IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 20, 2020, accepted May 17, 2020, date of publication May 25, 2020, date of current version June 5, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2997071*

# EQueue: Elastic Lock-Free FIFO Queue for Core-to-Core Communication on Multi-Core Processors

**JUNCHANG WANG[1,2], YANGFENG TIAN[3], AND XIONG FU[1]**
[1]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210003, China
[2]Jiangsu Key Laboratory of Big Data Security and Intelligent Processing, Nanjing 210003, China
[3]School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Xiong Fu (fux@njupt.edu.cn)

This work was supported 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** In recent years, the number of CPU cores in a multi-core processor keeps increasing. To leverage the increasing hardware resource, programmers need to develop parallelized software programs. One promising approach to parallelizing high-performance applications is pipeline parallelism, which divides a task into a serial of subtasks and then maps these subtasks to a group of CPU cores, making the communication scheme between the subtasks running on different cores a critical component for the parallelized programs. One widely-used implementation of the communication scheme is software-based, lock-free first-in-first-out queues that move data between different subtasks. The primary design goal of the prior lock-free queues was higher throughput, such that the technique of batching data was heavily used in their enqueue and dequeue operations. Unfortunately, a lock-free queue with batching heavily depends on the assumption that data arrive at a constant rate, and the queue is in an equilibrium state. Experimentally we found that the equilibrium state of a queue rarely happens in real, high-performance use cases (e.g., 10Gbps+ network applications) because data arriving rate fluctuates sharply. As a result, existing queues suffer from performance degradation when used in real applications on multi-core processors. In this paper, we present EQueue, a lock-free queue to handle this robustness issue in existing queues. EQueue is lock-free, efficient, and robust. EQueue can adaptively (1) shrink its queue size when data arriving rate is low to keep its memory footprint small to utilize CPU cache better, and (2) enlarge its queue size to avoid overflow when data arriving rate is in burstiness. Experimental results show that when used in high-performance applications, EQueue can always perform an enqueue/dequeue operation in less than 50 CPU cycles, which outperforms FastForward and MCRingBuffer, two state-of-the-art queues, by factors 3 and 2, respectively.

**INDEX TERMS** Lock-free queue, pipeline parallelism, multi-core processors.

## I. INTRODUCTION

In recent years, great efforts have been made in parallelizing high-speed applications [1]–[7] to utilize the state-of-the-art processors that bring an increasing number of CPU cores. One promising approach to parallelizing applications is via *pipeline parallelism* [8], in which a program is divided into a chain of subtasks, and these subtasks are assigned to different CPU cores. Each time a data element arrives, the first CPU core performs the first subtask on the data and then passes the data to other cores for remaining subtasks. Leveraging *pipeline parallelism*, however, is still challenging on existing multi-core processors that lack efficient core-to-core communication mechanisms.

A large body of works attempted to address this issue by leveraging single-producer-single-consumer (SPSC) first-in-first-out (FIFO) lock-free queues (henceforth *queues*) [4]–[7], [9]–[13] as the communication mechanism between subtasks. These algorithms typically avoid lock operations and shared control variables between the enqueue and dequeue threads to increase parallelism. Besides, the technique of *batching* is heavily used in enqueue and dequeue operations for higher throughput. Experimental results show that it takes these queues about 20 CPU cycles to perform an enqueue or dequeue operation [5], making the lock-free queue a promising solution to fast core-to-core communication on multi-core processors.

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

Unfortunately, in parallelizing high-performance applications by leveraging pipeline parallelism, we found that the performance of state-of-the-art queue implementations deteriorates dramatically, and the performance of the parallelized applications is limited by the queues. This performance deterioration has been noticed in literature [5], [7], [11], but, unfortunately, has not been well studied, let alone solved. When looking into existing lock-free queues, we found that the primary reason is as follows. The major design goal of state-of-the-art queue implementations was higher communication *throughput*. To that end, the technique of *batching* has been heavily used in both enqueue and dequeue operations. With batching, the producer buffers data elements and sends a group of data into the queue in batch, and the consumer pulls them out in one iteration. For example, to achieve the peak performance of throughput, Lynx [6] buffers data in one operating system page, which is typically 4KB in size, and sends the data of the whole page to the consumer in a single enqueue operation.

However, to leverage *batching*, the queue algorithms typically assume that data arrive at a constant rate and that the queue is in an equilibrium state [14]. Otherwise, a queue will become full or empty frequently, resulting in buffer overflow or increased tail latency. A constant data arriving rate, however, rarely happens in real applications. For example, researchers have found that for a web server which is parallelized and run on multi-core processors, the packet-arriving rate fluctuates sharply [15]. As a result, the data passed from one subtask to others will never stay at a constant rate. If one of the existing lock-free queues is used in the parallelized applications as the communication scheme, the queue will become *FULL* or *EMPTY* frequently, dramatically decreasing its performance [10].

We were recently facing this robustness issue in building a multi-10Gbps Intrusion Detection System (IDS) on multi-core servers, in which existing queues take up to 100 CPU cycles to perform an enqueue/dequeue operation (detailed in Section VII), which is 5 times slower compared to their peak performance. To solve this issue, we present EQueue, a novel Elastic lock-free FIFO Queue, for parallelizing high-performance applications by leveraging *pipeline parallelism*. EQueue is efficient and robust, such that it works well in situations where data arriving rate fluctuates sharply. The basic idea behind the robustness of EQueue is to adjust its queue size adaptively. When the data arriving rate is low (the queue will become empty), EQueue adaptively shrinks its queue size to keep its memory footprint as small as possible, which is extremely helpful in high-performance applications to avoid cache misses (especially, the L1 cache). On the other hand, to handle the burstiness of the data arriving rate (the queue will become full), EQueue enlarges its queue size to buffer more data, avoiding handling expensive buffer overflow.

Experimental results show that EQueue is robust and efficient in real applications, in which EQueue performs an enqueue/dequeue operation in 50 CPU cycles, which outperforms FastForward and MCRingBuffer, two of the best-known solutions, by factors of 3 and 2, respectively.

Our contributions can be summarized as follows:

- Existing queues are not designed for one kind of important use cases where data arriving rate fluctuates sharply. This paper presents EQueue, an elastic and efficient lock-free FIFO queue that works as the communication scheme for parallelizing applications on multi-core processors. As far as we know, EQueue is the first research from literature to handle the robustness issue of lock-free FIFO queues.
- We optimize EQueue to make it efficient and robust when used in high-performance applications. On the one hand, EQueue can adaptively adjust its queue size to handle the fluctuations of the incoming rate of data; on the other hand, EQueue adopts batching, which can dramatically avoid cache misses when the data arriving rate is in an equilibrium state.
- A customized less-than compare-and-swap (LT-CAS) primitive is invented, which, in theory, can reduce the number of CAS failures by a factor of 256. As far as we know, this is the first implementation of customized LT-CAS primitive from literature.

For real applications, the data arriving rate may fluctuate sharply. As a result, a pragmatic lock-free algorithm must handle this robustness issue. This paper serves as an example. We released EQueue as an open-source project under GPL V3.[1]

The rest of this paper is organized as follows. Section II provides background. Section III presents the basic idea of EQueue. Sections IV, V and VI present the details of EQueue algorithm and its correctness proof. We present evaluations in Section VII, discuss related work in Section VIII, and conclude in Section IX.

## II. BACKGROUND

This section reviews *pipeline parallelism* and then highlights the importance of an efficient and robust FIFO queue which is used as the core-to-core communication mechanism in parallelized applications.

### A. PIPELINE PARALLELISM

To leverage the increasing number of CPU cores in multi-core processors, applications must be first parallelized. It is widely accepted that there are two basic techniques to parallelize applications: *task parallelism* and *data parallelism*. Task parallelism typically runs multiple tasks on different CPU cores, and these tasks run independently, such that there is no communication and synchronization between them. Task parallelism is the most basic form of parallelizing applications. However, it is limited by the availability of independent tasks without the need to communicate with each other. Data parallelism runs multiple instances of the same task on different CPU cores and processes independent data elements

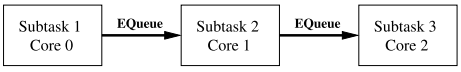
[1]The code is available at https://github.com/junchangwang/equeue.

in parallel. This technique can be used only if an application does not contain stateful processing routines, and if the data elements are fully independent.

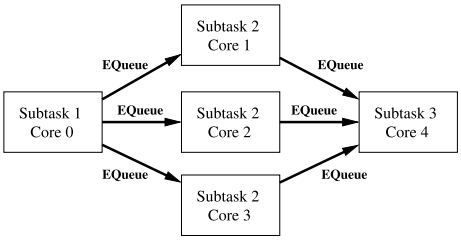
However, in practice, the conceptually simple and straightforward task- and data-parallel techniques cannot be applied to a large number of applications that have strict ordering requirements in their computation. As an example, consider how to parallelize the processing of a critical HTTP flow, which involves the workloads of receiving packets from hardware, running TCP/IP stack, (possibly) handling fragmented IP packets and out-of-order TCP frames, virus/attacks detection, and HTTP processing. Task and data parallelism cannot be used for the following reasons.

- There exists a total order in processing the packets of this HTTP flow. That is, we must process packets according to the order they arrive at the system. Otherwise, the out-of-order issue will arise, and the system performance will decrease.
- Processing the packets is stateful. For example, processing a TCP packet relies on the results of processing previous TCP packets.

As shown in Figure 1, pipeline parallelism parallelizes a single task by (1) dividing the task into a series of sequential subtasks, (2) mapping these subtasks to different CPU cores, and (3) providing a core-to-core communications scheme (e.g., EQueue) for the subtasks to transfer intermediate data elements. Note that the performance of an application utilizing the parallelism scheme is limited by the duration of the longest subtask. As an example, if the second subtask (*Subtask 1*) in Figure 1 takes longer time than other tasks to process a single data element, *Subtask 1* will become the performance bottleneck. To solve this performance issue, different parallelism techniques may be composed into more complex organizations. For example, Figure 1b shows a compounded architecture in which three instances of *Subtask 2* are created and mapped to three different CPU cores, allowing the system to perform three instances of *Subtask 2* in parallel.




(a) Parallelizing a single task by dividing the task into three sequential subtasks, and mapping them to three CPU cores.




(b) Parallelizing a single task by dividing the task into three sequential subtasks, creating three instances of the second subtask by leveraging task parallelism, and mapping all of the five instances of subtasks to five CPU cores.

**FIGURE 1.** Pipeline parallelism.

A good analogy of the benefits of the pipeline parallelism is the pipeline design in CPUs to increase processing throughput. We assume that we can divide the task of processing a packet equally, and hence each stage takes $T/n$ seconds, where $T$ denotes the total time to process a packet, and $n$ the number of stages. As a result, the parallelized application can process a packet in every $T/n$ seconds, resulting in a $n$ times speedup in throughput compared to the sequential version.

### B. CORE-TO-CORE COMMUNICATION

Figure 1b shows that the three basic parallelism schemes (*task parallelism*, *data parallelism*, and *pipeline parallelism*) are primitives and can be composed into more complex organizations for the purpose of performance. As the number of CPU cores increases, on the one hand, subtasks with higher overheads could be further parallelized, on the other hand, Amdahl's Law [16] shows that the communication overhead between CPU cores (the cost of an enqueue or dequeue operation of EQueue in Figure 1b) become the top limiting factor for parallelizing high-performance applications, necessitating the researches on fast and stable FIFO queues as the communication scheme between subtasks.

## III. BASIC ALGORITHM OF EQUEUE

EQueue is a practical realization of the following single-producer-single-consumer lock-free queue algorithm that is shown in Algorithms 1 and 2. The algorithm consists of an infinite array, *data*, and two indices, *head* and *tail*. Initially, each cell *data[i]* has a reserved value *ELEMENT_ZERO*, indicating that the cell is empty.

**Algorithm 1** Enqueue(Value) of Basic Algorithm

**Parameters**: *value*: data to be inserted

```
1 if(data[head] != ELEMENT_ZERO ){
2     return BUFFER_FULL;
3 }
4 data[head] = value;
5 head ++;
6 if(head ⩾ QUEUE_MAX_SIZE ){
7     head = 0;
8 }
9 return SUCCESS;
```

The pseudo-code of enqueue operation is shown in Algorithm 1. Specifically, an *enqueue(value)* operation first reads the value of *data[head]* and then checks if the queue is full by comparing the value to *ELEMENT_ZERO* (line 1). If the cell does not contain useful data, the enqueue thread inserts the value into the cell by (1) writing *value* to the queue, (2) incrementing index *head*, and (3) wrapping around if necessary (lines 4 – 8).

The pseudo-code of dequeue is shown in Algorithm 2. Specifically, a *dequeue()* operation first reads the value

**Algorithm 2** Dequeue() of Basic Algorithm

```
10 if(data[tail] == ELEMENT_ZERO ){
11     return BUFFER_EMPTY;
12 }
13 temp = data[tail] ;
14 data[tail] = ELEMENT_ZERO;
15 tail ++;
16 if(tail ⩾ QUEUE_MAX_SIZE){
17     tail = 0;
18 }
19 return temp ;
```

**Algorithm 3** Data Structure of EQueue

```
20 struct cas_info{
21     head: Indicator of enqueue (32-bits integer);
22     size: Size of the cyclic array (32-bits integer);
23 };
24 struct equeue{
25     info: instance of struct cas_info;
26     // padded to cache line size;
27     tail: Indicator of dequeue (32-bits integer);
28     // padded to cache line size;
29     traffic_full: Statistical counter of full cases;
30     // padded to cache line size;
31     traffic_empty: Statistical counter of empty cases;
32     // padded to cache line size;
33     data: pointer to cyclic array;
34 };
```

of *data[tail]* and then compares the value with *ELEMENT_ZERO* to check if the cell is empty (line 10). If yes, the *dequeue()* operation returns EMPTY. Otherwise, *dequeue()* (1) reads out the value of *data[tail]*, (2) writes value *ELEMENT_ZERO* to the cell to indicate that the data has been successfully extracted, and (3) increments index *tail* and wraps around if necessary (lines 13 – 18).

The basic algorithm is lock-free. However, one major performance drawback of the basic algorithm is that if the data incoming rate fluctuates, the queue will become full or empty frequently, preventing the basic queue algorithm from being used as the core-to-core communication mechanism for parallelized applications shown in Section II.

## IV. EQUEUE

The basic idea of EQueue is presented in Figure 2. On one hand, EQueue adaptively enlarges its queue size to handle the burstiness of data arriving rate, avoiding buffer overflow. On the other hand, EQueue adaptively shrinks its queue size when the data arriving rate is low. Shrinking its queue size is extremely useful when EQueue is used in real applications by reducing its memory footprint and increasing cache hit rate. The pseudo-code of EQueue is presented in Algorithms 3, 4, and 5.

### A. DATA STRUCTURE AND GLOBAL VARIABLES

Algorithm 3 shows the key data structures of EQueue. For each queue, an instance of *enqueue* is created. *data* is a pointer indicating the pre-allocated cyclic array of the queue, and *traffic_full* and *traffic_empty* are signed integers to count the number of full and empty states of the queue, respectively. Each time an enqueue operation fails, *traffic_full* is incremented by one. Similarly, *traffic_empty* is incremented by one each time a dequeue operation fails. As a result, the difference between *traffic_full* and *traffic_empty* reflects the fluctuation of the data arriving rate, and hence can be used by EQueue to decide if it is necessary to shrink or enlarge the queue size. It is worth noting that we chose this "two-indices" approach by utilizing the *traffic_full* and *traffic_empty* indices for ease of presentation. However, there are other more sophisticated approaches to capturing the statistics and characteristics of the data arriving rate, which are out the scope of this paper.

In initialization, EQueue (1) pre-allocates an array of DEFAULT_QUEUE_SIZE entries, where DEFAULT_QUEUE_SIZE is by default set to 1024*128, and (2) selects the first portion of the pre-allocated array as the cyclic array of the queue (the size of the portion used is determined by variable *size*). Similar to the basic algorithm, the two variables *head* and *tail* points to the first and the last cells containing useful values. It is worth noting that to shrink the queue size, variables *head* and *size* must be updated

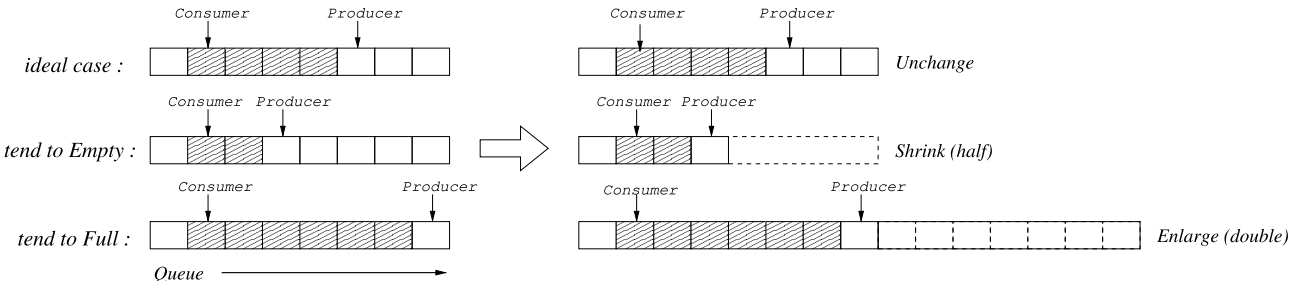



**FIGURE 2.** Basic idea of EQueue: shrinking or enlarging the cyclic array adaptively, if necessary, to reduce the memory footprint or to handle the burstiness of data arriving, respectively.

**Algorithm 4** Enqueue(Value) of EQueue

```
    Parameters: value: data to be inserted
35  // local variable
36  temp: local variable to buffer value of info.head
    /* Check if the queue is full.      */
37  if(data[info.head] != ELEMENT_ZERO){
38      traffic_full ++;
39      return BUFFER_FULL;
40  }
41  temp = info.head;
42  info.head ++;
43  if(info.head ⩾ info.size){
        /* Check if it is necessary to
           enlarge the queue size.       */
44      if( (traffic_full - traffic_empty) ⩾
        ENLARGE_THRESHOLD){
45          info.size = info.size * 2;
46          traffic_full = traffic_empty = 0;
47      }
48      else{
49          info.head = 0 ;
50      }
51  }
52  data[temp] = value;
53  return SUCCESS;
```

**Algorithm 5** Dequeue() of EQueue

```
54  // local variable
55  temp: local variable to buffer value of tail
56  temp_value: local variable to buffer value of data
    pointed by tail
57  info_tmp, info_tmp_new: local instances of structure
    info
58  if(data[tail] == ELEMENT_ZERO){
59      traffic_empty ++;
60      return BUFFER_EMPTY;
61  }
62  temp = tail;
63  tail ++;
64  tag1;
65  if(tail ⩾ info.size){
66      if((traffic_empty - traffic_full) ⩾
        SHRINK_THRESHOLD){
67          info_tmp = info_tmp_new = info;
            // Atomic read
68          if(info_tmp.head ⩽ info_tmp.size/2){
69              info_tmp_new.size = info_tmp_new.size
                / 2;
70              if(CAS(&info, info_tmp, info_tmp_new)){
71                  traffic_empty = traffic_full = 0;
72              }
73              else{
74                  goto tag1;
75              }
76          }
77      }
78      tail = 0;
79  }
80  temp_value = data[temp];
81  data[temp] = ELEMENT_ZERO;
82  return temp_value;
```

atomically (discussed in the following paragraphs), such that the two variables are packed into the data structure, *cas_info*. Structure *cas_info* is 64-bits long, which can be updated atomically, if aligned, on widely-used 64-bits servers.

### B. ENQUEUE OF EQUEUE

The *enqueue* operation of EQueue (shown in Algorithm 4) is based on Algorithm 1, such that we only discuss the differences between these two algorithms. The *enqueue* operation first checks if the queue is full. If yes, the *enqueue* operation increments the counter *traffic_full* and returns *BUFFER_FULL* (line 37 – lines 39). Otherwise, *enqueue* stores the value of *head* temporarily in a local variable *temp* (line 41) and increments *head* by one. If *head* is equal to *size* (line 43), which means that the *head* is pointing to the last cell of the cyclic array, the *enqueue* operation attempts to enlarge the queue size. For ease of presentation, we use the "two-indices" policy, by comparing the difference between variables *traffic_full* and *traffic_empty* to the pre-defined threshold value *ENLARGE_THRESHOLD* (line 44). The *enqueue* operation enlarges the queue size by doubling the value of variable *size* and then resets *traffic* (lines 45 – 46).

### C. ENLARGE QUEUE SIZE

We recall that EQueue uses the first portion of the pre-allocated array as its cyclic array, and the shared global variable *size* indicates the size of the cyclic array. The enqueue and dequeue operations read out the value of variable *size* each time they want to check if they have reached the last cell of the cyclic array that EQueue is using. Therefore, the basic idea of enlarging the queue is that if the value of *size* is doubled, the size of the cyclic array which the enqueue and dequeue operations can use is doubled accordingly. In practice, however, careful design is required. Figure 3 presents the

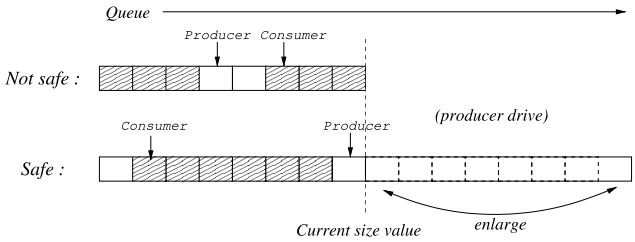



**FIGURE 3.** Safe and unsafe cases of enlarging the size of EQueue. In the *Safe* case, *producer* enlarges the queue by doubling the value of *size*.

two scenarios in which an enlarging operation can take place: (1) The producer wraps around and is behind the consumer (denoted as *Not Safe* in Figure 3). In this case, it is not safe to enlarge the queue size. Otherwise, the consumer may read out the updated value of *size* before wrapping around, and then try to read slots that have not been inserted data into, incurring errors. (2) The producer is in front of the consumer (denoted as *Safe* in Figure 3). In this case, when the producer reaches the last cell of the current cyclic array, it is safe for the producer to enlarge the queue by doubling the value of *size*. It is the truth that in a queue the consumer never crosses the producer prevents the consumer from touching the new allocated slots before the producer has inserted data into them. And as a result, doubling the queue size is safe in this case. The implementation of the case *Safe* is shown in Algorithm 4 (lines 43 – 51).

### D. DEQUEUE OF EQUEUE

The *dequeue* operation (shown in Algorithm 5) is based on Algorithm 2. The *dequeue* operation first checks if the queue is empty (line 58). If yes, the dequeue operation records the *EMPTY* state of the queue (line 59) and then returns. Otherwise, *dequeue* saves the value of *tail* temporarily in a local variable *temp* (line 62) and increments *tail* by one (line 63). If *tail* is equal to *size* (line 65), which means that *tail* is pointing to the last cell of the sub-array which EQueue is using as the cyclic array, *dequeue* checks if it is necessary to shrink the queue size by comparing the difference between variables *traffic_empty* and *traffic_full* to the predefined threshold value *SHRINK_THRESHOLD* (line 66). *SHRINK_THRESHOLD* is by default set to 128. The *dequeue* operation shrinks the queue size by dividing the value of variable *size* by 2 (line 69) and then updates the global shared variable *size* to force both *enqueue* and *dequeue* to use the reduced cyclic array (line 70). Reducing the size of the queue, however, contains some corner cases which must be handled carefully. We solve these corner cases in the following paragraph.

### E. SHRINK QUEUE SIZE

Figure 4 presents the two scenarios in which shrinking the queue size can take place: (1) The producer is in front of the consumer (denoted as case *Not safe* in Figure 4). In this case, it is not safe to shrink the queue size. Otherwise, the producer may insert data into the bottom half of the sub-array which EQueue will not use. What we learn from the *Not safe* case is that if EQueue wants to shrink the queue size, it must make sure that neither the *producer* nor the *consumer* is accessing the bottom half of the queue. Fortunately, this can be guaranteed in the case *Safe* where the *producer* has wrapped around and is behind the *consumer* thread. In this case, when the *consumer* reaches the last cell of the current cyclic array, it is safe for the consumer to shrink the size of the sub-array by dividing the value of *size* by two, *if the producer is in the first half of the queue*. Note that the operation of reducing the value of *size* and the operation of checking if the *producer* is in the first half of the queue must take place *atomically*. To achieve that, we pack fields *size* and *head* into a 64-bits long structure (*cas_info* shown in Algorithm 3), which can be manipulated atomically via one *CAS* instruction. We thus make the realistic assumption that the maximum queue size of EQueue does not exceed $2^{32}$. Another corner case is that when the *consumer* is trying to shrink the queue size, the *producer* is moving forward in the first half of the queue, which may incur CAS failures and force *consumer* to retry the shrinking process. To avoid this potential live-lock issue, we invented a novel Less-Than Compare-And-Swap (LT-CAS) primitive to dramatically reduce the number of CAS failures. Optimizations are discussed in Section V-B.

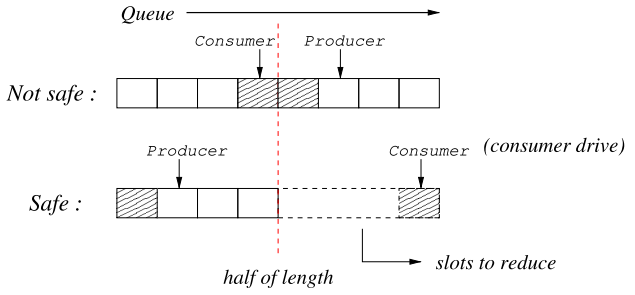



**FIGURE 4.** Safe and unsafe cases of shrinking the size of EQueue. In the *Safe* case, *consumer* reduces the queue by dividing the value of *size* by two.

The code to shrink the queue size is shown in Algorithm 5 from line 64 to 79. *Dequeue* first reads out the value of *info*, which includes fields *head* and *size*, in a single operation (line 67). Then, *dequeue* checks if the *producer* is in the first half of the queue by checking if the value of *head* is less than one half of the queue size (line 68). If the answer is yes, the *enqueue* operation prepares a new instance of *info* and reduces the value of *size* by half (line 69). Otherwise, it is not safe to shrink the queue size and hence *dequeue* returns.

If it is safe for *enqueue* to shrink the queue size, it tries to set the new value of *size* by leveraging one *CAS* instruction (line 70), which guarantees to perform the following two operations atomically: (1) Make sure that the *producer* did not move to the bottom half of the queue which we are going to discard, and (2) to cut the value of *size* by half. If the *CAS* instruction succeeds, the *dequeue* operation resets two traffic variables and returns (line 71). Optimizations to this basic algorithm are discussed in Section V-B.

## V. OPTIMIZATIONS FOR EQUEUE

This section details optimizations for EQueue, highlighting the utilizing of batching and a novel less-than compare-and-swap primitive.

### A. BATCHING

Careful readers may have noticed that EQueue has performance bottlenecks. One of them is that for each enqueue and dequeue operation, to detect if the queue is *FULL* or *EMPTY*, both the producer and the consumer need to read the shared

cyclic data array *data* (line 37 in Algorithm 4 and line 58 in Algorithm 5). Accessing shared global data, however, can incur performance deterioration because of *cache thrashing* and underneath cache traffic between multiple CPU cores [17].

To handle this performance issue, EQueue leverages a batching mechanism. The basic idea behind batching is to utilize prediction; each time when EQueue checks if the queue is *FULL* or *EMPTY*, it checks a group of cells, instead of a single cell. If a prediction succeeds, which implies that the cells in this group can be used, and as a result, it is not necessary for EQueue to check each cell before using it. If a prediction fails, EQueue can either give up batching or wait until a group of cells become available [11]. Inspired by BQueue [5], EQueue adopts a backtracking mechanism. With backtracking, when a prediction fails, EQueue can adaptively reduce its prediction distance, which can dramatically increase the algorithm's robustness.

Applying batching to EQueue, however, is non-trivial for the following reasons.

- Batching and enlarging/shrinking are designed for different scenarios. Specifically, batching can increase the throughput of EQueue when the traffic is in an equilibrium state, while enlarging/shrinking is expected to take effect when there are bursts of income data. How to switch between these two techniques is an interesting and hard problem. For brevity and clarity, we presented a "two-indices" solution in the paper. However, there are other more sophisticated and valuable approaches to capturing the statistics and characteristics of the data arriving rate, which is left as future work.
- Batching and enlarging/shrinking may take place simultaneously, such that EQueue must handle the possible conflicts between these two techniques. For example, when a producer is detecting if a group of cells is available (Line 90 in Algorithm 7), the consumer may be reducing the queue size. Without careful design, these two operations may conflict, and as a result, the producer may later enqueue data into the cells that will never be dequeued.

All these problems arise in integrating batching into EQueue, and they have not been well studied, let alone solved before. We solved these problems in this paper.

The basic idea of batching in the producer of EQueue is presented in Figure 5. In this figure, both the producer and consumer move from left to right, and the producer has wrapped around and is behind the consumer. Shaded slots indicate that the slots contain data; Otherwise, the slots are empty and are ready to be used by the producer. Each time the producer wants to detect available slots (i.e., slots that are empty and ready to be written data into), it first checks if the slot that is DEFAULT_BATCH_SIZE further away is empty. If that slot is empty, it is safe for the producer to insert data into the following DEFAULT_BATCH_SIZE slots, given the queue is a cyclic array and EQueue is fundamentally a single-producer-single-consumer FIFO queue. If the slot that is

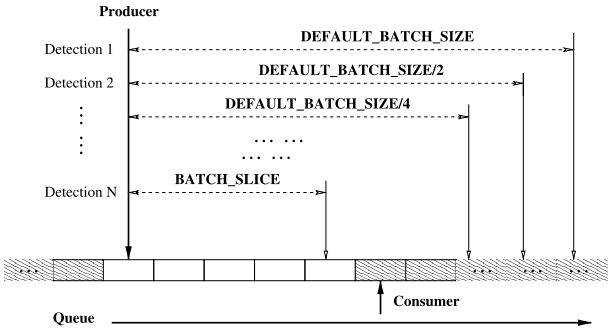



**FIGURE 5.** Batching in producer. The batching mechanism can dynamically adjust its distance, which makes the algorithm more robust.

DEFAULT_BATCH_SIZE further away is not empty, which means that the slot contains data that has not been extracted by the consumer yet, instead of waiting, EQueue adaptively reduces its batching size to *DEFAULT_BATCH_SIZE*/2. EQueue repeats the detection until there is a group of available slots, or the minimum batching size has been reached (BATCH_SLICE by default).

The pseudo-code of enqueue operation with batching is presented in Algorithm 6, 7, and 8. To support batching, a local variable *batch_head* is added into structure *equeue* as producer's local variable (Algorithm 6). Differences between enqueue operation with batching (Algorithm 7) and the basic version (Algorithm 4) are highlighted by using bold fonts. Specifically, to insert a value into the queue, EQueue first checks if there are any available slots, which are guaranteed to be available (line 90 of Algorithm 7). If there are any available slots, EQueue continues without touching the global shared cyclic array *data*, dramatically reducing cache misses. Otherwise, EQueue tries to detect a group of available slots by using help function *enqueue_detech_batching_size()*, which is presented in Algorithm 8 and discussed shortly. After that, Algorithm 7 inserts data into the queue, which is the same as in Algorithm 4.

**Algorithm 6** Amended Data Structure of EQueue

```
83 struct equeue{
84     batch_head: Producer's local variable;
85     ...
       /* Same as the definition of
          struct equeue in algorithm 3.
          */
86     ...
87 };
```

Algorithm 8 presents the helper function enqueue_detect_batching_size() which detects the available slots in batch. The algorithm first checks if the following DEFAULT_BATCH_SIZE slots are available by (1) adding DEFAULT_BATCH_SIZE to index *info.head* (line 109), and then (2) detecting if the farthest slot is empty (line 110). The value of DEFAULT_BATCH_SIZE is set by default to

**Algorithm 7** Enqueue(Value) With Batching. Differences to Algorithm 4 Are Highlighted by Using Bold Font

```
   Parameters: value: data to be inserted into EQueue
88 // local variable
89 temp: local variable to buffer value of batch_head
   /* Check if the queue is full.     */
90 if(info.head == batch_head){
       /* batch_head will be updated by
          function
          enqueue_detect_batching_size()
          if it returns SUCCESS.       */
91     if( enqueue_detect_batching_size() !=
       SUCCESS){
92         return BUFFER_FULL;
93     }
94 }
95 temp = info.head;
96 info.head ++;
97 if( info.head ⩾ info.size){
       /* Check if it is necessary to
          enlarge the queue size.      */
98     if( (traffic_full - traffic_empty) ⩾
       ENLARGE_THRESHOLD){
99         info.size = info.size * 2;
100        traffic_full = traffic_empty = 0;
101    }
102    else{
103        info.head = 0;
104    }
105    data[temp] = value;
106    return SUCCESS;
107 }
```

**Algorithm 8** Enqueue_detect_batching_size()

```
   Variables : data: (global cyclic array)
               head, size: (local integers)
               queue: (structure of EQueue)
108 size = DEFAULT_BATCH_SIZE;
109 head = MOD(queue.info.head + size);
110 while(data[head] && size > BATCH_SLICE){
111     wait_ticks(DEFAULT_PENALTY);
112     size = size / 2;
113     head = MOD(queue.info.head + size);
114 }
115 if(batch_size > BATCH_SLICE){
116     queue.batch_head = head;
117     return SUCCESS;
118 }
119 else{
120     return BUFFER_FULL;
121 }
```

one-quarter of the queue size, and *MOD* is a helper function that adds *batch_size* and *head* together and wraps around if necessary. If the farthest slot is unfortunately not empty, which means the slot contains data that has not been retrieved yet, the producer will (1) wait for a while (1,000 CPU cycles by default) to allow the consumer to make progress, and then (2) reduce the detection distance by dividing the value of *batch_size* by two (line 112). Algorithm 8 repeats the detection until a group of empty slots is found, or the minimum batch size is reached (line 110). If a group of empty slots is found, the algorithm updates *batch_head*. Otherwise, the queue is full and *BUFFER_FULL* is returned.

In summary, by leveraging the batching mechanism, EQueue can not only efficiently handle the burstiness of incoming data, but also increase its throughput. We omitted the implementation details of the batching mechanism for the consumer in this paper because it is similar to the mechanism for the producer and it is straightforward for readers to derive it.

### B. LESS-THAN COMPARE-AND-SWAP

Another performance issue in EQueue is the work wasted due to the CAS failures in shrinking the queue size (line 70 in Algorithm 5). Specifically, to shrink the queue size, the consumer needs to (1) make sure that the *producer* is in the first half of the queue, and (2) shrink the queue size. The challenge is that these two operations must be performed atomically. To achieve that, EQueue packs the two variables, *head* and *size*, into a 64-bits long structure, *cas_info* (Algorithm 3), and the *consumer* shrinks the queue size by performing one CAS operation on *cas_info* (line 70 in Algorithm 5). As a result, the CAS operation will fail, even if the *producer* merely moves forward a single slot while the *consumer* is performing the CAS loop. Unfortunately, in high-performance applications, the *producer* runs as fast as the *consumer* does, and as a result, most CAS operations may fail, forcing the *consumer* to retry. The time wasted due to the CAS failures can dramatically decrease the performance of EQueue.

Theoretically, if the *producer* is still in the first half of the queue, the operation of shrinking the queue size (dividing the value of variable *size* by 2) should be allowed to proceed. In other words, it is not necessary for the *producer* to stay still while the *consumer* is performing a CAS operation. For example, suppose that the queue size was larger than 4 and the *producer* pointed to the first slot of the queue, the *consumer* should be able to successfully shrink the queue size even if the *producer* has moved forward and is currently pointing to the second slot of the queue. To that end, what EQueue needs is a less-than compare-and-swap (LT-CAS) primitive, which, unfortunately, does not exist in commercial hardware.

Rather than giving up the idea of LT-CAS, which can help EQueue dramatically reduce the number of CAS failures and hence boost performance, we present a customized LT-CAS primitive which is built on top of normal CAS primitive.

The customized LT-CAS is efficient. Theoretical analysis and experimental results show that it can reduce the number of CAS failures in EQueue by a factor of 256.

Figure 6 presents the basic idea of LT-CAS. The variable *head* is a 16-bits long integer. We thus make the realistic assumption that the maximum queue size of EQueue in practice does not exceed $2^{16}$. The queue is divided into slices, each of which consists of 256 slots. For example, the index of the first slot in Slice 1 is $0 \times 0000$, and the index of the first slot in Slice 2 is $0 \times 0100$. The first bytes of the 256 slots in a single slice have the same value. The basic idea behind LT-CAS is that a 16-bits long integer is used as the producer index *head*, as a normal FIFO queue does. To check if the *producer* has moved when the *consumer* is performing a CAS primitive, EQueue uses the slice number (the most significant byte of the variable *head*). In that case, if the old value and new value of *head* are within the same slice, the CAS primitive will succeed. For example, as shown in Figure 6, the *producer* pointed to slot $0 \times 0202$ (*old value*). While the *consumer* is performing a CAS operation, the *producer* has successfully inserted a new value and moved to slot $0 \times 02FE$ (*new value*). Since the LT-CAS primitive only compares the most significant byte of the *old value* and *new value*, the LT-CAS primitive will succeed. In contrast, if a normal CAS primitive is used, the *consumer* will fail in shrinking the queue size and has to retry.

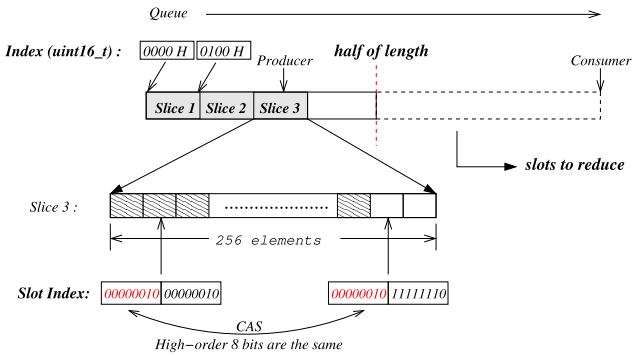



**FIGURE 6. Basic idea of the Less-Than Compare-And-Swap primitive. A 16-bits integer is used as index, and its most significant byte is used as the parameter of CAS.**

LT-CAS is atomic because it is essentially a CAS operation performed on the most significant byte of the variable *head*. The CAS primitive on most modern computer systems can be used with any integer/pointer type that is 1, 2, 4, or 8 bytes in length. For example, the CAS instruction provided by X86 processors (specifically, those later than Intel486) can compare the value in the AL (1 byte), AX (2 bytes), EAX (4 bytes), or RAX (8 bytes) register with the destination operand [18], and typically the CAS instruction is used with a LOCK prefix to allow the instruction to be executed atomically. For high-level programming languages (e.g., C/C++), the latest built-in function such as __atomic_compare_exchange [19] can be used with any integral scalar, such that applying it on the most significant byte of the variable head is atomic.

By utilizing the LT-CAS, theoretically, EQueue can reduce the number of CAS failures by a factor of 256. Experimentally we found that this customized LT-CAS can dramatically reduce the number of CAS failures, which in turn, boost the system performance (detailed in Section VII). Even though the LT-CAS primitive is customized for EQueue, we believe it is helpful to other use cases.

It is worth noting that the correct implementation of the LT-CAS primitive depends on the specific endianness of the underneath hardware running the code. In this paper, an x86-64 server from Intel is used and hence the sample code is for little-endian machines. For big-endian machines, the slice number is stored in the least significant byte of the variable *head*, and as a result, the code should be modified accordingly. A portable solution can be implemented by using conditional directive (such as #if) to instruct the preprocessor of the compiler to choose the appropriate byte as the slice variable.

## VI. CORRECTNESS

We design EQueue for an asynchronous shared memory system [20]. To simplify the presentation of the pseudocode, we assume a *sequentially consistent* memory model.

### A. LINEARIZABILITY

The EQueue algorithm is linearizable [20] because both the enqueue and dequeue operations have specific "linearization point." Line 52 of Algorithm 4 is the linearization point of the enqueue operation. With this single step, the effect of the enqueue operation becomes visible to the consumer. This linearization point takes effect when the new value has been successfully written into the queue. If the queue is full, the enqueue() operation has a linearization point where it returns a failure (Line 39). Similarly, lines 60 and 81 in Algorithm 5 are the linearization points of the dequeue operation. The linearization point of the procedure of shrinking queue size is at the *CAS* instruction.

### B. LIVENESS

The enqueue operation of EQueue is wait-free. The dequeue is lock-free due to the *CAS* instruction and may have livelock issue if enqueue keeps making progress and dequeue has to retry. However, it is worth noting that EQueue sets a threshold and the dequeue operation can escape and return if the times of failure exceed the pre-defined *LOOP_THRESHOLD* (detailed in Algorithm 5). So in practice, there is no livelock problem in EQueue.

## VII. EVALUATION

In this section, we show that EQueue outperforms existing best solutions (FastForward [10], B-Queue [5], and MCRingBuffer [11]) in performance. Specifically, we first show the experiment setup. In Section VII-B, we evaluate the queues on a simulated testbed where data incoming rate

fluctuates. In Section VII-C, we evaluate the performance of the queues in a parallelized high-performance network application. Experimental results show that EQueue can serve as a stable and efficient communication scheme for parallelizing high-performance applications where data incoming rate fluctuates.

### A. EXPERIMENT SETUP

We run the experiments on a 16 core Dell R730 server, with two Intel Xeon E5-2609 processors, each having 8 CPU cores running at 1.66GHz. Each CPU core has a 64KB L1 cache and a 256KB L2 cache. All of the 8 cores on the same die share 20MB L3 cache. The server uses 128GB DDR4 memory. The software includes Linux kernel version 4.4.0 and GCC version 5.4.0 with -O3 option.

For each test, the producer transfers one trillion data elements to the consumer thread. Each thread runs on a dedicated CPU core. We measure the cost of each enqueue/dequeue operation by leveraging *Time Stamp Counter* [21]. To collect other interesting architectural statistics, we utilize Linux Perf [22]. For each experiment, we run the program by 30 times and report the mean value.

### B. PERFORMANCE EVALUATION ON SIMULATED TESTBED

To demonstrate the robustness of EQueue in handling the burstiness of incoming data, we measure the performance of the four queues on a simulated testbed where the data incoming rate fluctuates. On this testbed, in each iteration, the consumer retrieves an element from the queue and then waits for 85 CPU cycles to simulate processing one packet in a 10Gbps link in line rate $((1.66*10^9)*(64*8)/(10*10^9) = 85$ CPU cycles). To simulate the burstiness of incoming data in real applications, the producer thread does the following. The producer thread performs a while-loop and in each iteration: (1) the producer thread sleeps for a short period (the time budget for processing a single data multiplied by the burst size) to simulate the peace period in network devices due to batching on hardware and operating systems, and then (2) the producer inserts a group of data of burst size into the queue in a batch, to simulate the burstiness of incoming data in a network application.

Figure 7 shows the average CPU cycles per operation. When the burst size is small (e.g., less than 2,048), different lock-free queues perform similarly; it takes a queue about 50 CPU cycles to perform an enqueue/dequeue operation. However, as the burst size increases, the performance of FastForward, MCRingBuffer, and BQueue decrease dramatically. The major reason is that because of the burstiness of the incoming data, these lock-free queues become full or empty frequently. In contrast, EQueue is insensitive to the fluctuation of the incoming data, because it can adjust its queue size adaptively.

To explore why EQueue performs better, we set the burst size in each iteration as 4,096 and collect the characteristics of different queues. Table 1 presents the performance number of different queues. Columns *Same* and *Cross* list CPU cycles a queue takes to perform an enqueue/dequeue operation, when the enqueue and dequeue threads are placed on the same CPU die or on different dies, respectively. The smaller the cycle number is, the better the performance of a queue. Table 1 shows that EQueue outperforms the other three queues, no matter the placement configuration of the enqueue thread and the dequeue thread. The reason that EQueue outperforms other queues is that it can adaptively adjust its queue size. Columns *Len. (Min)* and *Len. (Max)* denotes the minimum and maximum length of the queue during the experiments. We only list the number of EQueue because other queues cannot change their queue size on-the-fly. Experiments show that EQueue can not only enlarge its queue size up to 65,536 to handle the burstiness of incoming data, but also shrink its queue size to 256 when the data incoming rate is low. By shrinking the queue size, EQueue can dramatically reduce cache miss rate and as a result, improve the overall performance. Column *LLC Miss Rate* shows the Last Level cache miss rate reported by Linux Perf. Experimental results show that EQueue has the lowest LLC miss rate (0.07%) because it can enlarge and reduce its queue size adaptively. The cache behavior shown in this table explains why EQueue outperforms other queues.

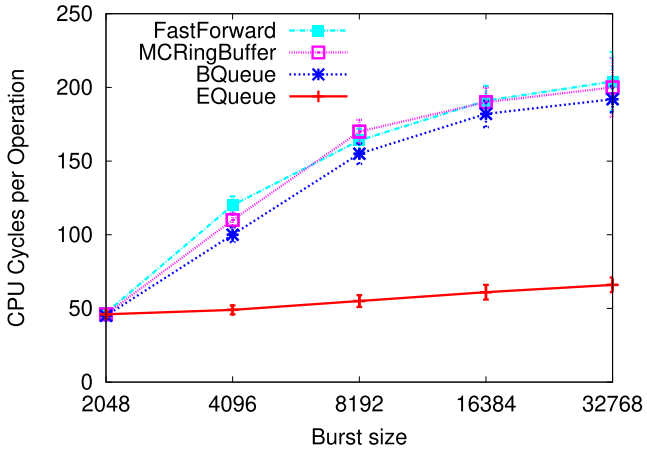



**FIGURE 7.** Performance of lock-free queues with different burst sizes.

**TABLE 1.** Performance of different queues.

| | Same (cycle) | Cross (cycle) | Len. (Min) | Len. (Max) | LLC Miss Rate |
|---|---|---|---|---|---|
| FastForward | 113 | 120 | – | – | 0.54% |
| MCRingBuffer | 101 | 110 | – | – | 0.23% |
| B-Queue | 109 | 115 | – | – | 0.20% |
| EQueue | 35 | 49 | 256 | 65,536 | 0.07% |

### C. APPLYING QUEUES TO REAL NETWORK APPLICATION

To measure the performance of different queues on real applications, we parallelize a high-performance network application with pipeline parallelism and then leverage the queues as the communication scheme between different stages. Specifically, we parallelize a Network Intrusion

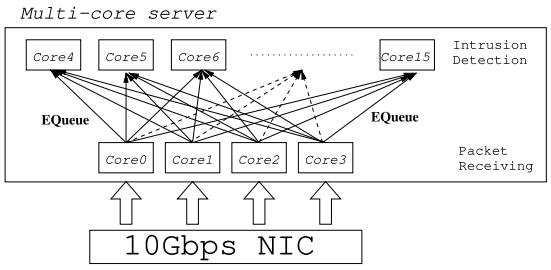



**FIGURE 8.** A parallelized network application leveraging queues.

Detection System (NIDS) that involves a TCP/IP stack from Libnids [23], a port-independent protocol identifier, and an HTTP parser for analyzing HTTP traffic. Figure 8 illustrates the pipelined organization of the system where pipeline parallelism is used to exploit multi-core processors. There are two pipeline stages in the system:

- **Packet Receiving** Four CPU cores (*Core0* to *Core3*) work as the first stage and receive packets from a 10Gbps network interface cards (NIC) through an optimized Linux NIC driver [24]. Upon receiving a packet, the CPU core steers the packet to one of the CPU core in the second stage (*Core4* to *Core15*), via a lock-free FIFO queue.
- **Intrusion Detection** Twelve CPU cores (*Core4* to *Core15*) receive packets by polling the queues, which is used to connect the first and second stages. Then the AP stage performs a complete Layer 2 to Layer 7 network processing by using the run-to-completion model.

Each of the four *Packet Receiving* cores can send packets to any of the twelve *Intrusion Detection* cores via 12 queues. In the experiments, to measure the performance of queues on real high-performance applications, we vary the number of concurrent queues by indicating each *Packet Receiving* core to connect to a different number of *Intrusion Detection* cores. For example, we first indicate each *Packet Receiving* core to send packets to two *Intrusion Detection* cores, and then to four *Intrusion Detection* cores, and so on. The maximum total number of lock-free queues in the system is 48.

Figure 9 measures the performance of different queues as a function of the number of queues in the system. *EQueue-LT* denotes the EQueue implementation with *LT-CAS* primitive, and *EQueue-LT-Batching* denotes the EQueue implementation with both the *LT-CAS* and *batching* technique discussed in Section V. Figure 9 shows that when the number of queues in the system is small (e.g., 8 queues), it takes FastForward about 75 CPU cycles to perform an enqueue/dequeue operation, and it takes the other three queues about 40 cycles to perform an enqueue/dequeue operation. However, as the number of queues increases (e.g., 48 queues), the performance of FastForward, MCRingBuffer, and B-Queue deteriorates sharply. In contrast, EQueue is more robust. The major reason is that in a massively-parallel application, EQueue can adaptively shrink its queue size (and hence reduce its memory footprint). As the number of queues in the system keeps increasing, the *LT-CAS* and batching techniques

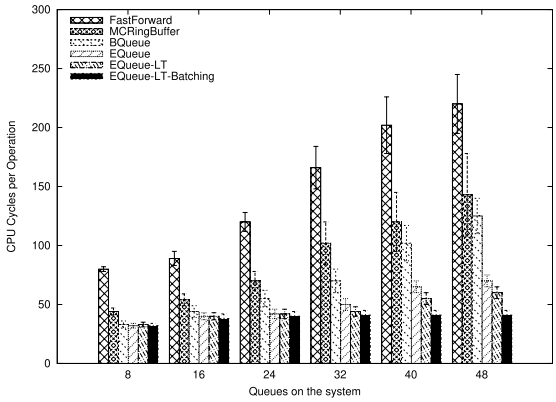



**FIGURE 9.** Queue performance in real Intrusion Detection system.

result in further performance improvements. For example, in the system with 48 concurrent queues, it takes EQueue 70 CPU cycles to perform an enqueue/dequeue operation. In contrast, the EQueue implementation leveraging the *LT-CAS* and batching techniques can perform an enqueue/dequeue operation in 41 CPU cycles, resulting in a 41% performance speedup.

## VIII. RELATED WORK

A variety of studies have focused on providing efficient core-to-core communication, by utilizing concurrent lock-free queues [25]–[30]. It is widely accepted that [25] is the first practical lock-free multi-producer-multi-consumer queue implementation, in which a dequeue thread may help a delayed enqueue thread to make progress. Most of these implementations are multiple-producer and/or multiple-consumer queues, and hence, may suffer the following two issues. The first is that they rely on dynamic memory allocation, which, unfortunately, may incur severe performance degradation. Besides, ABA problem [31] may arise in the case where a memory location is dequeued but then is reused shortly for one subsequent enqueue operation. In that case, for some other threads, the memory location could appear being untouched. To solve the ABA problem, one common solution is to add a new field for each node, and this field is updated each time the node is enqueued or dequeued. By utilizing this field, the system can distinguish dequeued-and-then-enqueued nodes.

To provide a high-speed core-to-core communication mechanism, researchers typically utilize single-producer-single-consumer FIFO queues (e.g. [4]–[7], [10]–[13], [32], [33]). The major benefit of these implementations is that it is not necessary for the queues to handle the aforementioned ABA and dynamic memory allocation problems. Therefore, they could be very fast compared to the multi-producer-multi-consumer implementations. However, lock-free queues have one potential performance issue; when the queue is becoming full or empty, cache thrashing happens. In that case, the enqueue thread and the dequeue thread are updating memory locations that are too close and reside in the same

cacheline. To solve this issue, the technique of batching is heavily used in these implementations [4], [10], [11] to utilize underlying CPU caches and to boost the throughput of the queue. The basic idea is that each time the enqueue thread inserts data into the queue, instead of writing a single data element into the queue, it inserts a group of data elements. Similarly, the consumer thread dequeues a group of data each time it reads data out of the queue. Batching naturally introduces a distance between the enqueue thread and the dequeue thread, and hence avoids cache thrashing.

One potential issue in batching-based lock-free queues are that they could block when the enqueue thread is waiting for more data to be inserted into the queue, and the dequeue thread is waiting for more data to read a batch of data out of the queue. To solve this problem, we presented B-Queue [5], an efficient single-producer-single-consumer ring buffer. B-Queue addresses the potential deadlock issue by exploiting a technique called *back-tracking*. The goal of EQueue, however, is to address the performance issue when lock-free queues are used in real applications where data arriving rate fluctuates.

## IX. CONCLUSION AND FUTURE WORK

This paper explores EQueue, an efficient and robust lock-free FIFO queue for parallelizing applications by utilizing pipeline parallelism strategy, on multi-core architectures. EQueue can shrink its queue size when data arriving rate is low to keep its memory footprint small. Besides, EQueue can enlarge its queue size when data arriving rate is in burstiness to avoid overflow. Optimizations such as batching and a customized less-than compare-and-swap (LT-CAS) primitive have been adopted to improve EQueue's performance and robustness, which makes EQueue a good candidate for building high-performance applications on multi-core processors. Experimental results show that EQueue can be optimized further by, for example, working with a real-time system scheduler. We will continue to optimize the algorithm.

## ACKNOWLEDGMENT

This article was presented at the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'17). Since then, the algorithm has been used in multiple open-source projects. In this journal version, we extend the algorithmic design of EQueue according to the users' feedback and conduct more experimental evaluation.

## REFERENCES

[1] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: A dynamic dependence-based parallel execution model," in *Proc. 14th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2009, pp. 85–96.

[2] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and exploiting concurrency in networked applications in aspen," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2007, pp. 13–23.

[3] G. D. Price, J. Giacomoni, and M. Vachharajani, "Visualizing potential parallelism in sequential programs," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 82–90.

[4] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures," in *Proc. 23rd Int. Conf. Super Comput.*, 2009, pp. 204–213.

[5] 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, pp. 137–159, Feb. 2013.

[6] K. Mitropoulou, V. Porpodas, X. Zhang, and T. M. Jones, "Lynx: Using os and hardware support for fast fine-grained inter-core communication," in *Proc. Int. Conf. Supercomputing*, 2016, p. 18.

[7] S. Arnautov, P. Felber, C. Fetzer, and B. Trach, "Ffq: A fast single-producer/multiple-consumer concurrent fifo queue," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*. Dec. 2017, pp. 907–916.

[8] 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*, 2007, pp. 27–36.

[9] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 162–173.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," in *Proc. 13th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2008, pp. 43–52.

[11] P. Lee, T. Bu, and G. Chandranmenon, "A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2010, pp. 1–12.

[12] T. Preud'homme, J. Sopena, G. Thomas, and B. Folliot, "Batchqueue: Fast and memory-thrifty core to core communication," in *Proc. 22nd Int. Symp. Comput. Archit. High Perform. Comput.*, 2010, pp. 215–222.

[13] T. B. Jablin, Y. Zhang, and J. A. Jablin, "Liberty queues for epic architectures," in *Proc. 8th Workshop Explicitly Parallel Instruct. Comput. Archit. Compiler Technol.*, 2010, pp. 1–5.

[14] L. Kleinrock and R. Gail, *Queueing Systems: Problems Solutions*. Hoboken, NJ, USA: Wiley, 1996.

[15] M. Alizadeh, A. Kabbani, T. Eds.,all, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proc. 9th USENIX Conf. Networked Syst. Design Implement.*, 2012, p. 19.

[16] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf.*, New York, NY, USA, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[18] Intel. (2019). *Intel Software Developer's Manual (Vol. 2a 3-189)*. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[19] GCC. *Gcc Built-In Functions for Memory Model Aware Atomic Operations*. Accessed: Dec. 20, 2019. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

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

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Mountain View, CA, USA, Dec. 2009.

[22] L. P. Team. (2019). *Linux Perf Tool*. [Online]. Available: https://perf.wiki.kernel.org

[23] R. Wojtczuk. (2019). *Libnids*. [Online]. Available: http://libnids.sourceforge.net/

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

[25] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th Annu. ACM Symp. Princ. Distrib. Comput.*, 2016, pp. 267–275.

[26] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," in *Proc. Int. Symp. Distrib. Comput.*, 2008, pp. 323–341.

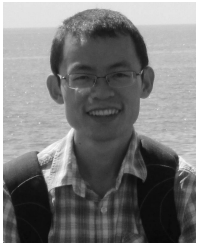
[27] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proc. 17th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2005, pp. 253–262.

[28] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proc. 13th Annu. Symp. Parallel Algorithms Archit.*, 2001, pp. 134–143.

[29] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proc. 16th ACM Symp. Princ. Pract. parallel Program.*, New York, NY, USA, 2011, pp. 223–234.

[30] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2013, pp. 103–112.

[31] M. M. Michael and M. L. Scott, "Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 51, pp. 1–26, May 1998.
[32] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 190–222, Apr. 1983.
[33] M. Torquati, "Single-producer/single-consumer queues on shared cache multi-core systems," *CoRR*, vol. abs/1012.1824, pp. 1–19, Oct. 2010.
[34] J. Wang, Y. Tian, T. Li, and X. Fu, "A flexible communication mechanism for pipeline parallelism," in *Proc. 15th IEEE Int. Symp. Parallel Distrib. Process. Appl.*, Dec. 2017, pp. 778–785.



**JUNCHANG WANG** received the Ph.D. degree in computer science from the University of Science and Technology of China, in 2014. He was a Visiting Student with Yale University for one year. He is currently an Assistant Professor with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include parallel and distributed computing systems.

**YANGFENG TIAN** received the bachelor's degree from the Nanjing University of Posts and Telecommunications, in 2019. He is currently pursuing master's degree in computer technology with the Beijing University of Posts and Telecommunications. His research interests include parallel and distributed computing systems.

**XIONG FU** received the B.S. and Ph.D. degree 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 and Telecommunications. His research interests include parallel and distributed computing, and cloud computing.