compare-and-swap; Linked Data Structure;

I. Introduction

The first-in-first-out (FIFO) *queue* is one of the fundamental data structure that is commonly used in parallel and concurrent systems [1]. Although, many studies have been proposed to improve the performance of algorithms, systems and applications that use concurrent queue, but they are highly challenged by correctness. Actually, the correctness here means that the queue must satisfy the queue specifications such as FIFO property, and it must satisfy the correctness properties of concurrent execution. To satisfy the correctness properties of concurrent execution, the concurrent processes, that access some shared queue objects, must execute as if they are sequential. For example, when there are two dequeue processes running concurrently, both of them read the tail position, and dequeue the element (that is in the tail of the queue), which means that both processes may dequeue the same element. This violates the queue specification (FIFO) and the concurrent execution correctness [2].

A. Literature Review

In concurrent execution, many processes access shared objects concurrently and some of them may read the value of the object while the others are changing it. The correctness of the output of each process depends on the order of all processes on the object, which can be satisfied through *linearizability*. In linearizability, each process appears as if it takes effect instantaneously at any time between its invocation and response which helps to know and predict the order of the processes [3]. By tacking the order of processes, linearizability prevents those processes that return unexpected and unreasonable output. Indeed, concurrent queue algorithms satisfy linearizability in different ways such as using locks [4], [5], [6]. However, using locks results in delays, contention, deadlock and some other problems. Instead, lock-free algorithms [1], [7], [8], [9], [10], [11], [12], [13] are introduced to overcome such issues and improve the performance.

From implementation perspective, some queue algorithms use array data structure to ease the enqueuing and dequeuing processes but in this case the size of queue is static [14], [15], [16]. On the other hand, many queue algorithms use linked data structure where the size of queue is dynamic but the enqueuing and dequeuing processes are more critical [9], [1], [10], [13]. Indeed, updating pointers, within enqueue and dequeue processes, requires more precision. In fact, the main focus of all concurrent queue algorithms that use linked data structure is to maintain pointers updating correctly; otherwise it may not be a queue any more or some part of the queue will not be accessible.

B. Proposed Solution

In this paper, we introduce new algorithms for concurrent FIFO queue that merge the ideas of array and linked data structure to get the advantages of both. Actually, our queue consists of circular linked list (which consists of nodes and pointers) of size k, with a *head* and a *tail* pointing to the positions of enqueue and dequeue. The nodes are already

978-1-7281-5467-1/20/$31.00

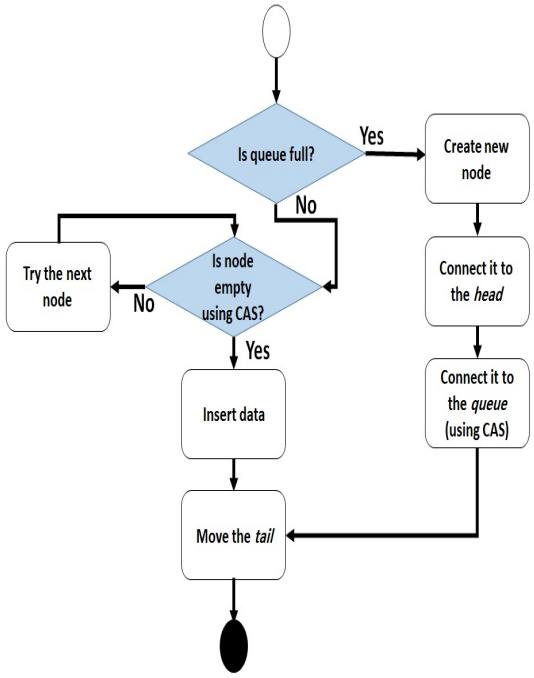

Figure 1: A Flowchart for Enqueue Process Summarizing the Proposed Solution.

exist in the queue while the initial value of every node is $Null$, and in each node there is a flag to show whether it is *empty* or not. If $empty = 1$ we can enqueue in it while the dequeue fails. Otherwise, if the node is not empty ($empty = 0$), then the dequeue works but the enqueue fails. Indeed, the processes read the head and/or the tail to find the right position, then they update the value of the node without maintaining the list. Actually the enqueue process changes the $empty$ to 0 and the value of the targeted node from $Null$ to a specific value, while the dequeue returns the exist value of the targeted node and changes it back to empty. After that, they update the head or tail position. However, when we need to change the value of k (the queue size), we create new nodes and update the pointers as needed. Figure 1 shows a flowchart for enqueue process as an example to summarize the proposed solution.

Figure 2 (a) shows the original status of a circular queue where the size of the queue is 4. The queue head points to a non-empty node (where the flag in the upper-right corner equals to 0) and the value is 1. The next node is also non-empty with a value equals to 2. The queue tail points to the first empty node in the queue where the flag is 1 and the value is $Null$. The last node is also empty and points to the head node. Moreover, the figure shows four concurrent processes which are two enqueue processes and another two dequeue processes. In Figure 2 (b), Process 1 and 2 read the tail position concurrently, while Process 3 and 4 read the head concurrently. In Figure 2 (c), since Process 1 and Process 2 read the tail position concurrently, then they try to enqueue in the tail node using CAS (which will be explained latter). Therefore, only one of them will succeed and the other should move to the next node. Let say Process 1 ($enq(3)$) finds that the tail node is empty, so updates it to non-empty and enqueues there. Process 2 ($enq(4)$) finds that the tail node is not empty (since Process 1 already gets it), so it traverses to the next node, finds it empty, then it enqueues there (notes that both processes move the tail and the queue is circular). Because of concurrency, Process 4 ($deq()$) finds that the head node is not empty, so dequeues it. Process 3 ($deq()$) finds that the head node is empty (since Process 4 already gets it), so it moves to the next node and dequeues. Thus the four concurrent processes completed successfully. Indeed it is clear that the head position in Figure 2 (b) is changed by two moves as it appears in Figure 2 (c), and the tail as well.

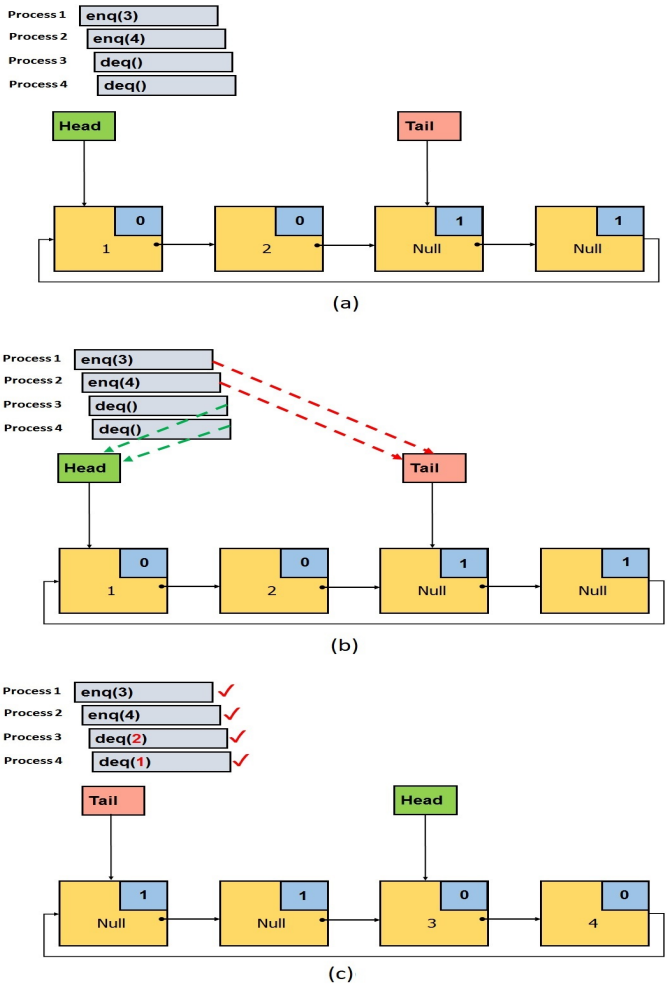

Figure 2: (a) The Original Status of the Queue and Four Concurrent Processes; (b) The Four Concurrent Processes Read Head/Tail of the Queue; (c) The Four Concurrent Processes Successfully Completed and the Queue is Updated.

Our experimental results shows that, our algorithm (Fill-in queue) outperforms the linked list queue of Michael&Scott (MS) [9], and the the array queue of Colvin et al. [16]. Also, it almost matches the performances of the algorithm of Morrison et al. [13], that uses FAA.

.

II. THE DESIGN OF THE ALGORITHMS

Our queue is a singly-linked based queue which consists of dummy nodes and pointers. Every dummy node consists of three items: (i) $data$ representing the value of the node which initializes to $Null$; (ii) $empty$ flag which shows the situation of the dummy node; and (iii) pointer $next$ pointing to the next node (Algorithm 1). The queue has two pointers which are the $head$ and the $tail$ to determine the positions of the elements in the queue (Algorithm 2). Initially, the size of queue is k, where k is an integer number, which means the queue has k dummy nodes (Algorithm 3). Our queue is circular where the last node points to the first one. Moreover, The queue has two kinds of processes which are enqueue $enq(data)$, and dequeue $deq()$. Enqueue process inserts data to a queue's node (where the $tail$ is pointing), while the dequeue process deletes the data of a queue's node (where the $head$ is pointing). Furthermore, we use CAS as an atomic operation that allows mutual exclusive access to a variable. It compares the value of the variable with an expected value. If the comparison is true, it sets a new value to the variable, so the others will find different value (false CAS). For example, CAS($x, 1, 0$), means if $x = 1$ then change it to 0. When there are many concurrent processes executing CAS, only one of them will succeed. Actually, some algorithms uses FAA to get the current value of the variable and increase it by a specific values. Thus, the processes read the head/tail positions and increase by 1, so they find the positions of enqueue and dequeue concurrently; which improves the performance [13].

Algorithm 1: Class Node

int $empty \leftarrow 1$; //Flag;
int $data \leftarrow Null$;
Node $*next \leftarrow Null$;

Algorithm 2: Class Queue Q

int $*head \leftarrow Null$;
int $*tail \leftarrow Null$;

A. Bounded Queue

Algorithms 4 and 5 show the enqueue and dequeue process on a bounded queue. For enqueue process $enq(data1)$ (Algorithm 4), process copies the value of queue $tail$ locally (in $localTail$). Then it tries the dummy node where the

Algorithm 3: Initialize

//Create k dummy nodes
for *(From* $i \leftarrow 0$ *to* $i < k$*)* **do**
  Node $node = newnode()$; //Create a new node
  $node.empty \leftarrow 1$;
  $node.data \leftarrow Null$;
  $node.next \leftarrow Null$;
  if *(i=0)* **then**
    //To create first node in the queue
    $head \leftarrow \&node$; //Set $head$ pointer
  else
    $tail \rightarrow next \leftarrow \&node$; //Let previous node's pointer to point to this node
  $tail \leftarrow \&node$; //Set $tail$ pointer
$tail \rightarrow next \leftarrow head$; //Let last node to point to the $head$ (create circular queue)
$tail \leftarrow head$; // After we create the queue, let the $head$ and $tail$ to point to the first node

$localTail$ points. Using CAS, we check if the node is empty or not. First possibility, if CAS succeeds (means the dummy node is empty), we make it non-empty by changing the value of $empty$ flag from 1 to 0, then we insert $data1$. After that, we check if $tail = localTail$, we move the $tail$, and the enqueue is done.

Second possibility, if CAS fails, which means another process already took the node, we move the $localTail$ to the next node and keep moving until either finds an empty node or reaches the $head$. (i) If it reaches the queue $head$, it returns the queue is $Full$; (ii) if we find an empty node, then we insert $data1$. Now, if $tail = localTail$, we move the $tail$, and the enqueue is done. Otherwise, there are some concurrent enqueues that should move the $tail$ first, so wait until they move the $tail$, then move the $tail$, and the enqueue is done.

For dequeue process $deq()$ (Algorithm 5), first we copy the value of queue $head$ locally (in $localHead$). First, if CAS succeeds so $localHead$ points to non-empty node), then: (i) we make it empty by changing the value of $empty$ flag from 0 to 1; (ii) we get the data and delete the node; (iii) After that, we check if $head = localHead$, we move the $head$, and the dequeue is done. Otherwise, another concurrent process is already there, so wait until the $head = localHead$.

Second, if CAS fails ,which means another process already took the node, we move the $localHead$ to the next node. If it reaches the $tail \rightarrow next$, it returns the queue is $Empty$. Otherwise we use CAS again until we find a non-empty node.

B. Unbounded Queue

Algorithm 6 shows the enqueue process on unbounded queue. The algorithm follows the same procedures of Algo-

Algorithm 4: Bounded Queue, enq($data1$)

```
localTail ← tail; //localTail is local
while (true) do
    //Try any dummy node
    if (CAS(localTail → empty, 1, 0)) then
        //If tail points to empty dummy node, then
        localTail → data ← data1;
        while (tail! = localTail) do
            //wait;
        tail ← localTail → next;
        return true; //Enqueue is done
    else
        //If CAS fails because of a concurrent successful enqueue,
          try the next dummy
        localTail ← localTail → next;
        if (localTail = head) then
            return false; //Queue is "Full"
```

Algorithm 5: Bounded Queue, deq()

```
localHead ← head; //localHead is local
while (true) do
    //Now if localHead points to nonEmpty node
    if (CAS(localHead.empty, 0, 1)) then
        data1 ← localHead → data;
        data ← Null;
        while (localHead! = head) do
            //wait;
        head ← localHead → next;
        return data1; //Dequeue is done
    else
        //If CAS fails because of a concurrent successful dequeue,
          try the next node
        localHead ← localHead → next;
        if (localHead = tail → next) then
            return false; //Queue is "Empty"
```

rithm 4, except when the queue is $Full$. In case of a full queue, we create a new node and increase the size of the queue.

Indeed, enqueue process $enq(data1)$ (Algorithm 6), copies the value of queue $tail$ locally (in $localTail$). Then it uses CAS to find an empty node. First possibility, if CAS succeeds (means the dummy node is empty), we follow the same procedure in Algorithm 4. Second possibility, if CAS fails, which means another process already took the node, we move the $localTail$ to the next node. We use CAS again until either we find an empty node or we reach the queue $head$ as follows: (i) In case of finding an empty node, we also use the same procedure in Algorithm 4; (ii) If we reach the queue $head$ (means the queue is $Full$), we create a new node containing $data1$ and pointing to the $head$. We set $localFirstTime$ to 0, so if the node enqueuing fails we keep trying to insert it using $while$ loop, without creating a new node again. Since we store the position of $localTail$ in $prevLocalTail$, we use CAS to guarantee that $prevLocalTail$'s node still pointing to the $head$ (to prevent conflicting with concurrent enqueues). We let the node of $prevLocalTail$ to point to the new node and change the queue $tali$ position, enqueue is done.

For space limits, dequeue process $deq()$ does not delete nodes from the queue, so we follow the same procedures in (Algorithm 5).

Algorithm 6: Unbounded Queue, enq($data1$)

```
localFirstTime = 1 localTail ← tail; //localTail is local
while (true) do
    //Try any dummy node
    if (CAS(localTail → empty, 1, 0)) then
        //If tail points to empty dummy node, then
        localTail → data ← data1;
        while (tail! = localTail) do
            //wait();
        tail ← localTail → next;
        return true; //Enqueue is done
    else
        //If CAS fails because of a concurrent successful enqueue,
          try the next dummy
        prevLocalTail ← localTail;
        localTail ← localTail → next;
        while (localTail = head) do
            //No more empty dummy nodes and queue is full,
              then create a new node
            if (localFirstTime = 1) then
                localFirstTime = 0;
                Node node1 ← newnode(); //Create a new node
                node1.empty ← 0;
                node1.data ← data1;
            node1.next ← head;
            if (CAS(prevLocalTail → next, head, &node1))
              then
                //Successful CAS and node is connected
                while (tail! = prevLocalTail) do
                    //wait();
                tail ← &node1;
                return true; //Enqueue is done
            else
                prevLocalTail ← prevLocalTail → next;
```

III. Correctness of the Algorithm

To avoid complications, we can apply the correctness proof for array concurrent queue in [3], [16], on algorithms 4 and 5. In fact, the bounded concurrent queue acts exactly like array queue.

Algorithm 6 are unbounded concurrent queue, which work as array concurrent queue and as linked list concurrent queue as well. Therefore, we apply the correctness proof for array concurrent queue in [16], for the normal processes. We use the correctness proof for linked list concurrent queue in [9] for the enqueues and dequeues that change the queue size.

IV. Experimental Results

In our experiment, we run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (12 cores total) clocked at 2.30 GHz. The codes are implemented using C++. We run the experiment using 24 threads, and we repeat each test for 10 times, then we show the average. We execute the same sets of enqueues and dequeues (as inputs) on four concurrent queue algorithms which are: Our algorithm (Fill-in), the array queue (Array) [16], the linked list queue (LL) [9], and the FAA queue [13].

Figure 3 shows performances comparison of Fill-in queue with Array, LL and FAA ones. The performance is measured using execution time of a same scenario of enqueue and dequeue processes on all algorithms. In Figure 3 (a), the performance of Fill-in algorithm lies between Array and LL ones. Obviously, having a reasonable number of enqueues comparing to the dequeues (considering their interleaves) highly influences the performances of the algorithms. In such scenarios, the case of full queue is rarely happens and no need to create new nodes (for Fill-in and LL). Actually, the average execution time of Fill-in queue is 45% of the LL execution time, and 112% comparing to Array execution time.

Clearly, selecting the suitable size of Fill-in queue is very critical as it determines whether the Fill-in queue mostly behaves as Array queue or as LL one.

Figure 3 (b), shows a performance comparison between Fill-in queue and FAA queue. The execution time of Fill-in queue is almost the same with the FAA algorithm.

Fiqure 4 shows the numbers of "Queue is Full" messages on an array queue of size 100. The queue is full massages appears with some enqueue processes when the data structure size is not dynamic. The the numbers of "Queue is Full" messages increase dramatically with the increment of the number of processes. On the other hand, the Fill-in, LL, and FAA queues do not return "Queue is Full" messages because of their ability of maintaining queue size.

V. Conclusion

In conclusion, our lock-free Fill-in queue algorithm guarantees the fruit of linked list queue and array queue. It uses CAS and still matching the performance of FAA ones. With slight modifications, Fill-in queue is applicable to satisfy wait-freedom property (that cannot be presented because of the space limit).

References

[1] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *International Symposium on Distributed Computing*, pages 117–131. Springer, 2004.

[2] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

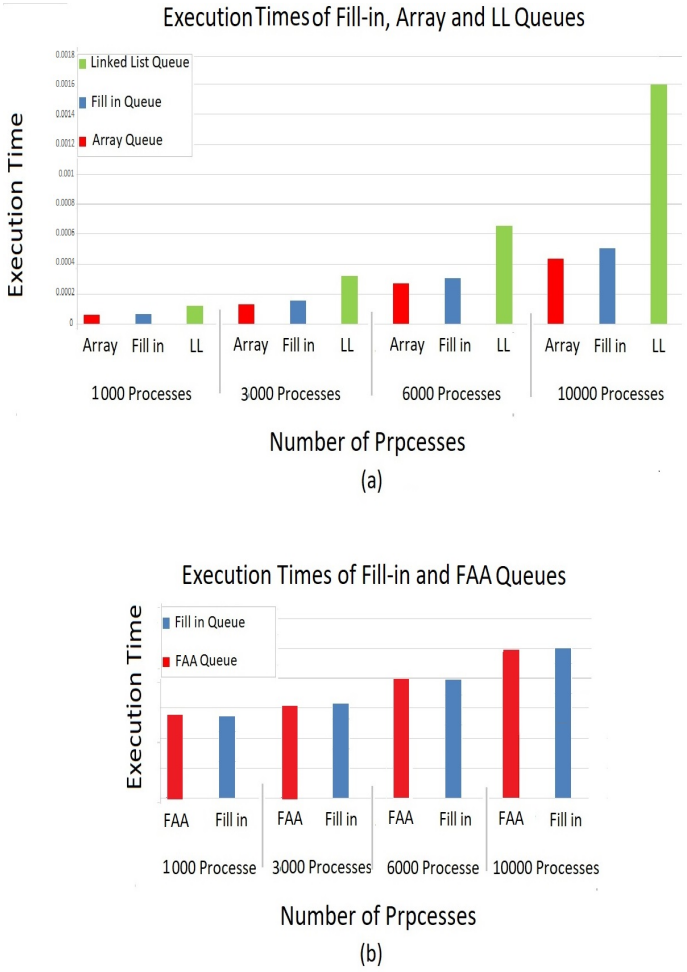

Figure 3: (a) Compare the Performance of Fill-in Queue with LL Queue and Array Queue; (b) Compare the Performance of Fill- in Queue with FAA Queue.

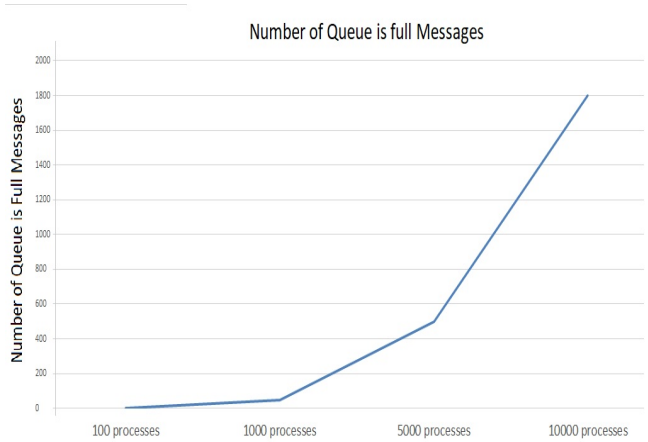

Figure 4: The Numbers of "Queue is Full" Messages on the Array Queue of Size 100, with Different Number of Processes.

[3] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[4] Travis Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993.

[5] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[6] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[7] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[8] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992.

[9] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[10] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262. ACM, 2005.

[11] John D Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[12] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center New York, 1986.

[13] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112. ACM, 2013.

[14] Matei David. A single-enqueuer wait-free queue implementation. In *International Symposium on Distributed Computing*, pages 132–143. Springer, 2004.

[15] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[16] Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 507–516. IEEE, 2005.