A Cache-Friendly Concurrent Lock-Free Queue for Efficient Inter-Core Communication

Xianghui Meng
School of Electrical and Communication Engineering
University of Chinese Academy of Sciences
Beijing, China
e-mail: mengxh@dsp.ac.cn

Xuewen Zeng, Xiao Chen, Xiaozhou Ye
National Network New Media Engineering Research Center
Institute of Acoustics Chinese Academy of Sciences
Beijing, China
e-mail: {zengxw, xxchen, yexz}@dsp.ac.cn

***Abstract*—Buffer sharing based on pipeline parallelism is quite susceptible to inter-core communication overhead. Existing work on concurrent lock-free (CLF) queue algorithm did not take full advantage of CPU cache features to improve performance. In order to implement a fast single-producer-single-consumer (SPSC) buffer scheduling queue, this paper proposes a cache-friendly CLF queue scheduling algorithm (CFCLF), which concentrates on cache-level optimization and minimizing inter-core communication overheads in pipeline parallelism. CFCLF innovatively employs a matrix (2D array), instead of one-dimensional array to design the shared queue structure, making CFCLF has a good cache behavior so as to avoid cache false sharing, and cache consistency problem. Besides, the algorithm implements batch processing efficiently to improve throughput. A deadlock prevention method is also proposed. Experimental results show that on Intel Xeon and Cavium OCTEON, CFCLF outperforms B-Queue which is the state-of-the-art concurrent lock-free queue, by up to 25.5%, and CFCLF is more stable than other algorithms.**

Keywords-concurrent queue; pipeline parallelism; multicore; lock-free; cache-friendly

I. INTRODUCTION

Multicore architectures can greatly benefit throughput driven applications like network applications [1], but conventional sequential applications or single-threaded applications won't gain much benefit if not to be optimized for multicore architecture or multi-thread system. Studies have been conducted to parallelize sequential applications with pipeline parallelism, as multicore architectures become more and more popular [2]. However, the severe inter-core communication overheads have seriously restricted performance of pipelined applications on multicore system.

Traditionally, lock-based queue is used for synchronization between multi-threads of parallel applications. Nevertheless, lock-based approach is very inefficient, because it makes thread accesses serialized, also restricts that only one thread can read or write the whole data buffer at a time. Researchers [3-12] have developed sophisticated algorithms to implement concurrent data structures. They usually avoid using a lock, but depend on subtle mutual relation between threads, and develop different approaches to reduce contention on data buffer. For instance, read thread and write thread can access the buffer at a time at different locations of the buffer indexed by read index and write index, respectively. And if the two indexes meet at a location, one of the thread will suspend.

A concurrent lock-free (CLF) data buffer is used to reduce the overheads. CLF data structure guarantees if multiple threads concurrently access the data, then some thread will finish its operation in finite steps, in spite of the delay and failure of other threads [3,4]. The single-producer-single-consumer (SPSC) problem is a classical data buffer scheduling problem [5], based on which extensive CLF queues [6-12] have been studied to implement data exchanging between consecutive pipeline stages.

Building an efficient inter-core communication needs to avoid synchronization problem, including software synchronization or hardware synchronization which is associated with the cache consistency system. The previous algorithms are not efficient enough, especially they have not adequately exploited cache performance. So we present an improved lock-free queue algorithm. We concentrate on cache-level optimizing, and we subtly use a cache-friendly queue structure instead of a conventional one.

CFCLF queue aims at inter-core communication protocols. It is advantageous to parallel network applications, especially for applications used in data forwarding scenario requiring for high throughput. But it's not suitable for applications which are sensitive to latency. The main contribution of our work can be summarized as follows:

- We present a new cache-friendly CLF algorithm for the parallel pipeline, which employs a 2D array to build the CLF queue structure, making it avoid cache coherence and false sharing problem.
- We propose a deadlock prevention method.
- We conduct experiments to validate our proposed method and algorithm. Experimental results show that the CFCLF algorithm outperforms the state-of-the-art B-Queue algorithm [11].

II. ALGORITHM DESIGN

In this section, firstly we introduce our CLF FIFO data structure which is used to organize the shared buffer, then we describe the algorithm and its operations on SPSC queue. Further, we discuss exception handling.

978-1-5090-3822-0/17/$31.00 ©2017 IEEE

A. Queue Memory Model

Generally, a shared queue buffer can be implemented with a static array or dynamic linked-list. Array-based queue is more efficient than a queue based on linked-list in practice because the former's compact memory footprint exploits cache locality for performance. Memory space of an array buffer is continuous. For a one-dimensional array, if the *head* index and *tail* index are very close to each other, the data element *queue[head]* and *queue[tail]* will be loaded into the same cache line. In the concurrent execution situations, two threads operating on the same cache line will cause cache false sharing [9], which is not cache friendly and not good for performance. To solve this problem, we should make sure that the locations to be read and written keep enough distance from each other so that the two elements won't be loaded in the same cache line.

An overview of our queue structure memory layout is shown as Fig. 1. The queue we build is a 2D-array, i.e., an array of array. We can name the queue itself the outer array, accordingly the queue's element is the inner array. Both the outer array and the inner array are stored sequentially, also the whole 2D-array space is sequential. The element of the inner array is called a datum, which is the actual product that the consumer consumes. We make each single row of the 2D array aligned to a single cache line.

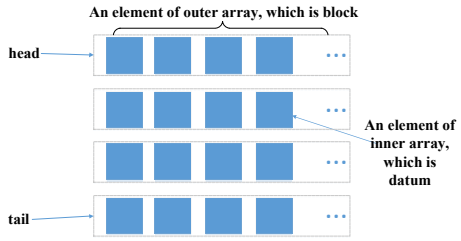

Figure 1. Queue structure from matrix layout

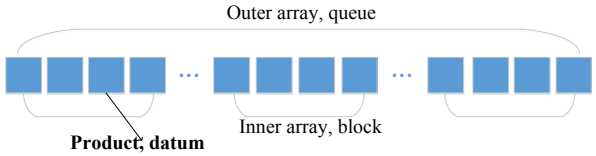

Figure 2. Queue structure from array layout

Producer and consumer operate on the shared buffer queue by a unit of an outer array's element which is a row of the matrix, not a single datum. We call this unit a block. Thus, each outer array's element can occupy a unique cache line. We can consider the 2D-array from another perspective which is more intuitive to understand in Fig. 2. The 2D-array is globally continuous in memory space. We make each of the outer array's element size be the size of a whole cache line. Commonly, a cache line size is 64B, for example if the datum is an integer whose size is 4B in general, then the outer array's element will hold 16 inner array's elements. So each time the consumer makes an operation on the buffer queue, it will consume 16 actual products, which implements a batch consuming and improves the throughput.

As shown in Fig. 1, the index *head* is where the consumer dequeues, and *tail* is where the producer enqueues. For an index *r* between *head* and *tail*, it presents a block *q[r]*. We use *q[r]*'s first datum *q[r][0]* and its last datum *q[r][ELEMENTS_ PER_CACHE-1]* indicate the state of the queue.

B. Cache-Friendly Queue Operations

Unlike clustered queue algorithm [8], CFCLF does not need to define extra flags to maintain the status of data element to indicate whether the buffer slot is full or empty, so it won't cause a waste of memory space. CFCLF uses the value of datum itself to describe the status. Initially, all data elements are initialized to be *EMPTY_VALUE* (depends on the data type) which indicates an empty or invalid value. If the value of a product is *EMPTY_VALUE*, then the producer can enqueue to that slot but consumer cannot dequeue from it. Similarly, *NON_EMPTY* indicates a valid element. Algorithm 1 shows the pseudo code of CFCLF.

Algorithm 1 CFCLF Queue Algorithm

```
1:  function ENQUEUE_ARRAY(T *data)
2:      while queue[tail][0] != EMPTY_VALUE or
3:          queue[tail][ELEMENTS_PER_CACHE − 1]
4:          != EMPTY_VALUE do
5:          do nothing,the array element has not been read
6:      end while
7:      memcpy(queue[tail], data, sizeof(T)∗
8:                  ELEMENTS_PER_CACHE)
9:      ▷ enqueue an array element
10:     tail ← (tail + 1)%QUEUE_SIZE
11: end function
12:
13: function DEQUEUE_ARRAY(T *data)
14:     while queue[head][0] = EMPTY_VALUE or
15:         queue[head][ELEMENTS_PER_CACHE − 1]
16:         = EMPTY_VALUE do
17:         do nothing,there's no complete written array element
18:     end while
19:     memcpy(data, queue[head], sizeof(T)∗
20:                 ELEMENTS_PER_CACHE)
21:     ▷ get an complete array element
22:     queue[head] ← (ELEMENTS_PER_CACHE,
23:                 EMPTY_VALUE)
24:     ▷ clear this array element after dequeue
25:     head ← (head + 1)%QUEUE_SIZE
26: end function
```

For a one-dimensional array named *array* with size of *N*, when we copy buffer data to or from it, it may begin to copy from the start or end position of the array (that depends on the compiler). Now we copy a buffer with *N NON_EMPTY* values to *array*, if both *array[0]* and *array[N-1]* are *NON_EMPTY*, then we are sure that this copying to a whole array is completed. However, if *array[0]= EMPTY_VALUE*

or *array[N-1]=EMPTY_VALUE*, we can tell that the copying process is not finished yet.

Based on the assumption described above, for our 2D-array *queue* we can use the values of inner array's elements *queue[head][0]*, *queue[head][ELEMENTS_PER_CACHE-1]* (as shown in Algorithm 1) to indicate the status of outer array's elements, so as to solve the synchronization between producer and consumer. *ELEMENTS_PER_CACHE* is the result that cache_line_size divides size of the datum (whose type is *ELEMENT_TYPE*), and it indicates how many data elements a single cache line can hold.

The producer executes *ENQUEUE_ARRAY* procedure to insert a batch of several products which constitute an inner array into the tail of the buffer *queue*. First, producer checks whether *queue[tail][0]* and *queue[tail][ELEMENTS_PER_CACHE-1]* equal *EMPTY_VALUE* or not. If any one of the two values is unequal to *EMPTY_VALUE*, it means that this outer array's element *queue[tail]* is holding old values and consumer has not read it yet, which also implies the queue buffer is full now. So the producer will come into a spin loop, until the consumer has dequeued this outer array's element completely and made both *queue[tail][0]* and *queue[tail][ELEMENTS_PER_CACHE-1]* equal to *EMPTY_VALUE*. If producer checks that both of the two values equal *EMPTY_VALUE*, then it directly enqueues data to the outer array's element *queue[tail]* with a copy from a batch of products. Finally, producer increases *tail* index.

Similarly, the consumer executes *DEQUEUE_ARRAY* procedure to extract an outer array's element containing a batch of products from the head of the buffer queue.

Notice that all operations consumer and producer make on buffer queue are by batch processing, so that the throughput is improved and frequency of modifying control variables is reduced to 1/*ELEMENTS_PER_CACHE*.

C. Handling Exception

When the producer halts for some reasons, or it produces fewer products than a batch number, the consumer won't dequeue these products and will wait all the time for more products to be inserted into queue. Thus a deadlock will happen. A simple way to avoid deadlock problem is to add some invalid products into the queue, so that the products will reach a batch number and re-activate the consumer. Then the consumer dequeues these products and discards those invalid data. However, obviously this is a waste of memory space and it's not efficient.

We propose a new method to deal with this exception, as shown in Fig. 3.

We set a counter and a threshold, and originally the variable *N* is equal to batch size. If the counter does not reach the threshold, consumer will always continue to check whether the *q[head][N-1]* equals *EMPTY_VALUE* or not. When the counter reaches the threshold, *N* is decreased by 1 (considering that a cache line size is not large, we decrease *N* by 1), and *q[head][N-2]* will be checked. The loop goes on, until a valid value is found at *q[head][r]* , then all products between *q[head][0]* and *q[head][r]* are consumed at a time.

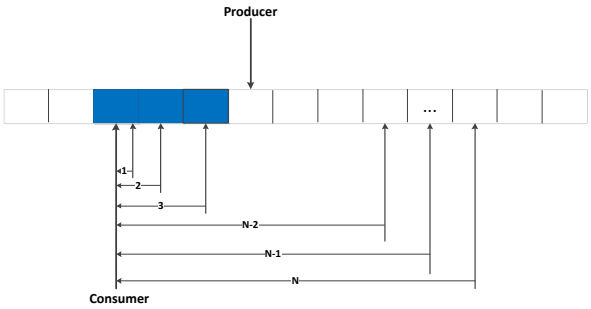

Figure 3. Deadlock protection

III. EXPERIMENTAL RESULTS

In this section, we conduct extensive experiments to verify our algorithm. We mainly compare our CFCLF with B-Queue, also some other algorithms. We obtain B-Queue test code from GitHub, then we modify it and make our experiments.

A. Experiment Environment

We test the algorithms on a DELL Linux server, which has an Intel Xeon E5-2620 CPU (2GHz) and runs Linux 2.6.32. E5-2620 is a 6-core processor, supporting Hyper-Threading and the number of threads is 12. Each core has a 64K L1 cache and 256K L2 cache. All of the cores share a 15MB L3 cache. 16GB memory is installed. We compile all of the algorithm codes using GCC 4.4.7 with –O3 optimization option (including optimization for inline functions). We use perf tool which is originally supported by kernel to instrument CPU performance counters, such as cache-misses.

We define Average Queue Operation Time (AQOT) as $\bar{\tau}$. We assume that in the algorithm the production time of one product is t_p, and consumption time is t_c. Since each datum of the queue will be enqueued and dequeued once, so we can define a single product's queue operation time as $\tau=t_p+t_c$.

Assume that we test for *N* iterations continuously, then the producer will execute enqueue for *N* times and the consumer will execute dequeue for *N* times, so we can derive Eq. (1):

$$\bar{\tau}=\frac{\sum_{k=1}^{N} t_{pk}+t_{ck}}{N} \tag{1}$$

B. Average Queue Operation Time

We use CPU cycles to present average queue operation time. The performance of all queue algorithms are shown in Table I. In this test, we only use one queue, which means there is only one producer and one consumer.

Table I shows that, CFCLF outperforms B-Queue and is almost 4 times faster than Lamport's. In CFCLF queue, consumer and producer would not access the same cache line at the same time, so it avoids cache false sharing and reduces reloading overhead from main memory, thus eventually reduces queue operation time. B-Queue's BACKTRACK-

ING (BACKTRACKING is a deadlock prevention mechanism which is designed for consumer to adaptively find filled slots if producer halts temporarily [11]) can be enabled or disabled while compiling, when enabled it can improve cache performance. As for lock-based queue, it takes more than 500 cycles to insert to and extract from a queue, which is to the disadvantage of fast inter-core communication, and that's why lock-based algorithm is not suitable for parallel pipeline on multicore processor.

TABLE I. PERFORMANCE OF DIFFERENT QUEUES

Algorithm	*AQOT*
Lock-based	544
Lamport's	46
B-Q without BACKTRACKING	16
B-Q with BACKTRACKING	14
MCRingBuffer	15
Clustered	15
CFCLF	12

Also experiment has been made to observe their performance with more parallel queues. In this test, we still use one thread to produce data but more than one threads to consumer data concurrently. Every consumer thread operates on a separate queue on a separate core, so we can fully utilize multiple cores to execute applications. As shown in Fig. 4, regardless of how many queues we test, CFCLF always performs better than other queues because CFCLF is always cache-friendly. CFCLF can always get the data it needs from all levels of cache, thus spends less time compared to others.

Since we only employ one thread to produce data elements, every single consumer will consume more CPU cycles as the number of concurrent queues increases. As for B-Queue, the difference between enabling BACKTRACKING and disabling BACKTRACKING becomes bigger as the number of queues increase, because cache behavior's importance becomes more significant. When there are more concurrent threads to get data from different concurrent queues, cache-unfriendly algorithms will spend more time on loading data from main memory which is much slower than cache.

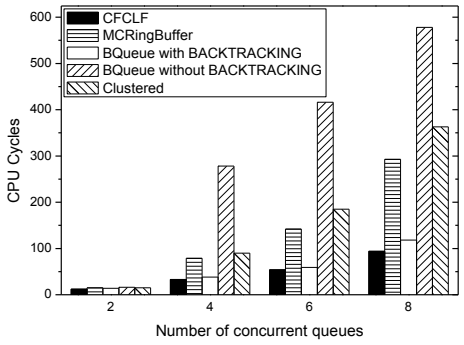

Figure 4. Performance under multiple queues

When the number of concurrent queues goes up to 8, CFCLF is more than 6 times faster than B-Queue without BACKTRACKING, and decreases the average CPU cycles by 25.5% compared to B-Queue with BACKTRACKING.

We also make experiments to test the effect of different size (512, 1024, 2048, 4096, and 8192) of 2D array on CFCLF's performance. Fig. 5 shows the result. We can see that, AQOT with different array size is basically at the same level, which means CFCLF's performance is not sensitive to the queue size. When 2D array size is small, like 512, the performance is slightly worse than those with larger sizes. The reason is that producer generates products too fast and *tail* goes around the buffer and catches up with *head* soon, as a consequence, producer has to wait for a short time. Besides, if the size is too large, this application will consume too much memory space, so we choose the size 4096 as an appropriate one.

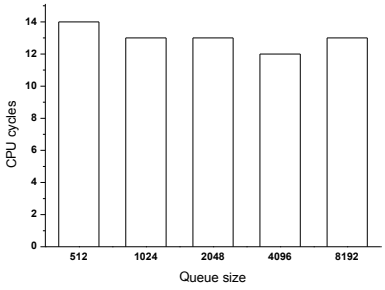

Figure 5. Performance with different array size

C. Cache Behavior

We use perf profiler tool to collect and analyze CPU cache performance data. We execute our program many times and take the average result as a sample point. Then we continuously take 20 runs of perf command so we get 20 sample points, and each one is an average of 5000 iterations. Table II shows the results of cache hit rate.

TABLE II. CACHE HIT RATE OF ALL CLF ALGORITHMS

Algorithm	*Cache hit rate (%)*
B-Q without BACKTRACKING	95.919
B-Q with BACKTRACKING	99.959
MCRingBuffer	99.728
Clustered	98.926
CFCLF	99.987

Both CFCLF and B-Queue's (with BACKTRACKING) cache hit rate can reach up to more than 99%, that's why both of them can implement fast enqueue and dequeue operations on concurrent queues. Cache hit rate of B-Queue without BACKTRACKING is much lower than CFCLF and B-Queue with BACKTRACKING, causing that its AQOT performance is much worse, especially when there are more than 2 concurrent queues.

Furthermore, we present the above 20 continuous cache miss rate sample points in Fig. 6. Apparently, we can see that CFCLF's cache behavior is more stable as time changes, which explains its AQOT is always better than B-Queue and other algorithms. Regardless of how long the CFCLF

algorithm runs, the operating time on each element tends to be a stable result.

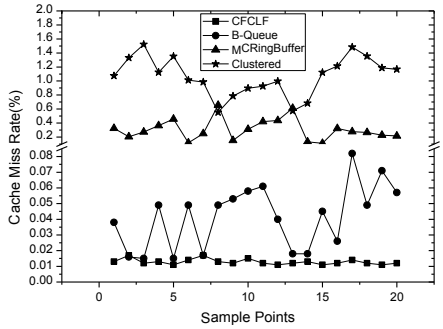

Figure 6. Cache miss rate

D. Real-World Application

Our team is developing a data acquisition and audit system which is based on traffic content analysis. It is running on Cavium OCTEON CN68XX architecture (a MIPS64 SoC with 10 cores). We apply CFCLF to a forwarding module of our system which is a SPSC problem in order to maximize the sending throughput.

We test throughput with 1-4 consumer cores respectively, as shown in Fig. 7. It's clear that CFCLF shows a better throughput performance than B-Queue and other algorithms, with increase of up to 30% and 221%, respectively, especially has a great advantage over Lock-based approach. For all methods, throughput gets higher as the core number increases. When core number is smaller than 4, the throughput basically shows a linear growth, which benefits from more consumer threads executing concurrently on shared data. However, when core number goes up to 4, throughput increases slowly due to OCTEON's protocol stack limitation which is not relevant to queue algorithms.

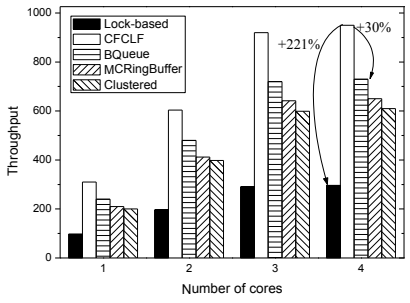

Figure 7. Throughput of real-world applications

IV. CONCLUSION

In this paper, we address the problem of minimizing inter-core communication overheads for buffer sharing based on pipeline parallelism. We propose a novel queue algorithm called CFCLF, which is an efficient SPSC lock-free queue. We innovatively use a 2D array to implement queue structure, making CFCLF has a good cache behavior. CFCLF implements a fast batch operation on products, so it improves throughput obviously. CFCLF has improved the performance of current CLF queues in terms of stability, speed and it is simpler to implement. For now we mainly focused on SPSC problem which is a basic and common situation for many parallel applications, however, there are also many MPMC problem. Therefore our future work will try to address this kind of problem.

ACKNOWLEDGMENT

Our research is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA06010302), and Innovation Institute Foresight Program (Grant No. Y555021601).

REFERENCES

[1] J. Li, J. Chen, H. Ni, and L.F. Wang, "Multi-core Platform Based Multimedia Collaboration Caching Algorithm", Journal of Network New Media, vol.3, issue 4, pp. 12-18, 2014.

[2] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: a dynamic dependence-based parallel execution model", 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, USA, pp. 85-96, 2009.

[3] D. L. Zhang, D. Dechev, "A L-ock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists", IEEE Transactions on Parallel and Distributed Systems, vol.27, issue 3, pp. 613-626,2016.

[4] P. Aggarwal, S. R. Sarangi, "Lock-free and wait-free slot scheduling algorithms", IEEE Transactions on Parallel and Distributed Systems,vol.27, issue 5, pp. 1387-1400, 2016.

[5] M. F. Dolz, R. D. Astorga, J. Fernández, "Embedding Semantics of the Single-Producer/Single-Consumer Lock-Free Queue into a Race Detection Tool", ACM Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, Barcelona, pp.20-29,2016.

[6] L. Lamport, "Specifying Concurrent Program Modules", ACM Transactions on Programming Languages and Systems, vol.5, issue 2, pp. 192-222, 1983.

[7] T. M. Giacomoni, M. Vachharajani, "FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue", ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, pp. 43-52, 2008.

[8] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba, "Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs", Iaeng International Journal of Computer Science, vol.36, issue 4, pp. 275-283, 2009.

[9] P. C. Patrick, T. B. Lee, "A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring", IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, pp.1-12, 2010.

[10] J. S. Preud'homme, G. Thomas, B. Folliot, "BatchQueue: Fast and Memory-thrifty Core to Core Communication", 22nd International Symposium on Computer Architecture and High Performance Computing, Petrópoli, pp.215-222, 2010.

[11] J. Wang, K. Zhang, X. Tang, and B. Hua, "B-Queue: Efficient and practical queuing for fast core-to-core communication", International Journal of Parallel Programming, vol. 41, issue 1, pp.137-159, 2013.

[12] K. Mitropoulou, V. Porpodas, X. Zhang, "Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication", Proceedings of the 2016 International Conference on Supercomputing, Istanbul, pp.1-12, 2016.