A Fast Wait-Free Multi-Producers Single-Consumer Queue

Dolev Adas
CS Technion
Israel

Roy Friedman
CS Technion
Israel

ABSTRACT

In sharded data processing systems, sharded in-memory key-value stores, data flow programming and load sharing, multiple concurrent data producers feed requests into the same data consumer. This can be naturally realized through concurrent queues, where each consumer pulls its tasks from its dedicated queue. For scalability, wait-free queues are preferred over lock based structures.

The vast majority of wait-free queue implementations, and even lock-free ones, support the multi-producer multi-consumer model. Yet, this comes at a premium, since implementing wait-free multi-producer multi-consumer queues requires utilizing complex helper data structures. The latter increases the memory consumption of such queues and limits their performance and scalability. Many such designs employ (hardware) cache unfriendly access patterns.

In this work we study the implementation of wait-free multi-producer single-consumer queues. Specifically, we propose Jiffy, an efficient memory frugal novel wait-free multi-producer single-consumer queue and formally prove its correctness. We compare the performance and memory requirements of Jiffy with other state of the art lock-free and wait-free queues. We show that indeed Jiffy can maintain good performance with up to 128 threads, delivers up to 50% better throughput than the next best construction we compared against, and consumes ≈90% less memory.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Wait-free, Unboubnded Queue, Multi Producer Single Consumer

ACM Reference Format:
Dolev Adas and Roy Friedman. 2022. A Fast Wait-Free Multi-Producers Single-Consumer Queue. In *23rd International Conference on Distributed Computing and Networking (ICDCN 2022), January 4–7, 2022, Delhi, AA, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3491003.3491004

1 INTRODUCTION

Concurrent queues are a fundamental data-exchange mechanism in multi-threaded applications. A queue enables one thread to pass a data item to another thread in a decoupled manner, while preserving ordering between operations. The thread inserting a data item is often referred to as the *producer* or *enqueuer* of the data, while the thread that fetches and removes the data item from the queue is often referred to as the *consumer* or *dequeuer* of the data. In particular, queues can be used to pass data from multiple threads to a single thread - known as *multi-producer single-consumer queue* (MPSC), from a single thread to multiple threads - known as *single-producer multi-consumer queue* (SPMC), or from multiple threads to multiple threads - known as *multi-producer multi-consumer queue* (MPMC). SPMC and MPSC queues are demonstrated in Figure 1.

MPSC is useful in sharded software architectures, and in particular for sharded in-memory key-value stores and sharded in-memory databases, resource allocation and data-flow computation schemes. Another example is the popular Caffeine Java caching library [20], in which a single thread is responsible for updating the internal cache data structures and meta-data. As depicted in Figure 1b, in such architectures, a single thread is responsible for each shard, in order to avoid costly synchronization while manipulating the state of a specific shard. In this case, multiple feeder threads (e.g., that communicate with external clients) insert requests into the queues according to the shards. Each thread that is responsible for a given shard then repeatedly dequeues the next request for the shard, executes it, dequeues the next request, etc. Similarly, in a data flow graph, multiple events may feed the same computational entity (e.g., a reducer that reduces the outcome of multiple mappers). Here, again, each computational entity might be served by a single thread while multiple threads are sending it items, or requests.

MPMC is the most general form of a queue and can be used in any scenario. Therefore, MPMC is also the most widely studied data structure [17, 18, 21, 25–27, 29]. Yet, this may come at a premium compared to using a more specific queue implementation.

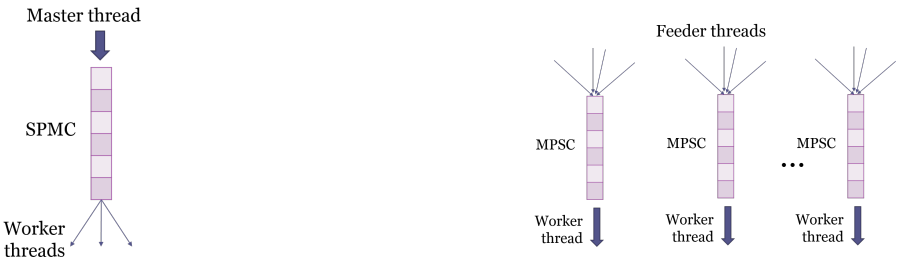
Specifically, concurrent accesses to the same data structure require adequate concurrency control to ensure correctness. The simplest option is to lock the entire data structure on each access, but this usually dramatically reduces performance due to the sequentiality and contention it imposes [12]. A more promising approach is to reduce, or even avoid, the use of locks and replace them with *lock-free* and *wait-free* protocols that only rely on atomic operations such as *fetch-and-add* (FAA) and *compare-and-swap* (CAS), which are supported in most modern hardware architectures [11]. Wait-free implementations are particularly appealing as they ensure that each operation always terminates in a finite number of steps.

Alas, known MPMC wait-free queues suffer from large memory overheads, intricate code complexity, and low scalability. In particular, it was shown that wait-free MPMC queues require the use of a helper mechanism [5]. On the other hand, as discussed above, there are important classes of applications for which MPSC queues are adequate. Such applications could therefore benefit if a more efficient MPSC queue construction was found. This motivates studying wait-free MPSC queues, which is the topic of this paper.

Contributions. We present Jiffy, a fast memory efficient wait-free MPSC queue. Jiffy is unbounded in the the number of elements that

ICDCN 2022, January 4–7, 2022, Delhi, AA, India

ACM ISBN 978-1-4503-9560-1/22/01…$15.00
https://doi.org/10.1145/3491003.3491004



(a) SPMC queue used in a master worker architecture. A single master queues up tasks to be executed, which are picked by worker threads on a first comes first served basis.

(b) MPSC queues used in a sharded architecture. Here, each shard is served by a single worker thread to avoid synchronization inside the shard. Multiple collector threads can feed the queue of each shard.

Figure 1: SPMC vs. MPSC queues.

can be enqueued without being dequeued (up to the memory limitations of the machine). Yet the amount of memory Jiffy consumes at any given time is proportional to the number of such items and Jiffy minimizes the use of pointers, to reduce its memory footprint.

To obtain these properties, Jiffy stores elements in a linked list of arrays, and only allocates a new array when the last array is being filled. As soon as all elements in a given array are dequeued, the array is released. This way, a typical enqueue operation requires little more than a simple FAA and setting the corresponding entry to the enqueued value and changing its status from `empty` to `set`. Hence, operations are very fast and the number of pointers is a multiple of the allocated arrays rather than the number of queued elements.

For linearizability and wait-freedom, a dequeue operation in Jiffy may return a value that is already past the head of the queue, if the enqueue operation working on the head is on-going. To ensure correctness, we devised a novel mechanism to handle such entries both during their immediate dequeue and during subsequent dequeues.

Another novel idea in Jiffy is related to its buffer allocation policy. In principle, when the last buffer is full, the naive approach is for each enqueuer at that point to allocate a new buffer and then try adding it to the queue with a CAS. When multiple enqueuers try this concurrently, only one succeeds and the others need to free their allocated buffer. However, this both creates contention on the end of the queue and wastes CPU time in allocating and freeing multiple buffers each time. To alleviate these phenomena, in Jiffy the enqueuer of the second entry in the last buffer already allocates the next buffer and tries to add it using CAS. This way, almost always, when enqueuers reach the end of a buffer, the next buffer is already available for them without any contention.

We have implemented Jiffy and evaluated its performance in comparison with three other leading lock-free and wait-free implementations, namely WFqueue [33], CCqueue [8], and MSqueue [21]. We also examined the memory requirements for the data and code of all measured implementations using valgrind [24]. The results indicate that Jiffy is up to 50% faster than WFqueue and roughly 10 times faster than CCqueue and MSqueue. Jiffy is also more scalable than the other queue structures we tested, enabling more than 20 million operations per second even with 128 threads. Finally, the memory footprint of Jiffy is roughly 90% better than its competitors in the tested workloads.

Jiffy obtains better performance since the size of each queue node is much smaller and there are no auxiliary data structures. For example, in WFqueue, which also employs a linked list of arrays approach, each node maintains two pointers, there is some per-thread meta-data, the basic slow-path structure (even when empty), etc. Further, WFqueue employs a lazy reclamation policy, which according to its authors is significant for its performance. Hence, arrays are kept around for some time even after they are no longer useful. In contrast, the per-node meta-data in Jiffy is just a 2-bit flag, and arrays freed as soon as they become empty. This translates to a more hardware cache friendly access pattern (see Tables 1 and 2). As further evidence, the full version [2] includes measurements where we artificially increase the memory size taken by Jiffy and experience a drop in performance. Also, dequeue operations in Jiffy do not invoke any atomic (e.g., FAA & CAS) operations at all.

2 RELATED WORK

Implementing concurrent queues is a widely studied topic [3, 4, 7, 10, 19, 25, 30, 32]. Below we focus on the most relevant works.

Multi-Multi Queues: The most well known lock-free queue constructions is Michael and Scott's [21], aka *MSqueue*. It is based on a singly-linked list of nodes that hold the enqueued values plus two references to the head and the tail. Their algorithm does not scale past a few threads due to contention on the queue's head and tail.

Kogan and Petrank have introduced a wait-free variant of the MSqueue [15]. Their queue extends the helping technique already employed by Michael and Scott to achieve wait freedom with similar performance characteristics. They achieve wait-freedom by assigning each operation a dynamic age-based priority and making threads with younger operations help older operations to complete through the use of another data structure named a *state* array.

Morrison and Afek proposed LCRQ [23], a non-blocking queue based on a linked-list of circular ring segments called CRQ. LCRQ uses FAA to grab an index in the CRQ. Enqueue and dequeue operations in [23] involve a double-width compare-and-swap (CAS2).

Yang and Mellor-Crummey proposed WFqueue, a wait free queue based on FAA [33]. WFqueue utilizes a linked-list of fixed size segments. Their design employs the fast-path-slow-path methodology [16] to transform a FAA based queue into a wait-free queue.

That is, an operation on the queue first tries the fast path implementation until it succeeds or the number of failures exceeds a threshold. If necessary, it falls back to the slow-path, which guarantees completion within a finite number of attempts. Each thread needs to register when the queue is started, so the number of threads cannot change after the initialization of the queue. In contrast, in our queue a thread can join anytime during the run.

Fatourou and Kallimanis proposed CCqueue [8], a blocking queue that uses combining. In CCqueue, a single thread scans a list of pending operations and applies them to the queue. Threads add their operations to the list using SWAP.

Tsigas and Zhang proposed a non-blocking queue that allows the head and tail to lag at most m nodes behind the actual head and tail of the queue [31]. Additional cyclic array queues are described in [9, 28]. Recently, a lock-free queue that extends MSqueue [21] to support batching operations was presented in [22].

Limited Concurrency Queues: David proposed a sublinear time wait-free queue [6] that supports multiple dequeuers and one enqueuer. His queue is based on infinitely large arrays. As stated in [6], the space requirement can be bounded at the cost of increasing the time complexity to *O(n)*, where n is the number of dequeuers.

Jayanti and Petrovic proposed a wait-free queue implementation supporting multiple enqueuers and one concurrent dequeuer [14]. Their queue is based on a binary tree whose leaves are linear linked lists. Each linked list represents a "local" queue for each thread that uses the queue. Their algorithm keeps one local queue at each process and maintains a timestamp for each element to decide the order between the elements in the different queues.

3 PRELIMINARIES

We consider a standard shared memory setting and a set of threads accessing shared memory only via the following atomic operations:

- Store - Atomically replaces the value stored in the target address with the given value.
- Load - Atomically loads and returns the current value of the target address.
- CAS - Atomically compares the value stored in the target address with the expected value. If those are equal, replaces the former with the desired value and the Boolean value `true` is returned. Otherwise, the shared memory is unchanged and `false` is returned.
- FAA -Atomically adds a given value to the value stored in the target address and returns the value in the target address held previously.

We assume that a program is composed of multiple such threads, which specifies the order in which each thread issues these operations and the objects on which they are invoked. In an execution of the program, each operation is invoked, known as its *invocation* event, takes some time to execute, until it terminates, known as its *termination* event. Each termination event is associated with one or more values *being returned by that operation*. An execution is called *sequential* if each operation invocation is followed immediately by the same operation's termination (with no intermediate events between them). Otherwise, the execution is said to be *concurrent*. When one of the operations in an execution σ is being invoked on object x, we say that x is being *accessed* in σ.

Given an execution σ, we say that σ induces a (partial) *real-time ordering* among its operations: Given two operations o_1 and o_2 in σ, we say that o_1 *appears before* o_2 in σ if the invocation of o_2 occurred after the termination of o_1 in σ. If neither o_1 nor o_2 appears before the other in σ, then they are considered *concurrent operations* in σ. In a sequential execution, the real-time ordering is a total order.

Also, we assume that each object has a *sequential specification*, which defines the set of allowed sequential executions on this object. A sequential execution σ is called *legal* w.r.t. a given object x if the restriction of this execution to operations on x (only) is included in the sequential specification of x. σ is said to be legal if it is legal w.r.t. any object being accessed in σ. Further, we say that two executions σ and σ‘ are *equivalent* if there is a one-to-one mapping between each operation in σ to an operation in σ' and the values each such pair of operations return in σ and σ' are the same.

Linearizability An execution σ is *linearizable* [13] if it is equivalent to a legal sequential execution σ' and the order of all operations in σ' respects the real-time order of operations in σ.

4 MPSC QUEUE

It is known that any wait-free implementation of multi producer multi consumer queues requires utilizing helper data structures [5], which are both memory wasteful and slow down the rate of operations. By settling for single consumer support, we can avoid helper data structures. Further, for additional space efficiency, we seek solutions that minimize the use of pointers, as each pointer consumes 64 bits on most modern architectures, and their usage tends to induce cache unfriendly access patterns. Supporting unbounded queues is needed since there can be periods in which the rate of enqueue operations surpasses the rate of dequeues. Without this property, during such a period some items might be dropped, or enqueuers might need to block until enough existing items are dequeued. In the latter case the queue is not wait-free.

4.1 Overview

Our queue structure is composed of a linked list of buffers. A buffer enables implementing a queue without pointers, as well as using fast *FAA* instructions to manipulate the head and tail indices of the queue. However, a single buffer limits the size of the queue, and is therefore useful only for bounded queue implementations. In order to support unbounded queues, we extend the single buffer idea to a linked list of buffers. This design is a tradeoff point between avoiding pointers as much as possible while supporting an unbounded queue. It also decreases the use of *CAS* to only adding a new buffer to the queue, which is performed rarely.

Once a buffer of items has been completely read, the (sole) consumer deletes the buffer and removes it from the linked list. Hence, deleting obsolete buffers requires no synchronization. When inserting an element to the queue, if the last buffer in the linked list has room, we perform an atomic *FAA* to a *tail* index and insert the element. Otherwise, the producer allocates a new buffer and tries to insert it to the linked list via an atomic *CAS*, which keeps the overall size of the queue small while supporting unbounded sizes[1]. The structure of a Jiffy queue is depicted in Figure 2.

[1]In fact, to avoid contention the new buffer is usually already allocated earlier on; more accurate details appear below.

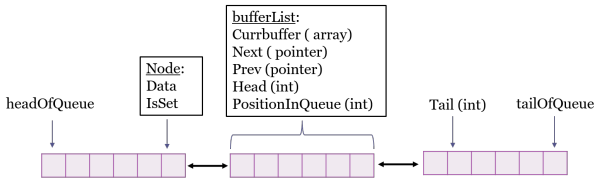


Figure 2: A node in Jiffy consists of two fields: the data itself and an *IsSet* field that notifies when the data is ready to be read. The *BufferList* consists of 5 fields: *Currbuffer* is an array of nodes, a *Next* and *Prev* pointers, *Head* index(integer) pointing to the last place in that buffer that the consumer read, and *PositionInQueue* that tracks the location of the *BufferList* in the list. The *Tail* index indicates the last place to be written to by the producers. *HeadOfQueue* is the consumer pointer for the first buffer; once the head reaches the end of the buffer, the corresponding *BufferList* is deleted. *TailOfQueue* points to the last BufferList in the linked list; once the *Tail* reaches the end of this buffer a new array BufferList is added.

Yet, the queue as described so far is not linearizable. Figure 3 exhibits why the naive proposal for the queue violates linearizability. This scenario depicts two concurrent enqueue operations $enqueue_1$ and $enqueue_2$ for items i_1 and i_2 respectively, where $enqueue_1$ terminates before $enqueue_2$. Also, assume a dequeue operation that overlaps only with $enqueue_2$ (it starts after $enqueue_1$ terminates). It is possible that item i_2 is inserted at an earlier index in the queue than the index of i_1 because this is the order by which their respective *FAA* instructions got executed. Yet, the insertion of i_1 terminates quickly while the insertion of i_2 (into the earlier index) takes longer. Now the dequeue operation sees that the *head* of the queue is still empty, so it returns immediately with an empty reply. But since $enqueue_1$ terminates before the dequeue starts, this violates linearizability.

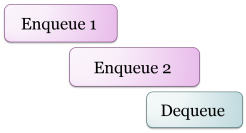
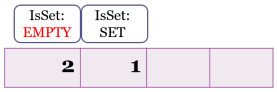
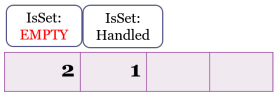


Figure 3: A linearizability problem in the basic queue idea.

To make the queue linearizable, if the dequeuer finds that the *isSet* flag is not `set` (nor `handled`, to be introduced shortly), as illustrated in Figure 4a, the dequeuer continues to scan the queue until either reaching the *tail* or finding an item x at position i whose *isSet* is `set` but the *isSet* of its previous entry is not, as depicted in Figure 4b. In the latter case, the dequeuer verifies that the *isSet* of all items from the head until position i are still neither `set` nor `handled` and then returns x and sets the *isSet* at position i to `handled` (to avoid re-dequeuing it in the future). This double reading of the *isSet* flag is done to preserve linearizability in case an earlier enqueuer has terminated by the time the newer item is found. In the latter case the dequeue should return the earlier item to ensure correct ordering. This becomes clear in the correctness proof.



(a) The dequeuer finds that the *isSet* flag for the element pointed by the head is Empty.



(b) The dequeuer found an item whose *isSet* is ready, the *isSet* of the entry pointed by head is not set. It removes this item and marks its *isSet* as `handled`.

Figure 4: An example of the solution for the basic queue's linearizability problem.

Algorithm 1 Internal Jiffy classes and fields

```
1: class Node {
2:   T data;
3:   atomic < State > isSet; }
4: // isSet has three values: empty, set and handled.
5: class bufferList {
6:   Node * currBuffer;
7:   atomic < bufferList* > next;
8:   bufferList * prev;
9:   unsigned int head;
10:  unsigned int positionInQueue; }
11: bufferList * headOfQueue;
12: atomic < bufferList* > tailOfQueue;
13: atomic < uint_fast64_t > tail;
```

4.2 Jiffy Queue Algorithm

4.2.1 Internal Jiffy classes and fields. Algorithm 1 depicts the internal classes and variables. The *Node* class represents an item in the queue. It holds the data itself and a flag to indicate whether the data is ready to be read (*isSet*=`set`), the node is empty or is in inserting process (*isSet*=`empty`), or it was already read by the dequeuer (*isSet*=`handled`). Every node in the queue starts with *isSet*=`empty`.

The *bufferList* class represents each buffer in the queue: *currbuffer* is the buffer itself – an array of nodes; *next* is a pointer to the next buffer – it is atomic as several threads (the enqueuers) try to change it simultaneously; *prev* is a pointer to the previous buffer – it is not concurrently modified and therefore not atomic; *head* is an index to the first location in the buffer to be read by the dequeuer – it is changed only by the single dequeuer thread; *positionInQueue* is the logical position of the buffer in the queue – since consumed buffers can be freed, we need a way to calculate the indices of items placed within an allocated buffer; this is done by multiplying *positionInQueue* by the number of elements in a buffer.

The rest are fields of the queue: *headOfQueue* points to the head buffer of the queue. The dequeuer reads from the buffer pointed by *headOfQueue* at index *head* in that buffer. It is only changed by the single threaded dequeuer. *tailOfQueue* points to the last queue's buffer. It is atomic as it is modified by several threads. *tail*, initialized to 0, is the index of the last queued item. All threads perform FAA on tail to receive a location of a node to insert into.

4.2.2 The Enqueue Operation. A high-level pseudo-code for enqueue operations appears in Algorithm 2; a more detailed listing appears in the full version [2]. The enqueue operation can be called

Algorithm 2 Enqueue operation

```
1:  function ENQUEUE(data)
2:      location = FAA(tail);
3:      while the location is in an unallocated buffer do
4:          allocate a new buffer and try adding it to the queue with a CAS on tailOfQueue
5:          if unsuccessful then
6:              delete the allocated buffer
7:      bufferList* tempTail = tailOfQueue
8:      while the location is not in the buffer pointed by tempTail do
9:          tempTail = tempTail→ prev
10:     //location is in this buffer
11:     adjust location to its corresponding index in tempTail
12:     tempTail[location].data = data
13:     tempTail[location].isSet = set
14:     if location is the second entry of the last buffer then
15:         allocate a new buffer and try adding it with a CAS on tailOfQueue; if unsuccessful,
        delete this buffer
```

by multiple threads acting as producers. The method starts by performing FAA on *Tail* to acquire an index in the queue (Line 2). When there is a burst of enqueues the index that is fetched can be in a buffer that has not yet been added to the queue, or a buffer that is prior to the last one. Hence, the enqueuer needs to find in which buffer it should add its new item.

If the index is beyond the last allocated buffer, the enqueuer allocates a new buffer and tries to add it to the queue using a *CAS* operation (line 3). If several threads try simultaneously, one *CAS* will succeed and the other enqueuers' *CAS* will fail. Every such enqueuer whose *CAS* fails must delete its buffer (line 6). If the *CAS* succeeds, the enqueuer sets *tailOfQueue* to the new buffer. If there is already a next buffer allocated then the enqueuer only updates the *tailOfQueue* pointer through a CAS operation.

If the index is earlier in the queue compared to the tail index (line 8), the enqueuer retracts to the previous buffer. When the thread reaches the correct buffer, it stores the data in the buffer index it fetched at the beginning (line 12), marks the location as `Set` (line 13) and finishes. Yet, just before returning, if the thread is in the last buffer of the queue and it obtained the second index in that buffer, the enqueuer tries to add a new buffer to the end of the queue (line 14). This is an optimization step to prevent wasteful contention prone allocations of several buffers and the deletion of most as will be explained next.

Notice that if we only had the above mechanism, then each time a buffer ends there could be a lot of contention on adding the new buffer, and many threads might allocate and then delete their unneeded buffer. This is why the enqueuer that obtains the second entry in each buffer already allocates the next one. With this optimization, usually only a single enqueuer tries to allocate such a new buffer and by the time enqueuers reach the end of the current buffer a new one is already available to them. On the other hand, we still need the ability to add a new buffer if one is not found (line 3) to preserve wait-freedom.

4.2.3 The Dequeue Operation. A high-level pseudo-code for dequeue operations appears in Algorithm 3; a more detailed listing appears in the full version [2]. A dequeue operation is called by a single thread, the consumer. A dequeue starts by advancing the head index to the first element not marked with *isSet* = `handled` as such items are already dequeued (line 4). If the consumer reads an entire buffer during this stage, it deletes it (line 7). At the end of this scan, the dequeuer checks if the queue is empty and if so returns false (line 8).

Algorithm 3 Dequeue operation

```
1:  function DEQUEUE
2:      mark the element pointed by head as n
3:      //Skip to the first non-handled element (due to the code below, it might not be pointed by
        head!)
4:      while n.isSet == handled do
5:          advance n and head to the next element
6:          if the entire buffer has been read then
7:              move to the next buffer if exists and delete the previous buffer
8:      if queue is empty then
9:          return false
10:     // If the queue is not empty, but its first element is, there might be a Set element further on
        – find it
11:     if n.isSet == empty then
12:         for (tempN = n; tempN.isSet != set and not end of queue; advance tempN to next ele-
            ment) do
13:             if the entire buffer is marked with handled then
14:                 "fold" the queue by deleting this buffer and move to the next buffer if exists
15:                 ▷ Notice comment about a delicate garbage collection issue in the description
                    text
16:         if reached end of queue then
17:             return false
18:     // Due to concurrency, some element between n and tempN might have been set – find it
19:     for (e = n; e = tempN or e is before tempN; advance e to next element) do
20:         if e is not n and e.isSet == set then
21:             tempN =e and restart the for loop again from n
22:     // we scanned the path from head to tempN and did not find a prior set element - remove
        tempN
23:     tempN.isSet= handled
24:     if the entire buffer has been read then
25:         move to the next buffer if exists and delete the previous buffer
26:     else if tempN = n then
27:         advance head
28:     return tempN.data
```

Next, if the first item is in the middle of an enqueue process (*isSet*=`empty`), the consumer scans the queue (line 12) to avoid the linearizability pitfall mentioned above. If there is an element in the queue that is already `set` while the element pointed by the head is still `empty`, then the consumer needs to dequeue the latter item (denoted *tempN* in line 12).

During the scan, if the consumer reads an entire buffer whose cells are all marked `handled`, the consumer deletes this buffer (line 14). We can think of this operation as "folding" the queue by removing buffers in the middle of the queue that have already been read by the consumer. Hence, if a thread fetches an index in the queue and stalls before completing the enqueue operation, we only keep the buffer that contains this specific index and may delete all the rest, as illustrated in Figure 5.

Further, before dequeuing, the consumer scans the path from head to *tempN* to look for any item that might have changed its status to `set` (line 21). If such an item is found, it becomes the new *tempN* and the scan is restarted.

Finally, a dequeued item is marked `handled` (line 23), as also depicted in Figure 4b. Also, if the dequeued item was the last non-`handled` in its buffer, the consumer deletes the buffer and sets the head to the next one (line 26).

There is another delicate technical detail related to deleting buffers in non-garbage collecting environments such as the runtime of C++. For clarity of presentation and since it is only relevant in non-garbage collecting environments, the following issue is not addressed in Figure 3, but is rather deferred to the full version of this work [2]. Specifically, when preforming the fold, only the array is deleted, which consumes the most memory, while the much smaller meta-data structure of the array is transferred to a dedicated garbage collection list. The reason for not immediately deleting the the entire array's structure is to let enqueuers, who kept a pointer

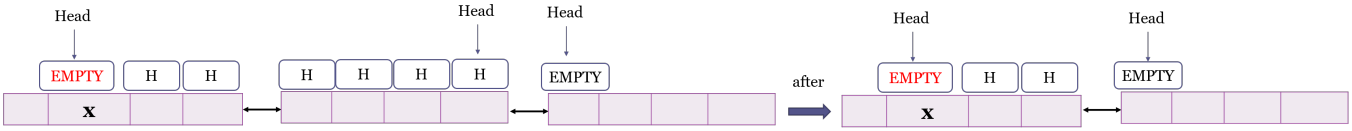


Figure 5: "folding" the queue - A thread fetches an index in the queue and stalls before completing the enqueue operation. Here we only keep the buffer that contains this specific index and delete the rest. H stands for *isSet* = `handled` and EMPTY stands for *isSet* = `empty`.

for tailOfQueue point to a valid bufferList at all times, with the correct prev and next pointers, until they are done with it. The exact details of this garbage collection mechanism are in [2].

4.2.4 Memory Buffer Pool Optimization. Instead of always allocating and releasing buffers from the operating system, we can maintain a buffer pool. This way, when trying to allocate a buffer, we first check if there is already a buffer available in the buffer pool. If so, we simply claim it without invoking an OS system call. Similarly, when releasing a buffer, rather than freeing it with an OS system call, we can insert it into the buffer pool. It is possible to have a single shared buffer pool for all threads, or let each thread maintain its own buffer pool. This optimization can potentially reduce execution time at the expense of a somewhat larger memory heap area. The code used for Jiffy's performance measurements does *not* employ this optimization. In Jiffy, allocation and freeing of buffers are in any case a relatively rare event.

5 CORRECTNESS

5.1 Linearizability

To prove linearizability, we need to show that for each execution σ that may be generated by Jiffy, we can find an equivalent legal sequential execution that obeys the real-time order of operations in σ. We show this by constructing such an execution. Further, for simplicity of presentation, we assume here that each value can be enqueued only once. Consequently, it is easy to verify from the code that each value can also be returned by a dequeue operation at most once. In summary we have:

Observation 5.1. Each enqueued value can be returned by a dequeue operation at most once.

Theorem 5.2. The Jiffy queue implementation is linearizable.

Proof. Let σ be an arbitrary execution generated by Jiffy. We now build an equivalent legal sequential execution σ'. We start with an empty sequence of operations σ' and gradually add to it all operations in σ until it is legal and equivalent to σ. Each operation inserted into σ' is collapsed such that its invocation and termination appear next to each other with no events of other operations in between them, thereby constructing σ' to be sequential.

First, all dequeue operations of σ are placed in σ' in the order they appear in σ. Since Jiffy only supports a single dequeuer, then in σ there is already a total order on all dequeue operations, which is preserved in σ'.

Next, by the code, a dequeue operation *deq* that does not return `empty`, can only return a value that was inserted by an enqueue operation *enq* that is concurrent or prior to *deq* (only enqueue operations can change an entry to `set`). By Observation 5.1, for each such *deq* operation there is exactly one such *enq* operation. Hence, any ordering in which *enq* appears before *deq* would preserve the real-time ordering between these two operations.

Denote the set of all enqueue operations in σ by ENQ and let $\widehat{ENQ}$ be the subset of ENQ consisting of all enqueue operations *enq* such that the value enqueued by *enq* is returned by some operation *deq* in σ. Next, we order all enqueue operations in $\widehat{ENQ}$ in the order of the dequeue operations that returned their respective value. □

Claim 5.3. The real time order in σ is preserved among all operations in $\widehat{ENQ}$.

Proof of Claim 5.3. Suppose Claim 5.3 does not hold. Then there must be two enqueue operations enq_1 and enq_2 and corresponding dequeue operations deq_1 and deq_2 such that the termination of enq_1 is before the invocation of enq_2 in σ, but deq_2 occurred before deq_1. Denote the entry accessed by enq_1 by in_1 and the entry accessed by enq_2 by in_2. Since the invocation of enq_2 is after the termination of enq_1, then during the invocation of enq_2 the status of in_1 was already `set` (from line 13 in Algorithm 2). Moreover, in_2 is further in the queue than in_1 (from line 2 in Algorithm 2). Since deq_2 returned the value enqueued by enq_2, by the time deq_2 accessed in_2 the status of in_1 was already `set`. As we assumed deq_1 is after deq_2, no other dequeue operation has dequeueud the value in in_1. Hence, while accessing in_2 and before terminating, deq_2 would have checked in_1 and would have found that it is now `set` (the loops in lines 4 and 19 of Algorithm 3 – this is the reason for line 19) and would have returned that value instead of the value at in_2. A contradiction. □

To place the enqueue operations of $\widehat{ENQ}$ inside σ', we scan them in their order in $\widehat{ENQ}$ (as defined above) from earliest to latest. For each such operation *enq* that has not been placed yet in σ: (*i*) let deq' be the latest dequeue operation in σ that is concurrent or prior to *enq* in σ and neither deq' nor any prior dequeue operation return the value enqueued by *enq*, and (*ii*) let deq'' be the following dequeue operation in σ; we add *enq* in the last place just before deq'' in σ'. This repeats until we are done placing all operations from $\widehat{ENQ}$ into σ'.

Claim 5.4. The real-time ordering in σ between dequeue operations and enqueue operations in $\widehat{ENQ}$ is preserved in σ' as built thus far.

Proof of Claim 5.4. Since we placed each enqueue operation enq before any dequeue operation whose invocation is after the termination of enq, we preserve real-time order between any pair of enqueue and dequeue operations. Similarly, by construction the relative order of dequeue operations is not modified by the insertion of enqueue operations into σ'. Thus, real-time ordering is preserved. □

Thus, any potential violation of real time ordering can only occur by placing enqueue operations in a different order (w.r.t. themselves) than they originally appeared in $\widehat{ENQ}$. For this to happen, it means that there are two enqueue operation enq_1 and enq_2 such that enq_1 appears before enq_2 in $\widehat{ENQ}$ but ended up in the reverse order in σ'. Denote deq'_1 the latest dequeue operation in σ that is prior or concurrent to enq_1 and neither deq'_1 nor any prior dequeue operation return the value enqueued by enq_1 and similarly denote deq'_2 for enq_2. Hence, if enq_2 was inserted at an earlier location than enq_1, then deq'_2 is also before deq'_1. But since the order of enq_1 and enq_2 in $\widehat{ENQ}$ preserves their real time order, the above can only happen if the value enqueued by enq_2 was returned by an earlier dequeue than the one returning the value enqueued by enq_1. Yet, this violates the definition of the ordering used to create $\widehat{ENQ}$.

Claim 5.5. The constructed execution σ' preserves legality.

Proof of Claim 5.5. By construction, enqueue operations are inserted in the order their values have been dequeued, and each enqueue is inserted before the dequeue that returned its value. Hence, the only thing left to show is legality w.r.t. dequeue operations that returned `empty`, which is addressed below.

Given a dequeue operation deq_i that returns `empty`, denote $\#\widehat{deq}_{\sigma',i}$ the number of dequeue operations that did not return `empty` since the last previous dequeue operation that did return `empty`, or the beginning of σ' of none exists. Similarly, denote $\#enq_{\sigma',i}$ the number of enqueue operations during the same interval of σ'. □

Sub-Claim 5.6. For each deq_i that returns empty, $\#\widehat{deq}_{\sigma',i} > \#enq_{\sigma',i}$.

Proof of Sub-Claim 5.6. Recall that the ordering in σ' preserves the real time order w.r.t. σ and there is a single dequeuer. Assume by way of contradiction that the claim does not hold, and let deq_i be the first dequeue operation that returned `empty` while $\#\widehat{deq}_{\sigma',i} \leq \#enq_{\sigma',i}$. Hence, there is at least one enqueue operation enq_j in the corresponding interval of σ' whose value is not dequeued in this interval. In this case, deq_i cannot be concurrent to enq_j in σ since otherwise by construction enq_j would have been placed after it (as it does not return its value). Hence, deq_i is after enq_j in σ. Yet, since each dequeue removes at most one item from the queue, when deq_i starts, the tail of the queue is behind the head and there is at least one item whose state is `set` between them. Thus, by lines 4 and 19 of Algorithm 3, deq_i would have returned one of these items rather then return `empty`. A contradiction. □

With Sub-Claim 5.6 we conclude the proof that σ' as constructed so far is legal.

The last thing we need to do is to insert enqueue operations whose value was not dequeued, i.e., all operations in $ENQ \setminus \widehat{ENQ}$. Denote by enq' the last operation in $\widehat{ENQ}$.

Claim 5.7. Any operation $enq'' \in ENQ \setminus \widehat{ENQ}$ is either concurrent to or after enq' in σ.

Proof of Claim 5.7. Assume, by way of contradiction, that there is an operation $enq'' \in ENQ \setminus \widehat{ENQ}$ that is before enq' in σ. Hence, by the time enq' starts, the corresponding entry of enq'' is already `set` (line 13 of Algorithm 2) and enq' obtains a later entry (line 2). Yet, since dequeue operations scan the queue from head to tail until finding a `set` entry (lines 4, 12, and 19 in Algorithm 3), the value enqueued by enq'' would have been dequeued before the value of enq'. A contradiction. □

Hence, following Claim 5.7, we insert to σ' all operations in $ENQ \setminus \widehat{ENQ}$ after all operations of $\widehat{ENQ}$. In case of an enqueue operation enq that is either concurrent with or later than a dequeue deq in σ, then enq is inserted to σ' after deq. Among concurrent enqueue operations, we break symmetry arbitrarily. Hence, real-time order is preserved in σ'.

The only thing to worry about is the legality of dequeue operations that returned `empty`. For this, we can apply the same arguments as in Claim 5.5 and Sub-claim 5.6. In summary, σ' is an equivalent legal sequential execution to σ that preserves σ's real-time order.

5.2 Wait-Freedom

We show that each invocation of enqueue and dequeue returns in a finite number of steps.

Lemma 5.8. Each enqueue operation in Algorithm 2 completes in a finite number of steps.

For lack of space, the technical proof of Lemma 5.8 is deferred to the full version of this paper [2].

Lemma 5.9. Each dequeue operation in Algorithm 3 completes in a finite number of steps.

For lack of space, the technical proof of Lemma 5.9 is deferred to the the full version of this paper [2].

Theorem 5.10. The Jiffy queue implementation is wait-free.

Proof. Lemmas 5.8 and 5.9 show that both enqueue and dequeue operations are wait-free. Therefore, the queue implementation is wait-free. □

6 PERFORMANCE EVALUATION

We compare Jiffy to several representative queue implementations in the literature: Yang and Mellor-Crummey's queue [33] is the most recent wait free FAA-based queue, denoted WFqueue; Morrison and Afek LCRQ [23] as a representative of nonblocking FAA-based queues; Fatourou and Kallimanis CCqueue [8] is a blocking queue based on the combining principle. We also test Michael and Scott's classic lock-free MSqueue [21]. We include a microbenchmark that only preforms FAA on a shared variable. This is a practical upper bound for the throughput of all FAA based queue implementations.

Table 1: Valgrind memory usage statistics run with one enqueuer and one dequeuer. I1/D1 is the L1 instruction/data cache respectively while L3i/L3d is the L3 instruction/data cache respectively.

	Jiffy	WF		LCRQ		CC		MS	
	Absolute	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Total Heap Usage	38.70 MB	611.63 MB	x15.80	610.95 MB	x15.78	305.18 MB	x7.88	1.192 GB	x31.54
Number of Allocs	3,095	9,793	x3.16	1,230	x0.40	5,000,015	x1,615	5,000,010	x1,615
Peak Heap Size	44.81 MB	200.8 MB	x4.48	612.6 MB	x13.67	420.0 MB	x9.37	1.229 GB	x28.08
# of Instructions Executed	550,416,453	5,612,941,764	x10.20	1,630,746,827	x2.96	3,500,543,753	x6.36	1,821,777,428	x3.31
I1 Misses	2,162	1,714	x0.79	1,601	x0.74	1,577	x0.73	1,636	x0.76
L3i Misses	2,084	1,707	x0.82	1,591	x0.76	1,572	x0.75	1,630	x0.78
Data Cache Tries (R+W)	281,852,749	2,075,332,377	x7.36	650,257,906	x2.3	1,238,761,631	x4.40	646,304,575	x2.29
D1 Misses	1,320,401	25,586,262	x19.37	20,037,956	x15.17	15,000,507	x11.36	11,605,064	x8.79
L3d Misses	652,194	10,148,090	x15.56	5,028,204	x7.7	14,971,182	x22.96	10,973,055	x16.82

Notice that Jiffy performs FAA only during enqueue operations, but not during dequeues. Also, measurements of Jiffy do *not* include the memory buffer pool optimization mentioned in Section 4.2.4.

Implementation: We implemented our queue algorithm in C++ [1]. We compiled Jiffy with GCC 4.9.2 with -Os optimization level and the cpp atomic library with memory_order_acq_rel and memory_order_release ordering settings. We use the C implementation provided by [33] for the rest of the queues mentioned here. They were also compiled with GCC 4.9.2 with -Os optimization level. The buffer size of our queue is 1620 entries. The segment size of Yang and Mellor-Crummey queue is 2^{10} and in LCRQ it is 2^{12}, the optimal sizes according to their respective authors.

Platforms: We measured performance on the following servers:

- AMD PowerEdge R7425 server with two AMD EPYC 7551 Processors. Each processor has 32 2.00GHz/2.55GHz cores, each of which multiplexes 2 hardware threads, so in total this system supports 128 hardware threads. The hardware caches include 32K L1 cache, 512K L2 cache and 8192K L3 cache, and there are 8 NUMA nodes, 4 per processor. The DRAM memory size is 128GB.
- Intel Xeon E5-2667 v4 Processor including 8 3.20GHz cores with 2 hardware threads, so this system supports 16 hardware threads. The hardware caches include 32K L1 cache, 256K L2 cache and 25600K L3 cache. DRAM size is 128GB.

Methodology: We used two benchmarks: one that only inserts elements to the queue (enqueue only benchmark) whereas the second had one thread that only dequeued while the others only enqueued.

In each experiment, x threads concurrently perform operations for a fixed amount of seconds, specified shortly. We tightly synchronize the start and end time of threads by having them spin-wait on a "start" flag. Once all threads are created, we turn this flag on and start the time measurement. To synchronize the end of the tests, each thread checks on every operation an "end" flag (on a while loop). When the time we measure ends we turn this flag on. Each thread then counts the amount of finished operations it preformed and all are combined with FAA after the "end" flag is turned on.

To check the sensitivity of our results to run lengths, we measure both 1 and 10 seconds runs. That is, the fixed amount of time between turning on the "start" and "end" flags is set to 1 and 10 seconds, respectively. The graphs depict the throughput in each case, i.e., the number of operations applied to the shared queue per second by all threads, measured in million operations per second (MOPS). Each test was repeated 11 times and the average throughput is reported. All experiments employ an initially empty queue.

With AMD PowerEdge for all queues we pinned each software thread to a different hardware thread based on the architecture of our server. We started mapping to the first NUMA node until it became full, with two threads running on the same core. We continue adding threads to the closest NUMA node until we fill all the processor hardware threads. Then we move to fill the next processor in the same way.

For Intel Xeon, we tested up to 32 threads without pinning whereas the server only has 16 hardware threads. This results in multiple threads per hardware thread.

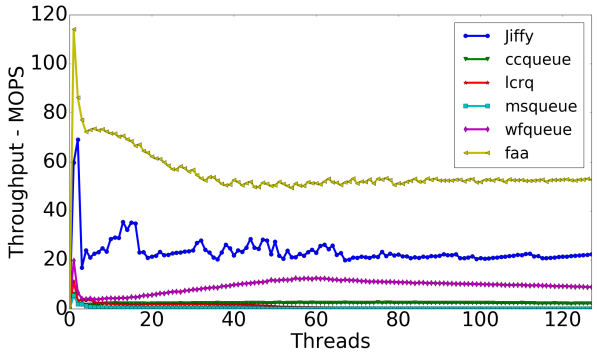
Total Space Usage. We collected memory usage statistics via valgrind version 3.13.0 [24]. We measure the memory usage when inserting 10^7 elements to the queue on AMD PowerEdge. Table 1 lists the memory usage when the queues are used by one enqueuer and one dequeuer. Jiffy's memory usage is significantly smaller than all other queues compared in this work. Jiffy uses 38.7 MB of heap memory which is 93.67% less than the WFqueue consumption of 611.63 MB, and 97% less than MSqueue. The peak heap size depicts the point where memory consumption was greatest. As can be seen Jiffy's peak is significantly lower than the rest. Jiffy's miss ratios in L1 and L3 data caches are better due to its construction as a linked list of arrays and small per-node meta-data size.

Memory Usage with 128 Threads. Table 2 lists Valgrind statistics for the run with 127 enqueuers and one dequeuer. Here, Jiffy's heap consumption has grown to 77.10 MB, yet it is 87% less than WFqueue's consumption, 87% less than CCqueue and 96.8% less than MSqueue. As mentioned above, this memory frugality is an important factor in Jiffy's cache friendliness, as can be seen by the cache miss statistics of the CPU data caches (D1 and L3d miss).

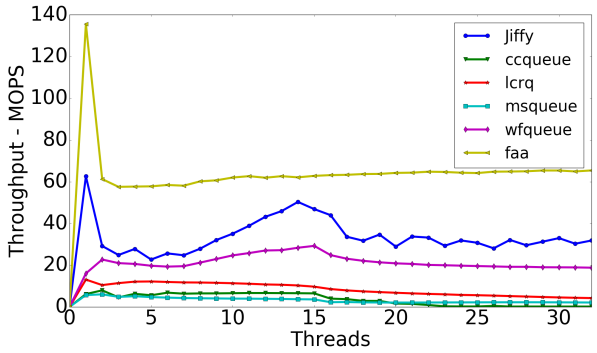
Throughput Results. Figure 6a shows results for the enqueues only benchmark on the AMD PowerEdge server for one second runs. Jiffy obtains the highest throughput with two threads, as the two threads are running on the same core and different hardware threads. The rest of the queues obtain the highest throughput with only one thread. Jiffy outperforms all queues, reaching as much as 69 millions operations per second (MOPS) with two threads. The FAA is an upper-bound for all the queues as they all preform FAA

Table 2: Valgrind memory usage statistics with 127 enqueuers and one dequeuer. I1/D1 is the L1 instruction/data cache respectively while L3i/L3d is the L3 instruction/data cache respectively.

	Jiffy	WF		LCRQ		CC		MS	
	Absolute	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Total Heap Usage	77.10 MB	611.70 MB	x7.93	1.19 GB	x15.8	605.68 MB	x7.85	2.36 GB	x31.42
Number of Allocs	6409	10045	x1.57	2819	x0.44	9922394	x1548	9922263	x1548
Peak Heap Size	87.42 MB	624.5 MB	x7.14	1.191 GB	x13.94	796.7 MB	x9.11	2.426 GB	x28.42
# of Instructions Executed	678,651,794	3,099,458,758	x4.57	1,671,748,656	x0.99	8,909,842,602	x13.13	11,944,994,600	x17.60
I1 Misses	2,238	1,724	x0.77	1,669	x0.75	1,668	x0.75	1,689	x0.75
L3i Misses	2,194	1,717	x0.78	1,658	x0.76	1,664	x0.76	1,684	x0.77
Data Cache Tries (R+W)	352,980,193	1,849,850,595	x5.24	690,008,884	x1.95	3,172,272,116	x8.99	3,237,610,091	x9.17
D1 Misses	2,643,266	51,230,800	x19.38	30,012,317	x11.35	68,349,604	x25.86	79,097,745	x29.92
L3d Misses	1,298,646	40,453,921	x31.15	19,946,542	x15.36	57,287,592	x44.11	64,219,733	x49.45

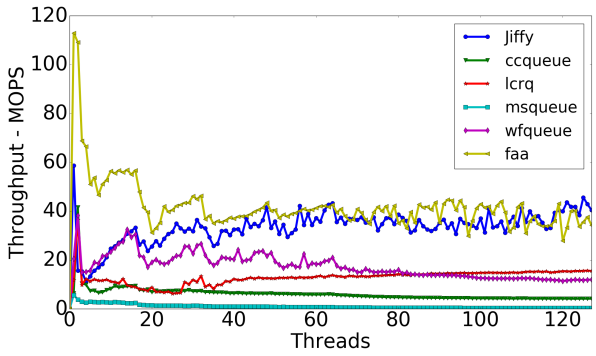


(a) AMD PowerEdge.

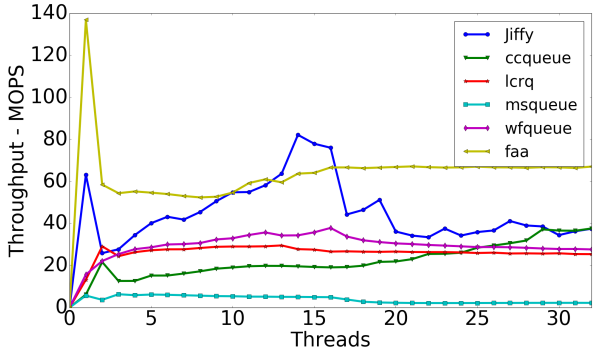


(b) Intel Xeon.

Figure 6: Enqueues only - 1 second runs.



(a) AMD PowerEdge.



(b) Intel Xeon.

Figure 7: Multiple enqueuers with a single dequeuer - 1s runs.

in each enqueue operation. The peak at 16 threads and the drop in 17 threads in Jiffy is due to the transition into a new NUMA node. At this point, a single thread is running by itself on a NUMA node. Notice also the minor peaks when adding a new core vs. starting the 2nd thread on the same core.

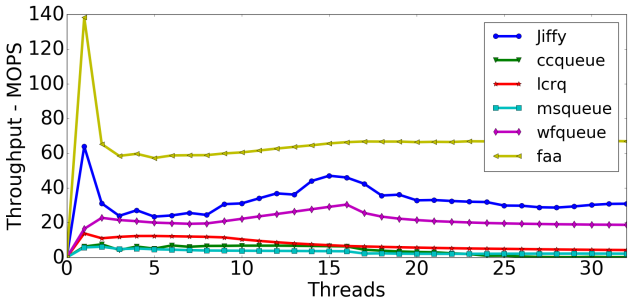
Beyond 64 threads the application already spans two CPUs. The performance reaches steady-state. This is because with 2 CPUs, the sharing is at the memory level, which imposes a non-negligible overhead. Jiffy maintains its ballpark performance even when 128 enqueuers are running with a throughput of 22 MOPS.

Figure 6b shows results for the enqueues only benchmark on the Intel Xeon server for one second runs. All queues suffer when there are more threads than hardware threads,i.e., beyond 16 threads.

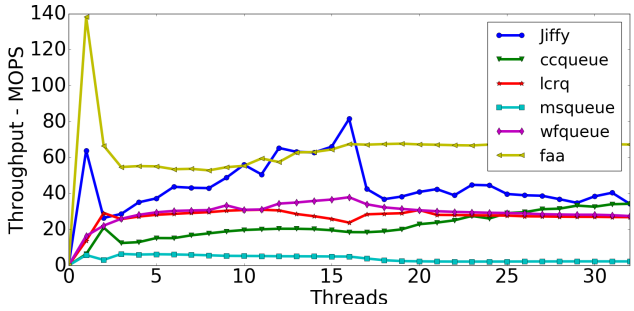
Figure 7a shows results with a single dequeuer and multiple enqueuers on the AMD PowerEdge server for one second runs. Jiffy is the only queue whose throughput improves in the entire range of the graph (after the initial 2-threads drop).

Figure 7b shows results with a single dequeuer and multiple enqueuers on the Intel Xeon server for one second runs. Here Jiffy outperforms in some points even the FAA benchmark. This is because the dequeuer of Jiffy does not preform any synchronization operations. None of the other queues can achieve this due to their need to support multiple dequeuers.

Figure 8a shows results for the enqueues only benchmark on the Intel Xeon server when the run length is set to 10 seconds. Figure 8b



(a) Enqueues only.



(b) Multiple enqueuers with a single deqeuer.

Figure 8: Intel Xeon - 10 seconds runs.

shows results for a single dequeuer and multiple enqueuers on the Intel Xeon server with 10 seconds runs. As can be seen, in both cases the results are very similar to the 1 second runs. The same holds for the AMD PowerEdge server.

Let us comment that in a production system, the enqueque rate cannot surpass the dequeue rate for long periods of time; otherwise the respective queue would grow arbitrarily. However, it is likely to have short bursts lasting a few seconds each, where a single shard gets a disproportionate number of enqueues. This test exemplifies Jiffy's superior ability to overcome such periods of imbalance.

7 CONCLUSIONS

In this paper we presented Jiffy, a fast memory efficient wait-free multi-producers single-consumer FIFO queue. Jiffy is based on maintaining a linked list of buffers, which enables it to be both memory frugal and unbounded. Most enqueue and dequeue invocations in Jiffy complete by performing only a few atomic operations.

Further, the buffer allocation scheme of Jiffy is designed to reduce contention and memory bloat. Reducing the memory footprint of the queue means better inclusion in hardware caches and reduced resources impact on applications.

Jiffy outperforms prior queues in all concurrency levels especially when the dequeuer is present. Moreover, Jiffy's measured memory usage is significantly smaller than all other queues tested in this work, ≈ 90% lower than WFqueue, LCRQ, CCqueue, and MSqueue. This makes Jiffy an attractive choice when implementing sharded in-memory data stores.

REFERENCES

[1] Dolev Adas. 2020. Jiffy's C++ Implementation. (2020). https://github.com/DolevAdas/Jiffy.

[2] Dolev Adas and Roy Friedman. 2020. A Fast Wait-Free Multi-Producers Single-Consumer Queue. *CoRR* abs/2010.14189 (2020).

[3] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The Spraylist: A Scalable Relaxed Priority Queue. In *Proc. of the ACM PPoPP*. 11–20.

[4] James Aspnes and Maurice Herlihy. 1990. Wait-Free Data Structures in the Asynchronous PRAM Model. In *Proc. of ACM SPAA*. 340–349.

[5] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. 2015. Help!. In *Proc. of ACM PODC*. 241–250.

[6] Matei David. 2004. A Single-Enqueuer Wait-Free Queue Implementation. In *International Symposium on Distributed Computing*. Springer, 132–143.

[7] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX ATC*. 373–386.

[8] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proc. of the ACM PPoPP*. 257–266.

[9] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for Efficient Pipeline Parallelism: a Cache-Optimized Concurrent Lock-Free Queue. In *Proc. of the ACM PPoPP*. 43–52.

[10] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *USENIX Annual Technical Conference (ATC)*. 301–314.

[11] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[12] Maurice Herlihy and Nir Shavit. 2011. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[13] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[14] Prasad Jayanti and Srdjan Petrovic. 2005. Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 408–419.

[15] Alex Kogan and Erez Petrank. 2011. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *Proc. of the ACM PPoPP*. 223–234.

[16] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-Free Data Structures. In *Proc. of the ACM PPoPP*. 141–150.

[17] Edya Ladan-Mozes and Nir Shavit. 2004. An Optimistic Approach to Lock-Free FIFO Queues. In *Proc. of DISC*. 117–131.

[18] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. 2013. Correct and Efficient Bounded FIFO Queues. In *25th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 144–151.

[19] Xue Liu and Wenbo He. 2007. Active Queue Management Design Using Discrete-Event Control. In *46th IEEE Conference on Decision and Control*. 3806–3811.

[20] Ben Manes. 2017. Caffeine: A High Performance Caching Library for Java 8. (2017). https://github.com/ben-manes/caffeine.

[21] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *ACM PODC*. 267–275.

[22] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *Proceedings of the ACM SPAA*. 99–109.

[23] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proc. of the ACM PPoPP*. 103–112.

[24] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic notes in theoretical computer science* 89, 2 (2003), 44–66.

[25] William N. Scherer, Doug Lea, and Michael L. Scott. 2006. Scalable Synchronous Queues. In *Proceedings of the ACM PPOPP*. 147–156.

[26] Michael L. Scott. 2002. Non-Blocking Timeout in Scalable Queue-Based Spin Locks. In *Proceedings of ACM PODC*. 31–40.

[27] Michael L. Scott and William N. Scherer. 2001. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the ACM PPOPP*. 44–52.

[28] Niloufar Shafiei. 2009. Non-Blocking Array-Based Algorithms for Stacks and Queues. In *Proceedings of ICDCN*. Springer, 55–66.

[29] Nir Shavit and Asaph Zemach. 1999. Scalable Concurrent Priority Queue Algorithms. In *Proceedings of the ACM PODC*. 113–122.

[30] Foteini Strati, Christina Giannoula, Dimitrios Siakavaras, Georgios I. Goumas, and Nectarios Koziris. 2019. An Adaptive Concurrent Priority Queue for NUMA Architectures. In *Proceedings of the 16th ACM International Conference on Computing Frontiers (CF)*. 135–144.

[31] Philippas Tsigas and Yi Zhang. 2001. A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *ACM SPAA*. 134–143.

[32] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S Nikolopoulos. 2013. Deterministic Scale-Free Pipeline Parallelism with Hyperqueues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 32.

[33] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-Free Queue as Fast as Fetch-and-Add. *Proc. of the ACM PPoPP* 51, 8 (2016), 16.