IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 6, 2019, accepted May 28, 2019, date of publication June 4, 2019, date of current version June 20, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2920781*

# Accelerating Wait-Free Algorithms: Pragmatic Solutions on Cache-Coherent Multicore Architectures

**JUNCHANG WANG[1,2], QI JIN[1], XIONG FU[1], YUN LI[1,2], AND PEICHANG SHI[3]**

[1]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China
[2]Jiangsu Key Laboratory of Big Data Security and Intelligent Processing, Nanjing 210023, China
[3]College of Computer Science, National University of Defense Technology, Changsha 410073 China

Corresponding author: Yun Li (liyun@njupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003702, in part by the National Natural Science Foundation of China under Grant 61602264 and Grant 61772030, and in part by the Primary Research and Development Plan of Jiangsu Province under Grant BE2017743.

**ABSTRACT** Parallelizing performance-critical applications are of critical importance for harnessing the power of multicore processors, which are now ubiquitous. Even though wait-free algorithms offer the appeal of completing each operation of a parallelized application in a finite number of steps, high-performance wait-free algorithms at high levels of concurrency are still rare. In this paper, we demonstrate one primary reason for this inefficiency: existing wait-free algorithms are not optimized for processors' caches and write buffers, two key components in modern hardware to accelerate memory accesses. As an example, a wait-free multi-producer-single-consumer queue algorithm, which faces common performance problems of wait-free algorithms, is studied in this paper. We accelerate the queue algorithm by (1) allowing producers to buffer enqueue requests in their local buffers and to write them into the shared queue in batch, to exhibit the spatial locality of the program, and (2) eliminating expensive atomic operations, by giving up some degree of internal consistency to avoid write buffers being drained frequently. The outcome is a write-buffer and cache-friendly queue algorithm which is wait-free and efficient on off-the-shelf multicore processors. The experiments show that the optimized queue algorithm outperforms prior queue algorithms on three different architectures (x86, Power8, and ARMv8). On x86, it outperforms WFQueue, the state-of-the-art solution, by 2-3x, and outperforms CCQueue, the representative of combining solution, by 4-12x. Applying the techniques presented in this paper to other wait-free algorithms is straightforward; our queue example demonstrates that these techniques can be applied to other wait-free algorithms while maintaining the control flow of the original algorithms without dramatic changes.

**INDEX TERMS** Multicore, wait-free algorithm, MPSC queue.

## I. INTRODUCTION

Over the past decade, programmers have been trying to parallelize applications on ubiquitous multicore processors. However, it is still challenging to parallelize and run *performance-critical* applications [1]–[4] due to the insufficient hardware support of fast communication and synchronization on commercial multicore processors. For example, on modern multicore processors, different threads must rely on shared memory, instead of dedicated hardware mechanism, to communicate and synchronize, which is not only time consuming but also susceptible to system interferes such as memory page faults.

One approach to addressing this issue is *wait-free* algorithms [5]–[9]. In contrast to other types of algorithms, such as (1) algorithms that are based on *locking*, which may cause some threads accessing shared data structures to block indefinitely, and (2) ones providing *lock-free* progress guarantee which cannot avoid one working thread to starve and delay, *wait-free* algorithms can provide the strongest progress guarantee of completing each operation in a finite number of steps [10].

While promising, existing wait-free algorithms are commonly considered inefficient [10]–[12]. Prior research works

The associate editor coordinating the review of this manuscript and approving it for publication was Muhamamd Aleem.

improve the performance of wait-free algorithms mainly at the algorithmic level [9], [13]. For example, in 2012, Kogan and Petrank [6] and subsequent works [8], [9], [14] proposed a fast-path-slow-path methodology for creating practical wait-free queues. The fast-path, usually a lock-free algorithm, provides good performance; the slow-path ensures wait-freedom.

In this paper, we argue that one other primary reason for the inefficiency of existing wait-free algorithms is that they are not optimized for processors' caches and write buffers. For example, modern processors always write data into *store buffers*, which reside in between CPU core and cache/memory to hide the overhead of writing data into caches and main memory. It is widely accepted that write buffers can help a CPU core run dozens of times faster [15]. However, the store buffer will be flushed if the hardware encounters a memory fence or an atomic instruction, which, unfortunately, has been heavily used in wait-free algorithms, including these utilizing the state-of-the-art fast-path-slow-path methodology [6], [8], [9], [14]. Another example is that a modern processor typically assumes that most applications can exhibit *temporal locality* and *spatial locality* [16] because of the fundamental data structures and algorithms (e.g., loop block and array), and hence, processors rely on *caches* to accelerate memory accesses. Existing wait-free algorithms, however, do not try to prevent multiple threads on different CPU cores from concurrently updating memory locations on the same cache line, which breaks the hardware's assumption and causes *false sharing* [16], For example, prior FIFO queue algorithms [6] allow an enqueue operation to (1) get a cell, where a value is to be inserted into, by using a fetch-and-add operation, and (2) write the value into the cell immediately. One consequence of this implementation is that, at high levels of concurrency, a group of producer threads may write data into a single cache line simultaneously, incurring severe cache misses.

In recent years, techniques such as padding data structures to avoid cache thrashing and applying relaxed memory models to programs [17] had become standard practices of good programmers. However, techniques that can explicitly benefit a CPU's caches and write buffers are neither widely recognized nor well studied. We use the term *cache-friendly* to describe wait-free algorithms that are explicitly optimized for modern CPUs' caches and write buffers. To the best of our knowledge, the only wait-free algorithm that is *cache-friendly* is the single-producer-single-consumer ring buffer (SPSC) [18], which, by performing enqueue/dequeue operations in batch, can exhibit localities and benefit CPU caches. However, the SPSC queue has a "minimum" consensus number of 2 [19]. In research, it remains an open question *if wait-free algorithms can be accelerated on modern multicore processors* and if the answer is yes, *which techniques and what is the expected speedup*.

This paper answers the above questions, by using a wait-free multi-producer-single-consumer FIFO queue (MPSC queue) as the example. We choose an MPSC queue for the following reasons. (1) MPSC queue is critical for parallelizing applications by using *pipeline parallelism* [16], [20], [21], which are usually performance-critical. (2) Its consensus number is $N$, where $N$ is the number of producer threads and is much larger than 2. (3) A wait-free MPSC queue is representative because ① multiple producer threads compete for a single cache line, and ② a suspended producer thread can block the consumer thread, two fundamental challenges that almost all of the wait-free algorithms must address.

Specifically, we first designed an MPSC queue which only relies on *Fetch-And-Add (FAA)* atomic primitive. To be wait-free, we chose the classic *helping* scheme [19]. Based on that, to be *cache-friendly*, we adopt the following key techniques. (1) A local buffer is added for each producer thread to buffer enqueue requests and to write them into the queue in batch, which can help DQueue exhibit *localities* and dramatically reduce cache misses. (2) Replacing expensive atomic operations (e.g., *Compare-And-Swap*) with regular plain access statements as many as possible, to benefit write buffers. The removal of atomic operations is not free; issues such as the non-atomic update to shared variables and properties such as that a variable could be written multiple times must be addressed/studied carefully (detailed in Section III-C). The design philosophy of our design is similar to that of RCU [22] and hazard pointers [23]; by giving up some degree of internal consistency, a wait-free algorithm can attain improved external consistency, performance, and scalability [15]. (3) Applying the above mentioned pragmatic techniques on an existing wait-free algorithm is straightforward; one can maintain the control flow of the original algorithm without dramatic changes, which makes it much easier to prove the correctness of the resulting algorithm, an expected feature in practice.

The outcome is DQueue[1] (short for Deposit Queue), an MPSC queue that is wait-free, *cache-friendly*, and hence efficient. As far as we know, this is the first *cache-friendly* wait-free algorithm with a consensus number of $N$, where $N$ is the number of producers. Our empirical study of DQueue under high levels of concurrency on three different architectures (x86, Power8, and ARMv8) shows that DQueue always utilizes a CPU's caches and write buffers better and outperforms prior queue implementations. On x86, DQueue outperforms WFQueue, the state of the art solution, by 3 times, and is 4-12 times faster than CCQueue, the representative of combining solution. DQueue can serve as an example of how to construct other *cache-friendly* algorithms.

The rest of this paper is organized as follows. We discuss related work in Section II. Section III presents our algorithm, its memory management scheme, and optimizations. Section IV proves its correctness. Section V presents the study of our queue's performance on various platforms. We conclude in Section VI.

[1] Source code of DQueue is available at https://github.com/junchangwang/dqueue under GPL v2.

## II. RELATED WORK

Michael and Scott's MS-Queue [24] is considered as a classic non-blocking queue. Even though it is lock-free and does not scale past a low level of concurrency, MS-Queue has motivated much recent work, which we classify into following categories.

### A. IMPROVING ALGORITHM

Several works [7], [25]–[27] attempt to improve MS-Queue's scalability at the algorithmic level. However, most of them still suffer the inherent CAS retry and ABA problem [28]. Addressing these problems requires extra atomic operations and memory fences, which are orders of magnitude slower than aligned read and write operations. In contrast, DQueue tries to avoid the use of atomic operations and memory fences, by giving up some degree of internal consistency.

### B. PROGRESS GUARANTEE

In 2012, Kogan and Petrank proposed a fast-path-slow-path methodology for creating practical wait-free queues [6]. In 2016, Yang proposed WFQueue [9], another implementation of fast-path-slow-path wait-free queue, which, by utilizing FAA, could be faster than previous implementations. The fast-path, usually a lock-free algorithm, provides good performance; the slow-path ensures wait-freedom. Each operation attempts the fast-path several times; if all attempts fail, it switches to the slow-path. The methodology, however, still relies on the heavy use of atomic operations. Besides, it is not *cache-friendly*.

### C. COMBINING TECHNIQUE

Another trend is combining technique [13], [29], which, by allowing the combiner to perform operations in batch, theoretically could relieve the competition at atomic operations and utilize CPU's cache better. However, experiments on CCQueue [29] show that communication between the combiner and other working threads heavily relies on memory fences which are expensive in today's multicore processors. Besides, combining queue could block [29].

### D. CACHE-FRIENDLY SOLUTIONS

Giacomoni's FastForward [18] and subsequent works [30], [31], by utilizing a cyclic array and exposing *spatial locality* of a program, can utilize CPU's caches better, with an enqueue/dequeue operation being less than 10ns. However, FastForward is an SPSC queue of consensus number of 2. It remains an open question *if the techniques presented can be applied to an algorithm of consensus number larger than 2*. DQueue answers this question by building an MPSC queue of consensus number of $N$, where $N$ is linear to the number of producers.

## III. THE DQueue ALGORITHM

### A. BASIC ALGORITHM

DQueue can be viewed as a practical realization of an ideal wait-free MPSC queue algorithm shown in Algorithm 1.[2] The basic algorithm represents the queue by using an infinite array *cell*, with two unbounded indices, *head* and *tail*, that identify the sub-array containing data.

**Algorithm 1** An Ideal Wait-Free Multi-Producer-Single-Consumer Queue.

```
1  struct Queue {integer head; integer tail; DATA_TYPE
   cell[ ]} q;
2  struct Producer {DATA_TYPE val; integer cid};
3  enqueue(Producer p, DATA_TYPE v ) {
4      p.val := v;
5      p.cid := FAA(q.tail,1);      /* Linearization
       point */
6      CAS(q.cell[p.cid], ⊥, p.val);
7  }
8  dequeue( ) {
9      if (q.cell[q.head] = ⊥) {
10         if (q.head = q.tail) return EMPTY;
11         else help_enqueue(q.head);
12     }
13     DATA_TYPE tmp := q.cell[q.head];
14     q.head := q.head + 1;        /* Linearization
       point */
15     return tmp;
16 }
17 help_enqueue( integer hid ) {
18     for (each Producer p in the system) {
19         integer cid := p.cid;
20         if (cid != hid) {
21             continue;
22         }
23         else {
24             DATA_TYPE val := p.val;
25             CAS(q.cell[cid], ⊥, val);
26             break;
27         }
28     }
29 }
```

Initially, both *head* and *tail* are set to 0, and each cell *cell[i]* is empty and contains a reserved value ⊥. Each producer records its state in a *Producer* data structure, which consists of two fields: variable *val* to record the value to be inserted into the queue, and integer *cid* to record the index of the cell where *val* is to be inserted into. DATA_TYPE could be of any type, only if the aligned read and write operation on *val* is atomic.

#### 1) ENQUEUE

An *enqueue* operation (1) first writes the value into field *val* (Line 4), (2) obtains a cell index by performing an fetch-and-add (*FAA*) operation on *tail* and writes the index into field *cid* (Line 5), (3) and then completes the enqueue operation

[2]We are not aware of this wait-free MPSC queue algorithm being explicitly presented or used before.

by storing the *val* in the cell indexed by *cid* (Line 6). Note that because the cell could be simultaneously written by an enqueue operation and a dequeue operation which is helping the enqueue (detailed in the next paragraph), a compare-and-swap (CAS) operation is required to synchronize the two concurrent updates on lines 6 and 25.

#### 2) DEQUEUE

A *dequeue* operation first checks if the cell pointed to by index *head* contains a meaningful value (Line 9). The cell does not contain a meaningful value for two reasons. (1) The queue is empty. For this case, the dequeue thread returns EMPTY (Line 10). (2) An enqueue thread has reserved the cell pointed to by index *head*, but, for some reason, the value has not been written into the cell yet. This could happen when the enqueue thread is swapped out between executing lines 5 and 6. A suspended enqueue thread could block the dequeue thread, even though other enqueue threads have successfully written values into subsequent cells, breaking the wait-free property of the queue. To solve this issue, Algorithm 1 allows the dequeue operation to adaptively help the suspended enqueue thread complete its enqueue operation (Line 11). The help strategy *help_enqueue* (detailed in the next paragraph) provides the guarantee that, by its completion, *cell[head]* contains a meaningful value. If the cell pointed to by index *head* contains a meaningful value, *dequeue* retrieves the value from the queue (Line 13), increments index *head* (Line 14), and returns.

#### 3) HELPING ENQUEUE

To prevent a suspended enqueue thread from blocking the dequeue thread, Algorithm 1 leverages Herlihy's classic helping scheme [19] and allows the dequeue thread to help the suspend enqueue thread to complete. For Algorithm 1, an enqueue request can be represented by using a *(val, cid)* pair which is stored in the enqueue thread's local *Producer* structure. Hence, on the one hand, an enqueue operation always first records its enqueue request in its local *Producer* structure *P* (Lines 4 and 5). On the other hand, each time the dequeue thread is blocked due to an incomplete enqueue request, it invokes function *help_enqueue* and passes *head* as the argument (Line 11). Function *help_enqueue* does the following to help the suspended enqueue thread. (1) It locates the enqueue thread which reserves *cell[head]*, by comparing variable *cid* of each *Producer* structure against *hid* (Line 20). If the suspended enqueue thread is found, *help_enqueue* (2) reads out the buffered value from variable *val* (Line 24), and (3) atomically writes the value into the cell by using a *CAS* operation (Line 25). The *CAS* operation succeeds if the cell's current value equals to $\perp$, which means that the enqueue thread has not stored the value into the cell yet.

### B. PERFORMANCE BOTTLENECKS AND SOLUTION

Even though Algorithm 1 is wait-free,[3] it is inefficient when applied to applications at high levels of concurrency. Experimentally, we found that DQueue has the following major performance issues.

#### 1) CACHE UNFRIENDLY

##### a: PROBLEM

We recall that the basic algorithm uses an array of *cells* to store values. Each time a producer writes a value into *cell[q.tail]* (Line 6) which does not reside on the CPU's cache, the cache line containing *cell[q.tail]* will be loaded from memory into the CPU's cache, which incurs a cache miss. Since multiple cells (e.g., 8 on 64-bit servers) reside on the same cache line, subsequent enqueue operations from this producer thread (and hence CPU) on adjacent cells (e.g., on *cell[q.tail+1]*) can be completed on caches, avoiding expensive cache misses. We refer to this behavior as *spatial locality* [16] that CPU caches can utilize to accelerate memory accesses.

Unfortunately, as the number of producers increases, Algorithm 1 exhibits less degree of *spatial locality* to hardware. The reason is that different producer threads unavoidably contend for cells. The more active producers in the system, the more active CPUs write values into the same cache line. Each time a CPU write to a shared cache line, other CPUs who are holding the same cache line in their caches must evict the cache line and reload it from memory later, which incurs expensive cache misses. This behavior is referred to as *false sharing* [16].

##### b: SOLUTION

To void *false sharing*, we explicitly introduce a local buffer for each producer thread. Each time a producer wants to enqueue a value, it first stores an enqueue request in its local buffer. Once the buffer is full, the producer thread flushes all of the buffered enqueue requests into the queue in batch, which reduces the chance of *false sharing* and allows the producer to complete a group of write requests without interference from other producer threads.

In addition to the local buffer, a global *dumping* field could be added to our algorithm and identifies the current producer thread which should be flushing its buffered enqueue requests into the shared queue. Before starting a flush operation, a producer thread checks if its thread id is equal to *dumping*. If not, the thread waits for a while and then sets *dumping* to its thread id by using a CAS primitive and starts flushing its buffer (even if the CAS fails). This divides the write operations on the cells of the queue into segments such that in each segment most operations are from the same CPU.

Introducing local buffers into a wait-free algorithm, however, is non-trivial because local buffers must be transparent to applications. Challenges come from the fact that from an application's point of view, a wait-free enqueue operation must complete in a finite number of steps, and a subsequent dequeue operation must be able to retrieve the value just enqueued. We discuss the issues and our solutions in detail in Section III-D.

[3] We omit the proof because it is similar to that of DQueue.

#### 2) EXPENSIVE ATOMIC PRIMITIVES

##### a: PROBLEM

In Algorithm 1, expensive *CAS* atomic operations are used to prevent concurrent updates on a single cell (Lines 6 and 25). It is widely recognized that *CAS* operations can deteriorate the performance of a concurrent program [6], [7], [9]. Besides, atomic operations typically contain memory fences that flush a CPU core's write buffers. Hence, it would be interesting for researchers to see if the two *CAS* operations on lines 6 and 25 could be removed. A traditional solution is to invent a new algorithm without *CAS* operations from scratch. However, as is proved by the research [32], without *CAS* operations, it is impossible to design a general wait-free queue algorithm, which has a consensus number of $N$, where $N$ is the number of producer threads.

##### b: SOLUTION

Instead of pursuing a generic solution, based on the domain knowledge of an MPSC queue, DQueue uses regular assignment statements in replace of the two *CAS* operations on lines 6 and 25, and the resulting algorithm is shown in the following code snippet.

The removal of the CAS operations brings the following two issues:

**(1) Non-atomic access to (val, cid) pair:** To help a suspended enqueue operation, *help_enqueue* needs to retrieve the corresponding *(val, cid)* pair, which can be found in the producer thread's local state. Since the (*val*, *cid*) pair is two 64-bit integers that cannot be read or written atomically, without any special care, when function *help_enqueue* reads the two variables, it may interleave with the producer thread which is concurrently updating these two variables. That is, the consumer thread may read out a mixed (*val*, *cid*) pair. For example, suppose ($val_1$, $cid_1$) is a legal enqueue request, ($val_2$, $cid_2$) is another subsequent legal enqueue request from the same producer thread, and $cid_1 < cid_2$, then *help_enqueue* may retrieve either the legal request $(val_1, cid_1)$ or an illegal mixed pair ($val_2$, $cid_1$) if the producer thread is performing enqueue operations concurrently.

To address this issue, our algorithm adopts a pragmatic solution that can synchronize concurrent *enqueue* and *help_enqueue* operations only with word-sized read and write operations. Specifically, as shown in Algorithm 2, *help_enqueue* reads out an *(val, cid)* pair (Lines 38 and 37) in the reverse order that they were written into a producer thread's local variables (Lines 31 and 32), such that the value of *val* read out belongs to the enqueue request that corresponds to the cell of index *cid* or belongs to a subsequent enqueue request that corresponds to a subsequent cell. That is, *help_enqueue* can retrieve either the legal request $(val_1, cid_1)$ or the illegal request $(val_2, cid_1)$, but not the illegal request $(val_1, cid_2)$.

To identify the illegal enqueue request $(val_2, cid_1)$, *help_enqueue* fully utilizes the knowledge of a linearizable FIFO queue: Once the producer thread starts buffering enqueue request ($val_2$, $cid_2$) (and hence the helper can read out the value of $val_2$), the enqueued request ($val_1$, $cid_1$) must have been successfully enqueued and linearized, and the cell of index $cid_1$ must have been containing a meaningful value. As a result, once *help_enqueue* retrieves an enqueue request, it checks if the value of *cell[cid]* equals to $\perp$ (Line 39). If not, which means the *help_enqueue* may retrieve an illegal enqueue request, *help_enqueue* returns without helping this enqueue request. Otherwise, *help_enqueue* helps store *val* into the cell (Line 40). That is, the following lemma holds.

**Algorithm 2** The Improved MPSC Queue Without CAS Operations. Differences Between Alg. 1 and this Alg. are Highlighted in Bold Font

```
30 enqueue(Producer p, DATA_TYPE v ) {
31     p.val := v;
32     p.cid:= FAA(q.tail,1);
33     q.cell[p.cid] := p.val;
34 }
35 help_enqueue( integer hid ) {
36     for (each Producer p in the system) {
37         integer cid := p.cid;
           /* The if-else statement has
              been removed                    */
38         DATA_TYPE val := p.val;
39         if (q.cell[cid] = ⊥) {
40             q.cell[cid] = val;
41         }
42     }
43 }
```

*Lemma 1: For any cell[cid], where cid* $\in \{0, 1, 2, \ldots\}$*, the value that enqueue wants to write into the cell (on line 33) and the value that help_enqueue wants to write into the cell (on line 40) are the same.*

*Proof:* Without loss of generality, we assume that ($val_1$, $cid_1$) is a legal enqueue request, that ($val_2$, $cid_2$) is a subsequent legal enqueue request from the same producer thread, and that $cid_1 < cid_2$. *help_enqueue* may retrieve one of the following four enqueue requests: (1) $(val_1, cid_1)$, (2) $(val_2, cid_2)$, (3) $(val_1, cid_2)$, and (4) $(val_2, cid_1)$. Obviously, requests (1) and (2) are correct because then *help_enqueue* will write the correct value into the correct cells. We then prove that *help_enqueue* can discard both requests (3) and (4).

We use $write_A(x = v)$ to denote the event in which $A$ assigns value $v$ to variable $x$, and $read_A(x = v)$ to denote the event in which $A$ reads $v$ from variable $x$. In the following proof, since *enqueue* is the only thread that writes to variables, and *help_enqueue* is the only thread that reads these variables, we omit thread symbol $A$ without introducing any ambiguity. One event $e_1$ *precedes* another event $e_2$, written $e_1 \prec e_2$, if $e_1$ occurs at an earlier time. We assume a sequentially consistent memory model (detailed in Section IV).

We prove that *help_enqueue* cannot retrieve request (3) using contradiction. Suppose that *help_enqueue* read out an

enqueue request (3), by inspecting the code of *help_enqueue*, we get that:

$$read(cid = cid_2) \prec read(val = val_1)$$

By inspecting the code of *enqueue*, we get that:

$$write(val = val_1) \prec write(cid = cid_1) \prec write(val = val_2) \prec write(cid = cid_2)$$

To make sure that *help_enqueue* can retrieve $cid_2$, we get that:

$$write(cid = cid_2) \prec read(cid = cid_2)$$

It follows that:

$$write(val = val_1) \prec write(val = val_2) \prec read(val = val_1)$$

This observation yields a contradiction because no thread can read the value $val_1$ out of variable *val* after it has been overwritten by value $val_2$, a contradiction. Hence, *help_enqueue* cannot retrieve request (3).

Interestingly, the above reasoning shows that *help_enqueue* may retrieve request (4). We prove that request (4) can be discarded by *help_enqueue* on Line 39. Specifically, by inspection the code of *enqueue*, we know that it writes the buffered enqueue request into the corresponding cell (Line 33) before a subsequent enqueue operation starts. Hence, we get that:

$$write(val{=}val_1) \prec write(cid{=}cid_1) \prec write(cell[cid_1]{=}val_1) \prec write(val{=}val_2) \prec write(cid{=}cid_2)$$

Suppose that *help_enqueue* retrieves an request (4), by inspecting the code of *help_enqueue*, we get the follow order. Note that the function read $cell[cid_1]$ on line 39.

$$read(cid = cid_1) \prec read(val = val_2) \prec read(cell[cid_1] = ?)$$

To make sure that *help_enqueue* can retrieve $val_2$, we get that:

$$write(val = val_2) \prec read(val = val_2)$$

It follows that:

$$write(cell[cid_1] = val_1) \prec read(cell[cid_1] = ?)$$

Since once $cell[cid_1]$ is set to $val_1$ it remains. It follows that the read operation $read(cell[cid_1])$ returns value $val_1$. As a result, the request (4) retrieved will be discarded because it checks if $cell[cid_1]$ has been containing a meaningful value (Line 39) before writing into the cell. Overall, *help_enqueue* may retrieve request (4), but *help_enqueue* discards the request and never writes it into the corresponding cell. □

**(2) Written-twice:** Another side effect of removing *CAS* operations is that a cell could be written twice, one by function *enqueue* on Line 33, and another by function *help_enqueue* on Line 40. This happens when the queue is close to empty; the dequeue operation helps a producer write the buffered enqueue request into a cell, and later the enqueue thread writes the value into the cell again. For example, suppose there is a producer thread *P*, a consumer thread *C*, and an empty queue in which *head* equals to *tail*, if *P* and *C* interleave in the following order, the cell pointed to by *head* and *tail* will be written twice:

1) *P* stores the value in local variable *val* (line 31),
2) *P* performs an FAA on *tail* and stores the index in local variable *cid* (line 32),
3) *P* is suspended,
4) *C* checks if the cell has meaningful value and invokes *help_enqueue*,
5) *help_enqueue* reads out the (*val*, *cid*) pair (Lines 38 and 37) written by *P*,
6) *help_enqueue* writes *val* into the cell (line 40),
7) *P* resumes,
8) *P* writes *val* into the cell (line 33).

The consequence is that a single cell could be written twice. We use the term *written-twice* to describe this property. To prove that the *written-twice* property does not affect the value read out by the dequeue operation in Section IV, we first prove the following important lemma.

*Lemma 2: The value of a cell that has been written by an enqueue operation on Line 33 or a help_enqueue operation on Line 40 cannot be changed to a numerically different value by future enqueue or help_enqueue operations.*

*Proof:* Lemma 1 shows that values that are written into the cell by both *enqueue* and *help_enqueue* are the same. Besides, on modern architectures, word-aligned word-sized stores are atomic. Hence, once a cell has been written once by either an *enqueue* or *help_enqueue* operation, its value cannot be changed further, even if it could be written twice. □

### 3) LOCATING THE EXACT SUSPENDED PRODUCER

#### a: PROBLEM

In Algorithm 1, each time function *help_enqueue* is invoked, it only helps the producer who is going to write a value into the cell where the consumer thread is looking for data (Line 20). Even though the algorithm is correct, it is inefficient on modern cache coherent systems. The major reason is that to locate the exact suspended producer, *help_enqueue* must walk through all of the producers' states to check their *cids*. During this process, all of the producers' states, including the (*val*, *cid*) pair, are loaded into the cache of the CPU running function *help_enqueue*. Besides, once function *help_enqueue* writes a value into a cell, the cache line containing the cell has been loaded into the cache. As a result, writing other values into adjacent cells could be very fast because of *spatial locality*.

#### b: SOLUTION

To solve this performance issue, in Algorithm 2, each time function *help_enqueue* is invoked, instead of looking for the exact suspended producer thread, the function walks through all of the producer threads in the system (Line 36) and helps all of the producer threads that have suspended enqueue requests.

Note that this strategy could introduce *false positives* because function *help_enqueue* may help producers that are active and do not need help at all. Fortunately, Lemma 2

guarantees that Algorithm 2 is correct because even if *help_enqueue* incidentally helps a producer that can perform enqueue operations by itself, the consumer can read out the right value.

### C. BUILDING BLOCKS OF DQueue

DQueue is built on top of Algorithms 1 and 2. With optimization techniques discussed in Section III-B, DQueue can utilize CPU caches and write buffers more efficiently.

**Algorithm 3** Structures and Auxiliary Methods of DQueue

```
44 struct Segment {Segment * next; integer id;
   DATA_TYPE cell[N];};
45 struct Queue {integer head; integer tail; Segment *
   qseg;} q;
46 struct Request {DATA_TYPE val; integer cid;};
47 struct Producer {Request local_buffer[L]; integer
   local_head; integer local_tail; Segment *pseg};
48 struct Consumer {Segment *cseg};
49 Segment * new_segment(int id) {
50     Segment * s := new Segment;
51     s->id := id;
52     s->next := NULL;
53     for (each int i in [0, N)) s->cell[i] := ⊥;
54     return s;
55 }
56 Segment * find_segment(Segment * sp, int cid) {
57     Segment * curr := sp;
58     for (i := curr->id; i < cid / N; i++) {
59         Segment * next := curr->next;
60         if (next = NULL) {
61             Segment * new := new_segment(i+1);
62             if (CAS(curr->next, next, new)) next :=
               new;
63             else free(new);
64         }
65         curr := next;
66     }
67     return curr;
68 }
69 wrap(int i) {return (i % L);}
70 next(int i) {return wrap(i+1);}
```

**Global state:** Algorithm 3 presents the data type used to implement DQueue. The queue itself is represented by structure *mpsc_queue*, which is similar to the queue structure in Algorithm 1, except that the infinite array *cell* being replaced by a singly-linked list, *qseg*. Each segment in the list consists of (1) a *next* pointer pointing to the next segment, (2) a monotonically increasing *id*, and (3) a data array. By default, the size of the data array is 1024. Initially, *qseg* points to the first segment, with its *id* being zero. We denote the segment with id $i$ as *segment[i]* and the $j$-th cell in a segment as *cell[j]*. Thus, cell of index $k$ locates at *cell[k%N]* in *segment[k/N]*. We refer to the singly-linked list of *Segment* as *segment list*. Fields *tail* and *head* of the queue, similar to the basic algorithm, are both 64-bit long integers. We thus make the realistic assumption that for DQueue, the maximum number of data enqueued and dequeued does not exceed $2^{64}$ on 64-bit machines. Initially, both *tail* and *head* are set to 0.

**Thread-local state:** The local state of each producer thread is stored in structure *Producer*. During initialization, all of the producer instances are linked in a singly-linked list by using their *next* pointers. Each *Producer* contains a segment pointer *pseg*, which initially points to *segment[0]*. *pseg* is a local cache of the starting segment from which, given a cell index $i$, the producer thread starts searching *segment[i/N]*, by invoking function *find_segment()*. Each time a producer thread moves to a new segment in searching the corresponding segment of a given cell index, *pseg* is updated accordingly. Each *Producer* contains an array of *Request*, which is the structure to record enqueue requests. For each enqueue operation, the producer thread first records the enqueue request by recording its (*val*, *cid*) pair. Indices *local_head* and *local_tail* are used to indicate the sub-array of *local_buffer* that contains enqueue requests. Note that both *local_head* and *local_tail* are only updated by the producer thread. The local state of the consumer thread, *Consumer*, has a single field *cseg*, which points to the segment where the consumer starts searching the corresponding segment for a given cell index.

**find_segment:** Recall that *segment list* is a singly-linked list of structure *Segment*. Given a cell index $i$, function *find_segment* traverses the list and returns a pointer to *segment[i/N]*. The function accepts two parameters: *sp* is a pointer to the segment where *find_segment* starts searching the expected cell, and *cid* is the index of the cell. If *find_segment* reaches the last segment before finding the cell, it invokes function *new_segment* to allocate a new segment (Line 61) and tries to append the new segment to the list by using the *CAS* operation (Line 62) on the last segment's *next* pointer. The *CAS* operation could fail because some other threads may have already appended a new segment to the *next* pointer. In that case, the thread frees the segment it failed to append and continues the traversal with the segment just appended. Eventually, function *find_segment* returns a pointer to *segment[cell_id/N]*.

We now turn to a walk-through of DQueue's pseudocode in the following two subsections.

### D. WAIT-FREE ENQUEUE

The flow of the wait-free enqueue operation is presented in Figure 1. For simplicity, Figure 1 considers the cases in which the first producer thread executes *enqueue* without interference from other threads. The pseudocode is presented in Algorithm 4. When a producer thread invokes the *enqueue* operation, it first checks if its local buffer becomes full (Line 82). *local_buffer* is a pre-allocated ring buffer. If *local_tail* is not equal to *local_head*, which means that the local buffer is not full, the producer thread stores the

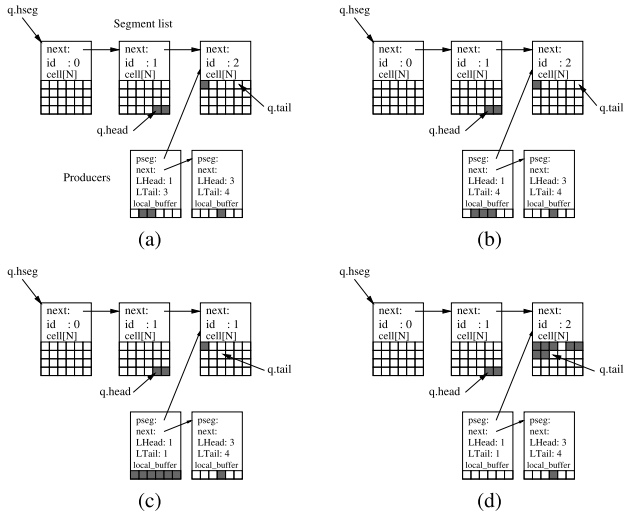



**FIGURE 1.** The flow of the *enqueue* operations performed solely by the first producer. *LHead* is short for *local_head*, and *LTail* for *local_tail*. (a) Some state of DQueue. Three values are buffered in two producers' local buffers. (b) The first producer performs an enqueue operation. It first stores the enqueue request in its local buffer, and then increments *LTail*. (c) The first producer performs other three enqueue operations, and its local buffer becomes full. Thus, the first producer flushes its buffered requests into *segment list*. (d) The first producer has successfully performed a flush operation. Note that *segment[2].cell[3]* is empty because the corresponding enqueue request is still buffered in the second producer's local buffer.

**Algorithm 4** Wait-Free Enqueue of DQueue

```
71 void dump_local_buffer( Producer p ) {
72     while ( p.local_head != p.local_tail ) {
73         Request r = p.local_buffer[p.local_head];
74         DATA_TYPE val := r.val;
75         Segment * seg := find_segment(p.pseg, r.cid);
76         seg->cell[r.cid % N] := val;
77         p.local_head := next(p.local_head);
78         if (p.pseg != seg) p.pseg := seg;
79     }
80 }
81 void enqueue( Producer p, DATA_TYPE v ) {
82     if (next(p.local_tail) = p.local_head) {
83         dump_local_buffer( p );
84     }
85     int tail := p.local_tail;
86     p.local_buffer[tail].val := v;
87     p.local_buffer[tail].cid := FAA(q.tail, 1);
       /* Linear. point */
88     p.local_tail := next(p.local_tail);
89 }
```

(*val*, *cid*) pair in its *local_buffer* (Lines 86–87) and increments *local_tail* by one (Line 88). If the local buffer becomes full, *enqueue* operation calls *dump_local_buffer* to flush buffered enqueue requests into *segment list* in batch (Line 83).

Function *dump_local_buffer*, as its name suggests, traverses a producer thread's local buffer, reads out each enqueue request (Lines 73–74), locates the corresponding segment (Line 75), and then writes the *val* into the corresponding cell (Line 76). Note that the field *pseg* works as the starting point segment where function *find_segment* starts from searching the segment containing the cell of index *cid*. Note that each time a producer thread finds that the segment containing expected cell has moved forward (Line 78), it updates its *pseg* (Line 78) accordingly, to accelerate next invocation of *find_segment*.

It is worth noting that DQueue provides concurrency at the level of producer threads; different producer threads are permitted to invoke function enqueue() in parallel. However, a producer thread must sequentially invoke function enqueue(), and hence the situation, where there are two or more enqueue requests of the same producer thread in the system, will not happen. We made this design choice for the following two reasons. (1) Because of the first-in-first-out property of queue algorithms, a producer thread must, in effect, invokes function enqueue() sequentially. Therefore, even if we can design new algorithms that allow function enqueue() to accept concurrent enqueue requests from a single producer thread, applications utilizing this type of queue algorithms could be extremely hard to design due to the raising timing issues. Therefore, applications using FIFO queues to transition data typically invokes function enqueue() synchronously, and the current design of DQueue is applicable for this design pattern. (2) If a programmer really wants the flexibility to issue concurrent enqueue requests from a single producer thread, he/she can create multiple working threads within this producer thread. Each of these working threads is for an enqueue request that must be issued concurrently. Each working thread can have its own instance of struct Producer, and hence they can run concurrently without any modification of DQueue. This strategy works for most multi-threading programming environments (e.g., POSIX) because they allow a programmer to efficiently create and manage threads within an existing thread.

### E. WAIT-FREE DEQUEUE

The flow of the wait-free dequeue operation is presented in Figure 2, and the pseudocode is given in Algorithm 5. Function *dequeue* is similar to Algorithm 1. The first step of a *dequeue* operation is to locate the cell of index *q.head* (Lines 104 - 106). Field *cseg* works as the starting point segment where *find_segment* start searching. Field *cseg* is updated if *dequeue* has moved to a subsequent segment (Line 105). Function *help_enqueue* is invoked if one or more producer threads need help to enqueue. That is, the producer threads have reserved one or more cells, where the consumer thread is going to read values out, but the producer threads have not written meaningful values into *segment list* yet.

Function *help_enqueue* is similar to the basic algorithm, except that instead of helping a single enqueue request, *help_enqueue* traverses the *local_buffer* of each producer thread, and help complete all of the buffered enqueue requests (from *local_buffer[0]* to *local_buffer[L-1]*) (Line 92).

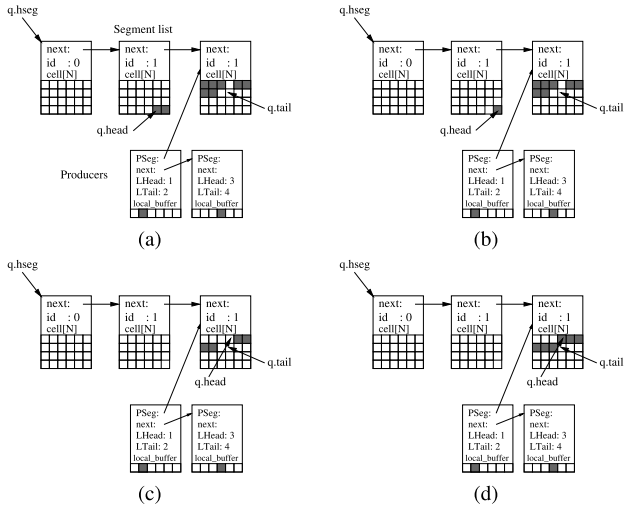



**FIGURE 2.** **The flow of the *dequeue* operations performed by the consumer thread. *LHead* is short for *local_head*, and *LTail* for *local_tail*. (a) Some state of DQueue. Two values are stored in two producers' local buffers. (b) Consumer performs a dequeue operation. It first reads out one value, and then increments *q.head* by one. (c) Consumer continuously read out values, until it reaches *segment[2].cell[3]* which does not contain the meaningful value yet. The corresponding enqueue request is buffered in the second producer's local buffer. (d) Consumer helps producers by writing their buffered enqueue requests into *Segment list*. Note that Consumer does not update producers' local states, and hence producers will later perform enqueue operations once again.**

Note that a producer may have written buffered enqueue requests into *segment list* and its *pseg* has moved forward, while the *help_enqueue* is in progress. For this case, *help_enqueue* skips this producer (Line 96).

### F. MEMORY MANAGEMENT

Most prior wait-free algorithms assume that they run on top of a memory management scheme that is both efficient and wait-free. Such a scheme, however, is one fundamental challenge facing programmers who are trying to utilize concurrent objects in practice. In this section, we present how DQueue can efficiently manage its memory in a wait-free manner.

#### 1) MEMORY RECLAMATION

The only garbage that needs reclamation in DQueue is the *retired* segments that are no longer in use. We use the term *retired* to describe a segment if (1) all of the producers have moved past it and (2) every enqueued value in this segment has been dequeued by the consumer.

To reclaim retired segments, there are two common solutions. (1) *Pointer-based reclamation* (e.g., hazard pointers [23]), in which producer threads reserve those segments they are actually using. This approach is robust but tends to have expensive programming and run-time overhead; it typically forces programmers to first explicitly "register" the pointer to the segment it is accessing and then "unregister" the pointer each time the pointer is no longer needed. Besides, a memory fence is required right after registering the pointer to a segment, and a second fence right before unregistering the pointer. (2) *Epoch-based reclamation* [33], in which a global *epoch counter* is incremented. A thread performing an operation records the instant value of *epoch counter*, right before it starts accessing dynamically allocated memory, implicitly reserving all of the dynamically allocated memory that have not been announced to be freed before the instant value.

**Algorithm 5** Pseudocode for Wait-Free Dequeue. *help_enqueue* is Called by Dequeue When the Cell Pointed by *Head* Does Not Contain Meaningful Value, and *Head* is Behind *Tail*, Which Means There is at Least One Suspended Enqueue Request.

```
90  void help_enqueue( ) {
91      for (each Producer p in the system) {
92          for (each Request r in p.local_buffer) {
93              int pos := r.cid;
94              DATA_TYPE val := r.val;
95              Segment * seg := find_segment(p.pseg,
                pos);
96              if (seg->id > pos/N) break;
97              if (seg->cell[pos % N] = ⊥) {
98                  seg->cell[pos % N] := val;
99              }
100         }
101     }
102 }
103 DATA_TYPE dequeue( Consumer c ) {
104     Segment * seg := find_segment(c.cseg, q.head);
105     if (c.cseg != seg) c.cseg := seg;
106     DATA_TYPE cell := seg->cell[q.head % N];
107     if (cell = ⊥) {
108         if (q.head = q.tail) return empty;
109         else help_enqueue( ) ;
110     }
111     q.head := q.head + 1; /* Linearization
        point                                  */
112     return cell;
113 }
```

#### 2) RECLAMATION SCHEME IN DQueue

DQueue leverages a customized *epoch-based reclamation* [33] scheme for the following reasons. (1) *epoch-based* mechanism does not require extra memory fences. In contrast, pointer-based mechanism is over-weight because memory fence instructions flush the write buffer of CPU cores. (2) Epoch-based mechanism is relatively more scalable because the pointer-based reclamation mechanism requires a thread to know the maximum number of concurrently-accessed memory blocks in advance. In contrast, epoch-based reclamation does not have such a limitation. (3) The integration of the epoch-based mechanism into our algorithm requires no modification of the algorithm.

**Algorithm 6** Garbage Collection of DQueue

```
114 void GC( ) {
115     Segment * min, * safe, * max;
116     while (true) {
117         min := q.hseg;
118         safe := max := c.hseg;
119         for (each producer p in the system) {
120             if (p.pseg.id < safe.id) safe := p.pseg;
121         }
            /* Reclaim segment[i], i∈[min,
               safe)                          */
122         for (each segment s in sublist [min, safe)) {
123             Segment * to_be_freed = min;
124             min = min->next;
125             free(to_be_freed);
126         }
            /* Reclaim segment[i], i∈[safe,
               max)                           */
127         for (each segment s in sublist [safe, max)) {
128             skip := 0;
129             for (each producer p in the system) {
130                 for (each request r in local_buffer) {
131                     if (r.cid/N = s.id) {skip := 1;
                        break;};
132                 }
133                 if (skip = 1) break;;
134             }
135             if (skip = 0) delete segment s from the list;
136         }
137     }
138 }
```

Specifically, when DQueue is initialized, a dedicated thread *GC* is created to repeatedly scan *segment list* and reclaim retired segments. Note that the garbage collection mechanism in DQueue could be a function call which is invoked each time a producer moves its *pseg* pointer forward to a new segment. For this type of garbage collection, a lock is required to protect concurrent accesses from multiple producers. For the simplicity of presentation, we show the standalone version of *GC* in the paper. Algorithm 6 shows the pseudocode for our memory reclamation scheme. At each iteration, *GC* initially attempts to reclaim every segment, i.e., segment[i], where $i \in [q.hseg, c.cseg)$. *q.hseg* is the head of the *Segment list* and *c.cseg* points to the segment where the consumer starts searching cells. The *GC* thread uses indices *min* and *max* to denote these two segments (Lines 117–118). However, a producer thread may still hold references to segments in between *min* and *max* because the producer thread's *pseg* pointer has not been updated yet. To avoid reclaiming segments that are still in use by these producer threads, *GC* inspects every producer thread by checking *pseg* of each producer. *GC* finds all of the segments that (1) are in between *min* and *max* and (2) are in use by any of the producer thread. *GC* records the minimum id of these segments by using variable *safe* (Lines 119–121). GC is safe to reclaim segment[i], where $i \in [q.hseq, safe)$.

### 3) AVOID BLOCKING

Segments in between *[safe, max)* may not be safe to reclaim because producers are pointing to some of them. This is a notable issue in the *epoch-based* reclamation scheme: if a thread stalls in the middle of a data structure operation, the memory reclamation thread will be prevented from reclaiming the memory block, which the suspended thread is accessing, and subsequent memory blocks [34]. Researchers typically use the term *robust* to describe a memory management scheme that does not suffer from this problem. To address this issue, we invented a customized reclamation scheme, which, by leveraging the domain knowledge of DQueue, is efficient and *robust*. Specifically, in reclaiming segments in *[safe, max)* the following domain knowledge can be used: (1) there is a single thread (*GC*) that is performing deletion operations, and (2) there is no thread inserting new nodes into *[safe, max)*. As a result, surprisingly, reclaiming segments in list *[safe, max)* can be formalized as follows. One and only one thread (*GC*) deletes nodes from a singly-linked list *[safe, max)* and skips some specific nodes which are known in advance.

The bottom half of Algorithm 6 presents the pseudocode for reclaiming segments in *[safe, max)*. For DQueue, a producer's local buffer may contain enqueue requests that will be written into *segment[i]*, where i∈*[safe, max)*. Note that even though the consumer may have helped write a value into a segment, the producer will write the value into the segment again (Section III-B). By checking producer threads' *local_buffer*, we can identify these segments. Specifically, for each segment in list [*safe*, *max*), *GC* traverses local buffers of all of the producers and checks if any of the buffered enqueue requests is referring to this segment (Lines 129–134). Note that it is not necessary to check the producer's *pseg* because at least one of its enqueue requests is referring to the segment which *pseg* points to. If the segment is in use, *GC* skips this segment; otherwise, the segment can be freed. By inspecting the code of *GC*, we can get the following Lemma:

*Lemma 3: The maximum number of retired segments that cannot be freed is $P * L$, where $P$ is the number of producers, and $L$ is the size of local buffer of each producer thread.*

*Proof:* A retired segment cannot be freed because one or more buffered enqueue requests in all of the local buffers are referring to it. A system utilizing DQueue has $P$ producers, each of which has a local buffer of size $L$. Hence, the system can buffer in maximum $P * L$ requests which can refer to in maximum $P * L$ retired segments that cannot be freed. □

*Lemma 4: For a DQueue containing $V$ values, the maximum number of segments in use is $P * L + \lceil V/N \rceil$, where $P$ is the number of producers, $L$ is the size of each local buffer, and $N$ is the size of each segment.*

*Proof:* Values in DQueue are stored in adjacent cells. Hence, a DQueue containing $V$ values uses in maximum

$\lceil V/N \rceil$ segments to store these values. Besides, Lemma 3 shows that the maximum number of segments that cannot be freed is $P * L$. Overall, the maximum number of segments in DQueue is $P * L + \lceil V/N \rceil$. □

#### 4) MEMORY ALLOCATION

The only memory that needs dynamic allocation in DQueue is segments that are appended at the tail of *segment list* (Line 62). In practice, DQueue uses the *jemalloc* memory allocator [35] to prevent memory allocation from being a bottleneck. Building a wait-free memory allocation scheme for DQueue is achievable because the minimum number of segments required by DQueue is given in advance (Lemma 3).

## IV. CORRECTNESS

In this section, we prove that DQueue is linearizable and wait-free.

### A. MODEL

To prove that DQueue is correct, we must first clarify the concepts that provide a foundation for our design.

#### 1) SYSTEM

We design our queue for an *asynchronous shared memory system* [36]. On such a system, a program is executed by $p$ deterministic threads, where $p$ may exceed the number of physical processors. A scheduler decides which producer threads to run and may suspend execution of any thread at any time for arbitrarily long. Besides, the scheduler in our system accepts hints to avoid swapping out the consumer thread and the GC thread. This could be closely achieved in Linux by utilizing a real-time scheduler (SCHED_FIFO). Besides, reading and writing aligned word-sized variables are atomic on the system.

#### 2) MEMORY MODEL

To simplify the presentation of the pseudocode for our queue, we assume a *sequential consistency* memory model. However, today's compilers and hardware are free to reorder instructions for performance [37]. To prevent any undesired reordering, especially reordering of ordinary memory accesses, memory fences and compiler directives are necessary to guarantee the order in which CPUs access different shared memory locations. We omit memory fences and compiler directives in the paper; Interested readers are referred to the source code.

#### 3) ATOMIC PRIMITIVES

We model memory as an array of 64-bit values. We use the notation *m[a]* for the value stored in address *a* of the memory. DQueue uses the following atomic primitives:

– *read(a)*: returns *m[a]*;
– *write(a, v)*: stores value *v* into *a*;
– *FAA(a, v)*: returns *m[a]* and store *m[a]+v* into *a*;
– *CAS(a, b, v)*: if *m[a]* = *m[b]*, stores *v* into *a* and returns *true*; otherwise, stores *m[a]* into *b* and returns *false*.

The x86 architecture supports hardware *FAA* and *CAS* primitives. IBM Power and ARM architectures, however, support Load-Linked/Store-Conditional (LL/SC) in hardware, and hence programmers typically use a *LL/SC* retry loop to emulate *FAA* and *CAS*. Emulating *FAA* with *LL/SC* retry loops sacrifices the wait-free property and performance. The wait-free property issue can be relieved by utilizing the doorway mechanism in Lamport's Bakery algorithm [38]. For the performance issue, experimentally we found that on Power8 and ARMv8, the retry loop by *LL/SC* is not the performance bottleneck of the algorithm (Section V-C).

### B. LINEARIZABILITY

We denote the head and tail indices of $L$ as $H_L$ and $T_L$, respectively, and the cell at index $k$ in $L$ as $L[k]$, $k \in \{0, 1, 2, \ldots\}$. We denote an eqnueue operation that enqueues value into $L[k]$ as $E_k$ and a dequeue that dequeues from $L[k]$ as $D_k$. $\overline{D}$ is to denote a dequeue that returns *empty*. The DQueue algorithm is linearizable [36] because both enqueue and dequeue have specific "linearization points." For DQueue, we have the following lemma:

*Lemma 5: An arbitrary execution history $W$ of DQueue can be linearized to the sequential execution of an MPSC queue.*

*Proof:* An enqueue operation $E_k$ is linearized on Line 87 that assigns the value of $T_L$ to local *cid* and increments $T_L$ by using *FAA*. $E_k$ linearizes on this line because it "takes effect" on this line, by allowing subsequent $D_k$ operation to successfully retrieve the value. Obviously, the *FAA* is performed by the enqueue operation and it must happen during the execution interval of $E_k$.

An dequeue operation $D_k$ is linearized at Line 111 that increments $H_L$ by one. Note that no atomic operations are required on this line because DQueue is a single-consumer queue. $D_k$ linearizes on this line because it notifies the *GC* that the value of *cell[$H_L$]* has been successfully retrieved, and that it can be reclaimed. If the queue is empty, the dequeue operation $\overline{D}$ is linearized at line 108. As a result, a dequeue operation's linearization point is always within the execution interval of $D_k$. □

Now we prove that DQueue is linearizable. We use $W = \{e_j : j = 0, 1, 2, \ldots\}$ to denote a possible infinite execution history of DQueue. We assume that the events in $W$ have been properly assigned linearization points according to the last paragraph. We denote the linearization point of an operation *op* as $e_{j(\overline{D})}$, and the precedence ordering between two operations as $op_1 \prec op_2$.

*Lemma 6: For $k \in \{0, 1, 2, \ldots\}$, any linearized history sequence of DQueue safisfies: (1) $E_k \prec E_{k+1}$, (2) $D_k \prec D_{k+1}$, (3) $E_k \prec D_k$, and (4) $H_L = T_L$ at every $e_{j(\overline{D})}$.*

*Proof:* Condition (1) means that the linearization point of an enqueue operation $E_k$ precedes that of the subsequent enqueue operations, including $E_{k+1}$. This is always true for DQueue because an enqueue operation linearizes at the point $T_L$ is incremented using an *FAA* atomic operation, and $T_L$ increments monotonically.

Condition (2) is obviously true because DQueue is a single-consumer queue, and a dequeue operation linearizes at the point $H_L$ is incremented and $H_L$ increments monotonically.

Condition (3) means that a dequeue is always linearized after its matching enqueue. We show that this condition holds using contradiction. Suppose $D_k \prec E_k$ at cell $k$, which means before the enqueue operation linearizes on Line 111, either ① *cell[k]* contains a meaningful value (Line 107), or ② *cell[k]* is empty but *help_enqueue* fails to write the buffered enqueue request into the queue (Line 98). By inspecting the code, we see that condition ② does not hold because *help_enqueue* in Algorithm 5 flushes all of the buffered enqueue requests of all of the producer threads into the queue. Hence, $D_k \prec E_k$ means that condition ① is true and that *cell[k]* contains a meaningful value before $E_k$ is linearized. However, by inspecting the code, we see that *enqueue* operation always first linearizes an enqueue operation (Line 87) and then writes the meaningful value into the cell (Line 76), a contradiction.

Condition (4) means that once a dequeue returns empty, $H_L$ equals to $T_L$. This is obviously true because a dequeue operation compares these two variables on line 108 before returning *empty*. □

*Theorem 1: DQueue is linearizable.*

*Proof:* Lemma 5 shows that an arbitrary execution history of DQueue can be linearized to a sequential history, and Lemma 6 proves that the sequential history is correct. Thus, DQueue is linearizable. □

### C. WAIT FREEDOM

We prove that DQueue is wait-free by showing that both the enqueue operation and dequeue operation are guaranteed to complete within finite steps.

*Lemma 7: Each enqueue operation completes in a bounded number of steps.*

*Proof:* An enqueue operation always first writes the value into its local buffer. If its local buffer becomes full, it will dump the local buffer into *segment list* by performing $L$ word-aligned write operations, where $L$ is the size of a producer's local buffer. One other instruction the enqueue operation must perform is *FAA* (Line 87), which is wait-free on X86. The enqueue operation may invoke the memory allocator (Line 75) and append a new segment to the tail of *segment list*. Section III-F shows that a wait-free memory allocator for DQueue is achievable and that appending a new segment is wait-free because the enqueue operation will use the appended segment if it failed in appending a new segment. Overall, an enqueue operation can complete in a bounded number of steps. □

*Lemma 8: Each dequeue operation completes in a bounded number of steps.*

*Proof:* If the queue is not close to empty, a dequeue operation performs a group of aligned read and write operations. Otherwise, function *help_enqueue* is invoked (Line 109), and in the worst case, it helps write $P * L$ values into *segment list*, where $P$ is the size of a producer's local buffer, and $L$ is their local buffer size. For both cases, a dequeue operation can complete in a bounded number of steps. □

*Lemma 9: The memory reclamation scheme completes in a bounded number of steps.*

*Proof:* For GC, the worst case scenario happens when a producer thread suspends for a long time and the segment referred to by the producer cannot be freed by using GC's fast-path (Lines 122–126). Since the *epoch-based* reclamation scheme is used, subsequent segments cannot be freed too. For this case, the slow-path (Lines 127–136) performs $S * P * L$ aligned read/write operations, where $S$ is the length of the segment list that cannot be freed, $P$ the number of producer threads, and $L$ the size of a producer's local buffer. Lemma 3 shows that the size of $S$ is in maximum $P * L$, and as a result, each GC iteration performs in maximum $(P * L)^2$ operations, which is bounded. □

In conclusion, we have shown the following.

*Theorem 2: The DQueue implementation presented is wait-free.*

## V. EVALUATION

This section presents the performance evaluation of MPSC queue implementations. Experiments show that on widely-used x86 processors, our algorithm (1) outperforms WFQueue, the state-of-the-art solution, by 2-3x, (2) runs 4-12x faster than CCQueue, the representative of combining solution, (3) and runs 7-60x faster than LTQueue, the latest research dedicated to MPSC queue. On Power8 and ARMv8 processors, despite that *FAA* is emulated by a *LL/SC* retry loop and becomes the performance bottleneck, DQueue outperforms all other queue implementations. Besides, DQueue's performance is workload invariant, insensitive to core placement, and is robust to the number of concurrent producer threads.

### A. EVALUATION METHODOLOGY

#### 1) EVALUATED ALGORITHMS

We compare DQueue to several representative queue implementations in the literature. In experiments, we use Yang's WFQueue [9] as the representative of wait-free queue implementations based on fast-path-slow-path design scheme [6]. We choose CCQueue by Fatourou [29] as the representative of queues based on the classic combining principle. We also compare DQueue with LTQueue [39], which, as far as we know, is the latest research dedicated to MPSC queue. Note that CCQueue is a blocking queue. WFQueue and CCQueue are MPMC queues and are used as MPSC queues in experiments by limiting the number of consumers to one. Note that, as far as we know, all of the prior MPSC queues suffer from performance degradation at a high level of concurrency. For example, LTQueue is orders of magnitude slower than DQueue (Figure 4) when more than 20 CPU cores are used. In contrast, the state-of-the-art MPMC queues perform much better. Therefore, for a fair comparison, we compare DQueue to both the MPSC (LTQueue) and MPMC (WFQueue and CCQueue) queues in experiments.

We implemented DQueue and LTQueue in C. For WFQueue and CCQueue, we reuse the implementation developed and publicly released by the authors of [9]. We compile each code with GCC 5.4.0 on all platforms where Ubuntu 16.04.5 is installed. We use −O3 as our optimization level without any special optimization flags.

**TABLE 1. Summary of experimental platforms.**

| Processor Model | Speed(Hz) | #Sockets | #Cores | LLC | FAA |
|---|---|---|---|---|---|
| Intel Broadwell | 2.6 G | 2 | 56 | 35 M | Yes |
| Intel Ivy Bridge | 2.6 G | 2 | 24 | 15 M | Yes |
| IBM Power8 | 3.4 G | 1 | 8 | 8 M | No |
| Cavium ARMv8 | 2.0 G | 2 | 96 | 16 M | No |

#### 2) HARDWARE PLATFORMS

We evaluated the performance of aforementioned queue implementations on four platforms based on the following processors: Intel Broadwell, Intel Ivy Bridge, IBM Power8, and Cavium ARMv8 ThunderX. Table 1 lists the key characteristics of these platforms. The last column (FAA) indicates if the architecture supports native hardware *FAA* atomic primitive. Since Power8 and ARMv8 do not directly implement a hardware *FAA* primitive, we implement *FAA* in these two platforms by using a retry loop that employs *load-linked* and *store-conditional* (LL/SC), which is a common strategy to support the missing *FAA* operation on these two platforms. It is important to note that implementing *FAA* with LL/SC could (1) lead to performance degradation (demonstrated in Section V-C), and (2) sacrifices the wait-freedom of DQueue and WFQueue because of the potential for unbounded retries in theory.

#### 3) TESTBED

The testbed consists of multiple producer threads and one consumer thread, each of which is mapped to a dedicated CPU core. In mapping producer threads to CPU cores, we use a *performance-first* mapping. For example, on the Broadwell server with two 14-core processor, where each core supports two *Hyper-Threading* threads, we map the consumer thread to core 0, the first producer thread to core 1, the next 12 producer threads to other cores on the same processor, the next 14 producer threads to the same processor by utilizing *Hyper-Threading*, and the last 28 threads to the second processor. Experiments performed on a single CPU socket are marked with an *, experiments performed on multiple CPU sockets are marked with a #, and experiments in which producer threads oversubscribe CPU cores are marked with a *!*. For each experiment, the consumer thread executes $10^7$ dequeue operations, and the producer threads execute $10^7$ enqueue operations partitioned evenly among all producer threads. Each enqueue operation consists of enqueuing a value into the queue, spinning on a specified number of cycles. Similarly, for each dequeue operation, the consumer thread extracts a value out of the queue and spins for a while to simulate workload. Spinning is used to approximate work in a controlled fashion to isolate queue performance. Each plotted data point, unless specified otherwise, consists of the sample mean and sample standard deviation of 20 trials.

### B. PERFORMANCE COMPARISON

We first compare DQueue against other representative queue implementations with different workloads. For performance comparison, the 56-core Intel Broadwell server was used, and Time Stamp Counter on x86 was used to simulate workload. In each experiment, all algorithms were evaluated with six different work period durations (0ns for the ideal test case without workload, 50ns, 100ns, 200ns, 400ns, and 800ns).

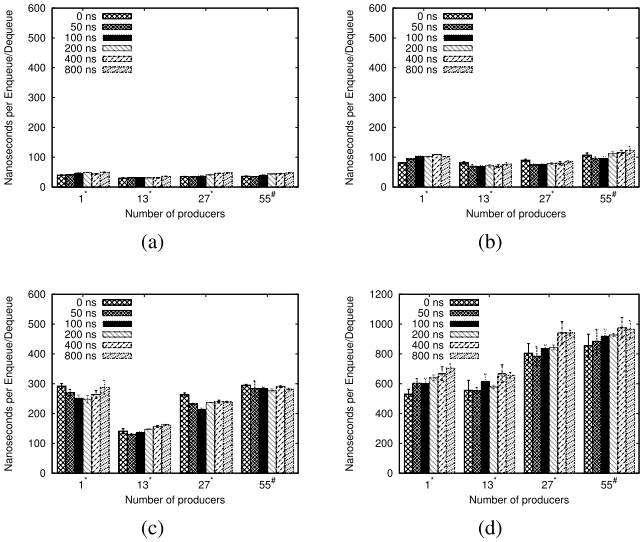



**FIGURE 3. Queue performance comparisons on the 56-core Intel Broadwell server [workload vs. number of producer threads]. (a) DQueue. (b) WFQueue. (c) CCQueue. (d) LTQueue.**

Figure 3 compares the performance of DQueue against other queue implementations, with respect to workload and number of producer threads. Each graph shows the average per operation costs for a single enqueue or dequeue operation. For this experiment, we vary the number of producer threads and the workload of each producer thread. For DQueue the segment size is set to 1024 and the size of a producer's local buffer is 64. When a single producer thread is used (the first column), the queue to be evaluated becomes a single-producer-single-consumer queue, another classic lock-free algorithm. The second column (with 13 producers) represents the scenario where all of the 13 producer threads and a single consumer thread are mapped to the same CPU die, and by carefully choosing the mapping strategy, each thread is mapped to a dedicated physical CPU core. That is, *Hyper-Threading* is turned off. The third column (with 27 producers) represents the scenario where all of the 27 producer threads and a single consumer thread are mapped to the same CPU die. For this case, *Hyper-Threading* is turned on. The last column (with 55 producers) represents the scenario in which all of the 55 producer threads and a single consumer thread are mapped to the two CPU dies and occupy all of the CPU cores on this server.

The main observation is that DQueue is more efficient than other queue implementations, and DQueue is insensitive to the number of concurrent producer threads and the simulated workload, while other queue implementations are not. Specifically, DQueue takes 30-51 nanoseconds with typical standard deviations being less than 1 nanosecond to perform an enqueue or dequeue operation successfully. In contrast, WFQueue, the next best solution, takes 71-123ns with standard deviations being about 8 ns. Overall, DQueue is about 2.4x faster than WFQueue, 5.4-8.8x faster than CCQueue, and 17.7-30.2x faster than LTQueue. Note that the y-axis of Figure 3d has a larger range because it performs much worse than other queue implementations. Further, except DQueue, all of the other three queue implementations exhibit large variations in the average enqueue/dequeue cost, showing that the algorithm is not stable on the testbed.

**TABLE 2. Performance counter statistics. (a) 1 Producer. (b) 27 Producers. (c) 55 Producers.**

(a)

| | DQueue | WFQueue | CCQueue | LTQueue |
|---|---|---|---|---|
| Cores utilized | **1.75** | 1.995 | 1.253 | **1.85** |
| Branch miss rate | **0.12%** | **0.32%** | 0.18% | 2.40% |
| L1 miss rate | **0.78%** | **4.58%** | 4.99% | 7.99% |
| Cache miss rate | **0.01%** | **0.19%** | 42.86% | 17.33% |
| SB_DRAIN(million) | **15.72** | **260.97** | **501.66** | 1,533.94 |

(b)

| | DQueue | WFQueue | CCQueue | LTQueue |
|---|---|---|---|---|
| Cores utilized | **10.99** | 5.78 | 6.63 | **1.29** |
| Branch miss rate | **0.28%** | **0.43%** | 1.61% | 1.96% |
| L1 miss rate | **3.71%** | **7.43%** | 11.68% | 24.28% |
| Cache miss rate | **4.75%** | **24.33%** | 36.39% | 25.349% |
| SB_DRAIN(million) | **156.98** | **260.06** | **2,300.18** | 2,559.70 |

(c)

| | DQueue | WFQueue | CCQueue | LTQueue |
|---|---|---|---|---|
| Cores utilized | **22.15** | 11.91 | 10.183 | **1.30** |
| Branch miss rate | **0.40%** | **0.43%** | 1.11% | 2.02% |
| L1 miss rate | **3.78%** | **7.22%** | 6.07% | 32.38% |
| Cache miss rate | **16.20%** | **49.21%** | 42.33% | 20.47% |
| SB_DRAIN(million) | **263.55** | **301.10** | **4,686.57** | 3,335.36 |

Table 2 explains why DQueue performs better. By utilizing Linux *Perf*, we collect the performance statistical data, including the number of CPU cores utilized by the program (denoted *Cores utilized*), *Branch miss rate*, *L1 cache miss rate*, the miss rate of caches of all levels (denoted *Cache miss rate*), and the number of store buffer drains (performance event 0x0704 on Intel Xeon and denoted as *SB_DRAIN*).

For each enqueue or dequeue operation, DQueue uses a single *FAA* primitive. In contrast, WFQueue requires an *FAA* and a *SWAP* in the fast path. Besides, the helping scheme in DQueue is designed to minimize the number of atomic operations and memory fences. In contrast, WFQueue's helping scheme heavily relies on atomic operations and memory fences to synchronize and contains a dozen branches. Table 2 shows that the WFQueue's *Branch miss rate* and *CPU write buffer drain* is much higher than DQueue. Besides, the local buffer in DQueue decreases the chance of cache false-sharing, and hence can dramatically reduce DQueue's cache miss rate. That's the reason DQueue has lower *cache miss rates*, shown in Table 2.

It is reasonable that CCQueue has a relatively higher *cache miss rate* because CCQueue fundamentally relies on a singly-linked list; traversing a list incurs much higher cache miss rate than walking through an array. So it is fair that we do not compare *cache miss rate* against CCQueue. Another major performance issue in CCQueue is that the combiner and the working threads communicate extremely frequently. In practice, the communication involves memory fences, which forces CPU cores to flush their write buffers. Table 2 shows that CCQueue typically drains CPU cores' write buffers 17-31 times more often than DQueue does, one major reason that CCQueue is slower than DQueue.

For LTQueue, even though its producers have local buffers to store data, they must synchronize to identify the cell with minimum timestamp value from these distributed local buffers. Hence, the synchronization becomes the bottleneck. Table 2 shows that the CPU cores utilized by LTQueue is less than 2, even if 55 producer threads are concurrently inserting data into the queue.

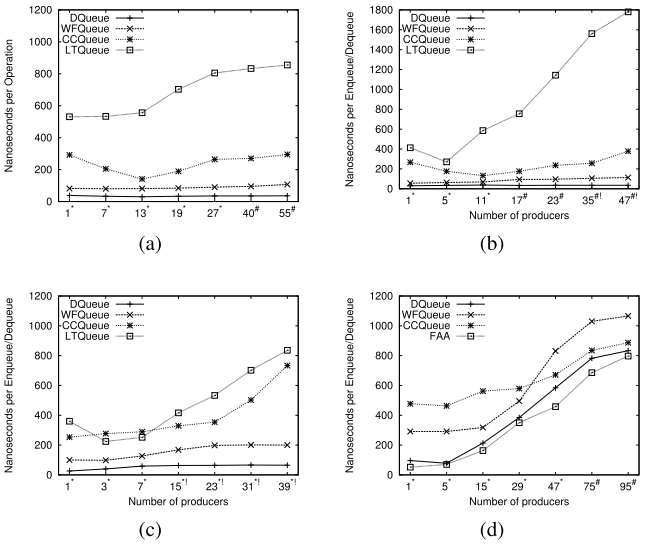



**FIGURE 4. Performance evaluation on different architectures. (a) Intel Broadwell. (b) Intel Ivy Bridge. (c) IBM Power8. (d) ARMv8.**

### C. PERFORMANCE ON DIFFERENT ARCHITECTURES

Figure 4 shows the experiment results of the four queue implementations on different platforms. For ARMv8, we omit the results of LTQueue because they are similar to those on PowerPC. We instead show the results of a baseline benchmark *FAA* which is used to evaluate the cost of concurrent *FAA* operations on a shared variable. Specifically, in the *FAA* benchmark, what an enqueue operation does is to

increment the shared variable by one to provide a practical upper bound for the cost of an *FAA* operation.

#### 1) SINGLE-PROCESSOR PERFORMANCE

In this experiment, we use at most 27 producer threads on Broadwell, 11 on Ivy Bridge, 7 on Power8, and 47 on ARMv8. Figure 4 shows that DQueue outperforms all other queue implementations on all platforms. DQueue outperforms WFQueue, the state-of-the-art solution, by $2-3\times$ on Broadwell, Ivy Bridge, and Power8. On ARMv8, DQueue is about $4\times$ faster than WFQueue when the number of producer threads is less than 5. As the number of producer threads increases (e.g., $>29$), the performance of DQueue decreases. This is because of the high contention of *FAA* primitive, which is simulated by using a LL/SC loop. Despite that, DQueue is $1.3\times$ faster than WFQueue.

On Broadwell and Ivy Bridge, the performance of CCQueue improves as the number of cores increases. This is because as the number of producer threads increases, the combiner can perform a larger group of operations in batch, which allows hardware to perform optimizations. However, this advantage disappears at a higher level of concurrency when two CPUs are used.

#### 2) MULTI-PROCESSOR PERFORMANCE

On Broadwell and Ivy Bridge, DQueue is insensitive to the number of CPUs used, performance of WFQueue slightly decreases, and the performance of both CCQueue and LTQueue deteriorate sharply. For example, the experiment with 23 producer threads on Ivy Bridge shows that DQueue is about $3\times$ faster than WFQueue, $5\times$ faster than CCQueue, and $20\times$ faster than LTQueue. Figure 4d shows that the performance of DQueue is very close to that of the *FAA* benchmark, which demonstrates that the performance deterioration of DQueue on ARMv8 is mainly due to *FAA* operations. Despite that, DQueue is still faster than other queue implementations.

#### 3) OVERSUBSCRIBED WORKLOADS

Problems related to blocking usually occur in oversubscribed scenarios, in which the number of producer threads exceeds the number of hardware threads and forces the operating system to context switch between threads. If a critical thread (e.g., the combiner in CCQueue) is scheduled out, the algorithm cannot make progress until it runs again. We show this by increasing the number of producer threads beyond the number of hardware cores. Figure 4b and 4c show that the performance of CCQueue deteriorates sharply for oversubscribed workloads on both Ivy Bridge and Power8. In contrast, DQueue maintains its peak performance.

### D. SEGMENT SIZE SENSITIVITY STUDY

The segment size plays an important role in the performance of memory management and hence the performance of DQueue. Given the fundamental structure of DQueue is a singly-linked list of *Segment*, intuitively, as the segment size increases the memory management subsystem has less workload in allocating new segments and in reclaiming retired segments.

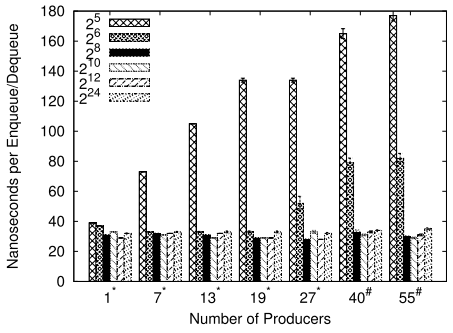



**FIGURE 5.** DQueue performance vs. segment size.

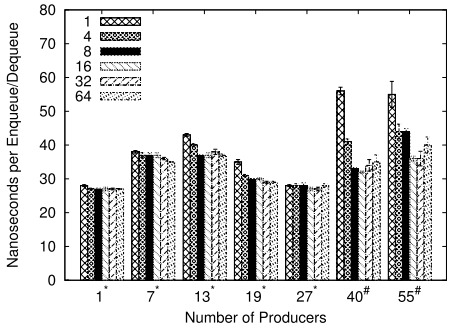



**FIGURE 6.** DQueue performance vs. local buffer size.

To quantify this effect, we evaluate DQueue at different concurrency levels on the Broadwell server with various segment sizes. Figure 5 shows that, as long as the segment size is larger than $2^8$, the performance of DQueue is insensitive to its segment size, no matter how many producer threads exist in the system.

Note that when the segment size is set to $2^{24}$, which is larger than the number of values enqueued into the queue in a test, we in effect turn off the memory management subsystem because all of the enqueue/dequeue operations happen in a single segment. However, as shown in Figure 5, a pre-allocated huge segment does not benefit DQueue, which demonstrates that in practice the memory management subsystem in DQueue is efficient.

### E. BATCH SIZE SENSITIVITY STUDY

Another key parameter in DQueue is the size of each producer's local buffer which helps reduce cache false-sharing when producers writing data into the shared queue. To quantify this effect, we test DQueue on an initially empty queue at different concurrency levels on the Broadwell server with various batch sizes. Figure 6 shows that, as long as its batch size is larger than 16, DQueue is insensitive to its batch size.

## VI. CONCLUSIONS

This paper explores two pragmatic techniques for accelerating wait-free algorithms on modern multicore processors.

These two techniques can help a wait-free algorithm leverage a multicore processor's caches and write buffers better. Besides, they can be applied to a wait-free algorithm while maintaining the control flow of the original algorithm without dramatic changes. As far as we know, this is the first research in the literature exploring these pragmatic techniques in wait-free algorithms. As an example, we present DQueue, an accelerated multi-producer-single-consumer queue, and show that DQueue is efficient and scalable on a variety of benchmarks using three widely-used architectures.

We believe the techniques presented in this paper can guide the design of more complicated algorithms, such as MPMC queues and hash tables.

## REFERENCES

[1] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting parallelism to scale software routers," in *Proc. SOSP*, Oct. 2009, pp. 15–28.

[2] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. SIGCOMM*, Sep. 2010, pp. 195–206.

[3] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures," in *Proc. ICS*, Jun. 2009, pp. 204–213.

[4] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable multi-core packet processing in user-space," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oct. 2012, pp. 27–38.

[5] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proc. PPOPP*, Feb. 2011, pp. 223–234.

[6] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," in *Proc. PPOPP*, Feb. 2012, pp. 141–150.

[7] A. Morrison and Y. Afek, "Fast concurrent queues for ×86 processors," in *Proc. PPOPP*, Feb. 2013, pp. 103–112.

[8] S. Arnautov, P. Felber, C. Fetzer, and B. Trach, "FFQ: A fast single-producer/multiple-consumer concurrent FIFO queue," in *Proc. IPDPS*, May /Jun. 2017, pp. 907–916.

[9] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proc. PPOPP*, Mar. 2016, Art. no. 16.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[11] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit, "Obstruction-free algorithms can be practically wait-free," in *Proc. DISC*, 2005, pp. 78–92.

[12] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" 2017, *arXiv:1701.00854*. [Online]. Available: https://arxiv.org/abs/1701.00854

[13] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proc. SPAA*, Jun. 2011, pp. 325–334.

[14] Y. Peng and Z. Hao, "Fa-stack: A fast array-based stack with wait-free progress guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 843–857, Apr. 2018.

[15] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it?" Linux Technol. Center, IBM Beaverton, Hillsboro, OR, USA, Tech. Rep. 2011(16), 2011.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[17] *Built-in Functions for Memory Model Aware Atomic Operations*. Accessed: Mar. 9, 2019. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

[18] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," in *Proc. PPOPP*, Feb. 2008, pp. 43–52.

[19] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[20] J. Giacomoni, J. K. Bennett, A. Carzaniga, D. C. Sicker, M. Vachharajani, and A. L. Wolf, "Frame shared memory: Line-rate networking on commodity hardware," in *Proc. ANCS*, Dec. 2007, pp. 27–36.

[21] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: High-level and efficient streaming on multicore," *Programming Multi-Core and Many-Core Computing Systems*. 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119332015

[22] M. Desnoyers, P. E. McKenney, A. S. Stern, M. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, Feb. 2012.

[23] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[24] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. PODC*, 1996, pp. 267–275.

[25] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proc. SPAA*, 2001, pp. 134–143.

[26] W. N. Scherer, D. Lea, and M. L. Scott, "Scalable synchronous queues," *Commun. ACM*, vol. 52, no. 5, pp. 100–111, May 2006.

[27] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank, "Bq: A lock-free queue with batching," in *Proc. SPAA*, Jul. 2018, pp. 99–109.

[28] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 51, no. 1, pp. 1–26, May 1998.

[29] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. PPOPP*, Feb. 2012, pp. 257–266.

[30] J. Wang, K. Zhang, X. Tang, and B. Hua, "B-queue: Efficient and practical queuing for fast core-to-core communication," *Int. J. Parallel Program.*, vol. 41, no. 1, pp. 137–159, Feb. 2013.

[31] V. Maffione, G. Lettieri, and L. Rizzo, "Cache-aware design of general-purpose single-producer–single-consumer queues," *Software: Pract. Exper.*, vol. 49, no. 5, pp. 748–779, May 2019.

[32] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," in *Proc. ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 487–498, Jan. 2011.

[33] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, Dec. 2007.

[34] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott, "Interval-based memory reclamation," in *Proc. PPOPP*, Jan. 2018, pp. 1–13.

[35] J. Evans, "Scalable memory allocation using jemalloc," *Notes Facebook Eng.*, Jan. 2011. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/

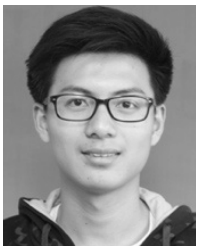
[36] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[37] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "×86-TSO: A rigorous and usable programmer's model for ×86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[38] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, Aug. 1974.

[39] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," in *Proc. Int. Conf. Found. Softw. Technol. Theor. Comput. Sci.*, 2005, pp. 408–419.

**JUNCHANG WANG** received the Ph.D. degree in computer science from the University of Science and Technology of China, in 2014. He is currently a Lecturer with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interest includes parallel and distributed computing systems.



**QI JIN** received the bachelor's degree from the Nanjing University of Posts and Telecommunications, in 2018, where he is currently pursuing the degree in computer science. His research interests include parallel computing systems and data storage.

**XIONG FU** received the B.S. and Ph.D. degrees in computer science from the University of Science and Technology, Hefei, in 2002 and 2007, respectively. He is currently a Professor with the School of Computer Science, Nanjing University of Posts & Telecommunications. His research interests include parallel and distributed computing and cloud computing.

**YUN LI** received the Ph.D. degree in computer science from Chongqing University, Chongqing, China. He was the Postdoctoral Fellow with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is the Principal Investigator (PI) of several national scientific research projects and provincial projects in recent years. He is currently a Professor with the School of Computer Science, Nanjing University of Posts and Telecommunications, China. He has published more than 50 refereed research papers in AAAI, ECML, ICASSP, IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, and *Pattern Recognition*. His research mainly focuses on machine learning, data mining, and cloud computing. He is the member of the IEEE Data Mining and Big Data Analytics TC. He has served in many international conferences as a program chair, publication chair, or TPC member and has also served on international journals as a guest editor for some special issues.

**PEICHANG SHI** received the Ph.D. degree from the National University of Defense Technology (NUDT), in 2012, Changsha, China, where he is currently an Associate Professor. His research interests include operating systems and distributed computing technology.