IEICE TRANS. INF. & SYST., VOL.E95–D, NO.12 DECEMBER 2012

LETTER *Special Section on Parallel and Distributed Computing and Networking*

Scalable Cache-Optimized Concurrent FIFO Queue for Multicore Architectures**

Changwoo MIN[†*], **Hyung Kook JUN**[††], **Won Tae KIM**[††], ***Nonmembers***, ***and*** **Young Ik EOM**[†a)], ***Member***

SUMMARY A concurrent FIFO queue is a widely used fundamental data structure for parallelizing software. In this letter, we introduce a novel concurrent FIFO queue algorithm for multicore architecture. We achieve better scalability by reducing contention among concurrent threads, and improve performance by optimizing cache-line usage. Experimental results on a server with eight cores show that our algorithm outperforms state-of-the-art algorithms by a factor of two.

key words: *FIFO queue, multicore processor, cache-line contention, compare-and-swap, fetch-and-store*

1. Introduction

Multicore architecture has been widely adopted from smart phones to servers in data centers. A concurrent FIFO queue is a widely used fundamental data structure for parallelizing software to fully exploit capability of multicore architecture. Efficient and scalable concurrent FIFO queue is critically important to build high performance producer/consumer and software pipeline architecture. Lamport [1] proposed a lockless concurrent FIFO queue. It uses a fixed size array and supports single enqueuer and single dequeuer. FastForward [2] improved performance of the Lamport queue by optimizing cache-line usage for modern multicore architecture. Michael and Scott [3] introduced a lock-free FIFO queue, called MS-queue, which uses singly linked list and supports multiple concurrent enqueuers and dequeuers. MS-queue updates `Head` and `Tail` using compare-and-swap (CAS) atomic primitive. When a CAS operation fails because of contention among concurrent threads, it retries the operation until succeeds. Because enqueue requires two successful CAS operations whereas dequeue requires a single successful CAS operation in order to complete, possibility of CAS failure in enqueue is higher than that of dequeue [4]. In modern multicore architectures, a CAS operation costs an order-of-magnitude longer than a load or store, since they require exclusive ownership and flushing of the processors store buffer. Ladan-Mozes and Shavit [4] introduced a doubly linked list based lock-free FIFO queue, called LS-queue, to reduce the possibility of CAS failure in enqueue. They showed that reducing the number of CAS failure improves performance and scalability.

In this letter, we propose a novel concurrent FIFO queue. It uses singly linked list and supports multiple concurrent enqueuers and dequeuers. To achieve high performance and scalability in multicore architecture, we propose two schemes. First, we design an enqueue algorithm that does not require retrying failed CAS. LS-queue reduces possibility of CAS failure in enqueue when comparing it to the MS-queue, but the number of failed CAS operations is still high as we shown in Fig. 2 (b). Because our algorithm needs to retry failed CAS operation only in dequeue, it significantly reduces the number of failed CAS operations: it significantly improves performance and scalability. Second, we further improve performance by reducing cache-line interference between enqueuer and dequeuer unlike MS-queue and LS-queue. Since, in our algorithm, `Tail` is read only by enqueuer and `Head` is read only by dequeuer, there is no shared cache-line between enqueuer and dequeuer: no cache-line interference between enqueuer and dequeuer.

2. The New Queue Algorithm

We introduce two techniques for our new queue algorithm. First, we fundamentally remove the need of retrying failed CAS in enqueuer. Instead of CAS, we update `Tail` by using fetch-and-store (FAS) primitive. FAS atomically updates a memory location to a new value and returns the old value. Most architectures including Intel x86, ARM, and PowerPC support FAS primitive in hardware. Figure 1 depicts the flow of enqueue and dequeue in the new algorithm. In enqueue operation, it first atomically updates `Tail` to new node using FAS (E1), and then updates old `Tail`'s `next` to the new node (E3). Since FAS primitive guarantees atomic update of `Tail`, it is possible to enqueue without retrying even under

Manuscript received January 6, 2012.
Manuscript revised March 2, 2012.

[†]The authors are with School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea.
[††]The authors are with Software Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Korea.
[*]Presently, with Samsung Electronics, Suwon, Korea.

[**]This work was supported by IT R&D Program of MKE/KEIT [10035708] and Next-Generation Information Computing Development Program of NRF [2012-0006423].

a) E-mail: yieom@ece.skku.ac.kr

DOI: 10.1587/transinf.E95.D.2956

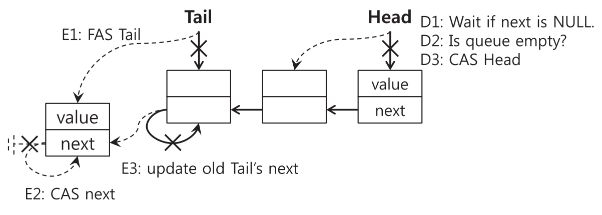
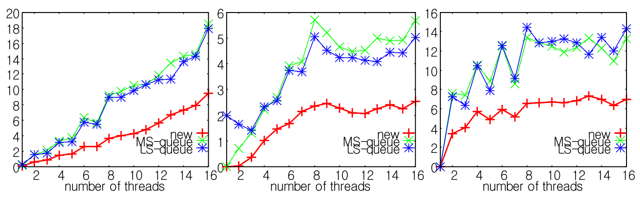


Fig. 1 The flow of enqueue and dequeue in the new algorithm.



(a) Time (millisecond) (b) Num. of failed CAS operations (c) Num. of cache miss

Fig. 2 Performance comparison for one enqueue-dequeue pair.

concurrent operations. Second, to avoid unnecessary cache-line invalidation that incurs expensive cost in multicore architectures, we propose a new technique to check whether a queue is empty or not. Before enqueuing a new node, its next is initialized to `NULL` . After `Tail` is updated to the new node (E1), it updates the `next` of the new node to itself using CAS (E2). Thus, a node whose `next` points to itself is the last node in the queue. We can easily recognize that a queue is empty if the next of the `Head` node points to itself (D2).

In dequeue operation, it first checks if the `next` of the `Head` node is `NULL` (D1) and waits until it is changed to non-`NULL`. That is because `next` is `NULL` while enqueue is in progress. In case that the queue is not empty (D2), `Head` is updated to the `next` using CAS (D3). If the CAS operation succeeds, it returns the value. Otherwise, it retries the dequeue operation. Since `Head` is updated and retried using CAS, we therefore must deal with ABA problem [3]. We use a tagging technique [3], [4] that associates a modification counter with a pointer (`pointer_t`) and increments it in each successful CAS. In Algorithm 1, we describe the new algorithm in detail.

3. Evaluation

We performed our experiments on a server that has 8 cores (two Intel Xeon 5506 2.13 GHz Quad-core processors) with Linux Kernel 3.0. All the experiments employ an initially empty queue to which threads perform 10 million pairs of enqueue and dequeue. We measured the time, the number of failed CAS operations, and the number of cache miss for one enqueue-dequeue pair. Figure 2 (a) shows that the performance and scalability of the new algorithm is significantly better than those of MS-queue and LS-queue. In case of running 16 concurrent threads, MS-queue and LS-queue are slower than the new algorithm by 195% and 188%, respectively. As shown in Fig. 2 (b), the number of failed CAS operations in the new algorithm is significantly lower, which mainly contributes better performance and scalability. For 16 concurrent threads, the number of failed CAS operations in MS-queue and LS-queue are greater by 225% and 198%, respectively. As shown in Fig. 2 (c), the number of cache miss in the new algorithm is significantly lower. For 16 concurrent threads, the number of cache miss in MS-queue and LS-queue are greater by 189% and 204%, respectively.

Algorithm 1 The New Queue Algorithm

```
structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
structure node_t {value: data type, next: pointer_t}
structure queue_t {Head: pointer_t cacheline aligned , Tail: pointer
    to node_t cacheline aligned }
procedure INITIALIZE(Q: pointer to queue_t)
    node = new_node()                          ▷ First node is a sentinel node.
    node→next.ptr = node
    Q→Head = Q→Tail = node
end procedure
procedure ENQUEUE(Q: pointer to queue_t, value: data type)
    node = new_node()
    node→value = value
    node→next.ptr = NULL
    old_tail = FAS(&Q→Tail.ptr, node)
    CAS(&node→next.ptr, NULL, node)
    old_tail→next.ptr = node
end procedure
procedure DEQUEUE(Q: pointer to queue_t, pvalue: pointer to data
    type)
    loop
        if Q→Head.ptr→next.ptr == NULL then
            continue
        end if
        head = Q→Head
        next = head→next
        if head.ptr == next.ptr then
            return FALSE
        end if
        if !CAS(&Q→Head, head, <next.ptr, head.count+1>) then
            continue
        end if
        *pvalue = next.ptr→value
        free(head→ptr)
        break
    end loop
    return TRUE
end procedure
```

4. Conclusion

Concurrent FIFO queue is a fundamental data structure, which is critical to high performance parallel software. We proposed a concurrent FIFO queue algorithm that significantly improves performance and scalability by reducing the number of failed CAS operations and optimizing cache-line usage. Our experiments show the new algorithm outperforms the state-of-the-art algorithms by factor of two.

References

[1] L. Lamport, "Specifying concurrent program modules," ACM Trans. Program. Lang. Syst., vol.5, pp.190–222, April 1983.

[2] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp.43–52, 2008.

[3] M.M. Michael and M.L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," Proc. Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '96, pp.267–275, 1996.

[4] E. Ladan-mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," Proc. 18th International Symposium on Distributed Computing, LNCS 3274, pp.117–131, Springer, 2004.