Integrating Lock-Free and Combining Techniques for a Practical and Scalable FIFO Queue

Changwoo Min and Young Ik Eom

Abstract—Concurrent FIFO queues can be generally classified into lock-free queues and combining-based queues. Lock-free queues require manual parameter tuning to control the contention level of parallel execution, while combining-based queues encounter a bottleneck of single-threaded sequential combiner executions at a high concurrency level. In this paper, we introduce a different approach using both lock-free techniques and combining techniques synergistically to design a practical and scalable concurrent queue algorithm. As a result, we have achieved high scalability without any parameter tuning: on an 80-thread average throughput in our experimental results, our queue algorithm outperforms the most widely used Michael and Scott queue by 14.3 times, the best-performing combining-based queue by 1.6 times, and the best performing $\times$86-dependent lock-free queue by 1.7 percent. In addition, we designed our algorithm in such a way that the life cycle of a node is the same as that of its element. This has huge advantages over prior work: efficient implementation is possible without dedicated memory management schemes, which are supported only in some languages, may cause a performance bottleneck, or are patented. Moreover, the synchronized life cycle between an element and its node enables application developers to further optimize memory management.

Index Terms—Concurrent queue, lock-free queue, combining-based queue, memory reclamation, compare-and-swap, swap

1 Introduction

The FIFO queues are one of the most fundamental and highly studied concurrent data structures. They are essential building blocks of libraries [1], [2], [3], runtimes for pipeline parallelism [4], [5], and high performance tracing systems [6]. These queues can be categorized according to whether they are based on static allocation of a circular array or on dynamic allocation in a linked list, and whether or not they support multiple enqueuers and dequeuers. This paper focuses on dynamically allocated FIFO queues supporting multiple enqueuers and dequeuers. Although extensive research has been performed to develop scalable and practical concurrent queues, there are two remaining problems that limit wider practical use: (1) *scalability* is still limited in a high level of concurrency or it is difficult to achieve without sophisticated parameter tuning; and (2) *the use of dedicated memory management schemes* for the safe reclamation of removed nodes imposes unnecessary overhead and limits the further optimization of memory management [7], [8], [9].

In terms of scalability, avoiding the contended hot spots, `Head` and `Tail`, is the fundamental principle in designing concurrent queues. In this regard, there are two seemingly contradictory approaches. *Lock-free approaches* use fine-grain synchronization to maximize the degree of parallelism and thus improve performance. The MS queue presented by Michael and Scott [10] is the most well-known algorithm, and many works to improve the MS queue have been proposed [10], [11], [12], [13], [14]. They use `compare-and-swap` (`CAS`) to update `Head` and `Tail`. In the `CAS`, however, of all contending threads, only one will succeed, and all the other threads will fail and retry until they succeed. Since the failing threads use not only computational resources, but also the memory bus, which is a shared resource in cache-coherent multiprocessors, they also slow down the succeeding thread. A common way to reduce such contention is to use an *exponential backoff* scheme [10], [15], which spreads out `CAS` retries over time. Unfortunately, this process requires manual tuning of the backoff parameters for a particular workload and machine combination. To avoid this disadvantage, Morrison and Afek [16] proposed a lock-free queue based on `fetch-and-add` (`F&A`) atomic instructions, which always succeeds but is $\times$86-dependent. Though the $\times$86 architecture has a large market share, as the core count increases, industry and academia have actively developed various processor architectures to achieve high scalability and low power consumption and thus we need more portable solutions.

In contrast, *combining approaches* [17], [18], [19] use an opposite strategy to the lock-free approaches. A single *combiner* thread, that acquires a global lock at a time, combines all concurrent requests from other threads, and then performs their combined requests. In the meantime, each thread that does not hold the lock busy waits until either its request has been fulfilled by the combiner or the global lock has been released. This technique has the dual benefit of reducing the synchronization overhead on hot spots, and at the same time reducing the overall cache invalidation traffic. So the waiting thread does not slow down the combiner's performance.

• *The authors are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Gyeonggi-do, Korea. E-mail: {multics69, yieom}@skku.edu.*

Manuscript received 12 Jan. 2014; revised 17 June 2014; accepted 20 June 2014. Date of publication 24 June 2014; date of current version 5 June 2015.
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2333007

However, this comes at the expense of parallel execution and, thus, at a high concurrency level, the sequential execution becomes a performance bottleneck [20].

Memory management in concurrent queue algorithms is a non-trivial problem. In previous work, nodes that have been already dequeued cannot be freed using a standard memory allocator, and dedicated memory management schemes are needed. There are two reasons for this. First, the last dequeued node is recycled as a *dummy node* pointed by `Head` so it is still in use. Second, even after a subsequent `dequeue` operation makes the dummy node get out of the queue, the old dummy node, which is not in the queue, can be accessed by other concurrent operations (also known as *the repeat offender problem* [8] or *read/reclaim races* [9]). Therefore, dedicated memory management schemes need to be used, such as *garbage collection* (GC), *freelists*, *lock-free reference counting* [21], [22], and *hazard pointers* [7], [8], [23]. However, they are not free of charge; GC is only supported in some languages such as Java; freelists have large space overhead; others are patented [21], [22] or under patent application [23]. Moreover, the mismatch at the end of life between an element and its node limits further optimization of the memory management.

In this paper, we propose a scalable out-of-the-box concurrent queue, LECD queue, which requires neither manual parameter tuning nor dedicated memory management schemes. We make the following contributions:

- We argue that prior work is stuck in a dilemma with regard to scalability: lock-free techniques require the manual tuning of parameters to avoid the contention meltdown, but combining techniques lose opportunities to exploit the advantages of parallel execution.

  In this regard, we propose a linearizable concurrent queue algorithm: *Lock-free Enqueue and Combining Dequeue (LECD) queue*. In our LECD queue, `enqueue` operations are performed in a lock-free manner, and `dequeue` operations are performed in a combining manner. We carefully designed the LECD queue so that the LECD queue requires neither retrying atomic instructions nor tuning parameters for limiting the contention level; a SWAP instruction used in the `enqueue` operation always succeeds and a CAS instruction used in the `dequeue` operation is designed not to require retrying when it fails. Using the combining techniques in the `dequeue` operation significantly improves scalability and has additional advantages together with the lock-free `enqueue` operation: (1) by virtue of the concurrent `enqueue` operations, we can prevent the combiner from becoming a performance bottleneck, and (2) the higher combining degree incurred by the concurrent `enqueue` operations makes the combining operation more efficient. To our knowledge, the LECD queue is the first concurrent data structure that uses both lock-free and combining techniques.
- Using dedicated memory management schemes is another aspect that hinders the wider practical use of prior concurrent queues. The fundamental reason for using dedicated schemes is in the mismatch at the end of life between an element and its node.

  In this regard, we made two non-traditional, but practical, design decisions. First, to fundamentally avoid read/reclaim races, the LECD queue is designed for one thread to access `Head` and `Tail` at a time. Since `dequeue` operations are serialized by a single threaded combiner, only one thread accesses `Head` at a time. Also, a thread always accesses `Tail` through a private copy obtained as a result of `SWAP` operation, so there is no contention on accessing the private copy. Second, we introduce a *permanent dummy node* technique to synchronize the end of life between an element and its node. We do not recycle the last dequeued node as a dummy node. Instead, the initially allocated dummy node is *permanently* used by updating `Head`'s `next` instead of `Head` in `dequeue` operations. Our approach has huge advantages over prior work. Efficient implementations of the LECD queue are possible, even in languages which do not support GC, without using dedicated memory management schemes. Also, synchronizing the end of life between an element and its node opens up opportunities for further optimization, such as by embedding a node into its element.
- We compared our queues to the state-of-the-art lock-free queues [10], [16] and combining-based queues [19] in a system with 80 hardware threads. Experimental results show that, except for queues [16] which support only Intel $\times 86$ architecture, the LECD queue performs best. Even in comparison with the $\times 86$-dependent queue [16] whose parameter is manually tuned beforehand, the LECD queue outperforms the $\times 86$-dependent queue in some benchmark configurations. Since the LECD queue requires neither parameter tuning nor dedicated memory management schemes, we expect that the LECD queue can be easily adopted and used in practice.

The remainder of this paper is organized as follows: Section 2 describes related work and Section 3 elaborates our LECD queue algorithm. Section 4 shows the extensive evaluation results. The correctness of the LECD queue is discussed in Section 5. Finally, in Section 6, we conclude the paper.

2 Related Work

Concurrent FIFO queues have been studied for more than a quarter of a century, starting with work by Treiber [24]. In this section, we will elaborate on the lock-free queues in Section 2.1, the combining-based queues in Section 2.2, and `SWAP`-based queues in Section 2.3. Finally, we will explain the prior work on memory management schemes in Section 2.4.

2.1 Lock-Free Queues

MS Queue presented by Michael and Scott [10] is the most widely used lock-free queue algorithm. It updates `Head`, `Tail`, and `Tail`'s `next` in a lock-free manner using `CAS`. When the `CAS` fails, the process is repeated until it succeeds. However, when the concurrency level is high, the frequent `CAS` retries result in a contention meltdown [18], [19]. Though bounded exponential backoff delay is used to reduce such contention, manual tuning of the backoff parameters is required for the particular combinations of workloads and machines [14]. Moreover, if they are backed off too far, none of the competing threads can progress. Consequently, many implementations [1], [2], [3] are provided without the backoff scheme.

Ladan-Mozes and Shavit [11] introduced the *optimistic queue*. The optimistic queue reduces the number of CAS operations in an enqueue operation from two to one. The smaller number of necessary CAS operations also reduces the possibility of CAS failure and contributes to improving scalability. However, since the queue still contains CAS retry loops, it suffers from the CAS retry problem and manual backoff parameter tuning.

Moir et al. [13] used elimination as a backoff scheme of the MS queue to allow pairs of concurrent enqueue and dequeue operations to exchange values without accessing the shared queue itself. Unfortunately, the *elimination backoff queue* is practical only for very short queues because the enqueue operation cannot be eliminated until all previous values have been dequeued in order to keep the correct FIFO queue semantics.

Hoffman et al. [14] reduced the possibility of CAS retries in an enqueue operation by creating *baskets* of mixed-order items instead of the standard totally ordered list. Unfortunately, creating a *basket* in the enqueue operation imposes a new overhead in the dequeue operation: linear search between Head and Tail is required to find the first non-dequeued node. Moreover, a backoff scheme is still needed to limit the contention among losers who failed the CAS. Consequently, in some architectures, the *baskets queue* performs worse than the MS queue [18].

Morrison and Afek [16] recently proposed *LCRQ* and *LCRQ+H* (LCRQ with hierarchical optimization). LCRQ is an MS queue where a node is a concurrent circular ring queue (CRQ). If the CRQ is large enough, enqueue and dequeue operations can be performed using F&A without CAS retries. Since LCRQ relies on F&A and CAS2, which are supported only in Intel ×86 architectures, porting to other architectures is not feasible. To obtain the best performance, hierarchical optimization (LCRQ+H), which manages contention among clusters, is essential. If the running cluster ID of a thread is different from the currently scheduled cluster ID, the thread voluntarily yields for *a fixed amount of time* and then preempts the scheduled cluster to proceed. This creates batches of operations that complete on the same cluster without interference from remote clusters. Since it reduces costly cross-cluster traffic, this process improves performance. However, the yielding time should be manually tuned for a particular workload and machine combination. Moreover, at a low concurrency level with little chance of aggregation on the same cluster, the voluntarily yielding could result in no aggregation and thus could degrade performance. We will investigate the performance characteristics of LCRQ and LCRQ+H in Section 4.4.

2.2 Combining-Based Queues

In combining techniques, a single thread, called the *combiner*, serves, in addition to its own request, active requests published by the other threads while they are waiting for the completion of processing their requests in some form of spinning. Though the single-threaded execution of a combiner could be a performance bottleneck, the combining techniques could outperform traditional techniques based on fine-grain synchronization when the synchronization cost overshadows the benefit of parallel execution. A combining technique is essentially a universal construction [25] used to construct a concurrent data structure from a sequential implementation. In most research, combining-based queues are constructed from the two-lock queue presented by Michael and Scott [10] by replacing the locks with combining constructions, so they are blocking algorithms with no read/reclaim races. The first attempt at combining operations dates back to the software combining tree proposed by Yew et al. [26]. Since then, most research efforts have been focused on minimizing the overhead of request management.

Oyama et al. [17] presented a combining technique which manages announced requests in a stack. Though serving requests in LIFO order could reduce the cache miss rate, contention on updating the stack Top with a CAS retry loop could result in contention meltdown under a high level of contention.

In the *flat combining* presented by Hendler et al. [18], the list of announced requests contains a request for each thread independent of whether the thread currently has an active request. Though this reduces the number of insertions to the list, each request in the list should be traversed regardless of whether it is active or not. The unnecessary scanning of inactive requests decreases the efficiency of combining as concurrency increases.

Fatourou and Kallimanis presented a blocking combining construction called *CC-Synch* [19], in which a thread announces a request using SWAP, and *H-Synch* [19], which is an optimized CC-Synch for clustered architectures such as NUMA. H-Synch manages the lists of announced requests for each cluster. The combiner threads, which are also per-cluster and synchronized by a global lock, process the requests from their own clusters. Among the combining techniques proposed so far, H-Synch performs best, followed by CC-Synch. Therefore, among combining-based queues, H-Queue, which is a queue equipped with H-Synch, performs best, followed by CC-Queue, which uses CC-Synch.

2.3 SWAP-Based Queues

Though the majority of concurrent queues are based on CAS, there are several queue algorithms based on SWAP (or fetch-and-store). Mellor-Crummey [27] proposed a SWAP-based concurrent queue which is linearizable but blocking. Since enqueue and dequeue operations access both Head and Tail, enqueuers and dequeuers interfere each other's cacheline and thus result in limited scalability. Min et al. [12] proposed a *scalable cache-optimized queue*, which is also linearizable but blocking. They completely remove CAS failure in enqueue operation by replacing CAS with SWAP and significantly reduce cacheline interference between enqueuers and dequeuers. Though the queue shows better performance than the *optimistic queue* [11], it still contains a CAS retry loop in dequeue operation. While these two algorithms support *multiple enqueuers and multiple dequeuers*, there are a few SWAP-based queue algorithms which support *multiple enqueuers and single dequeuer*. Vyukov's queue [28] is non-linearizable and non-blocking; Desnoyers and Jiangshan's queue [29] is linearizable but blocking.

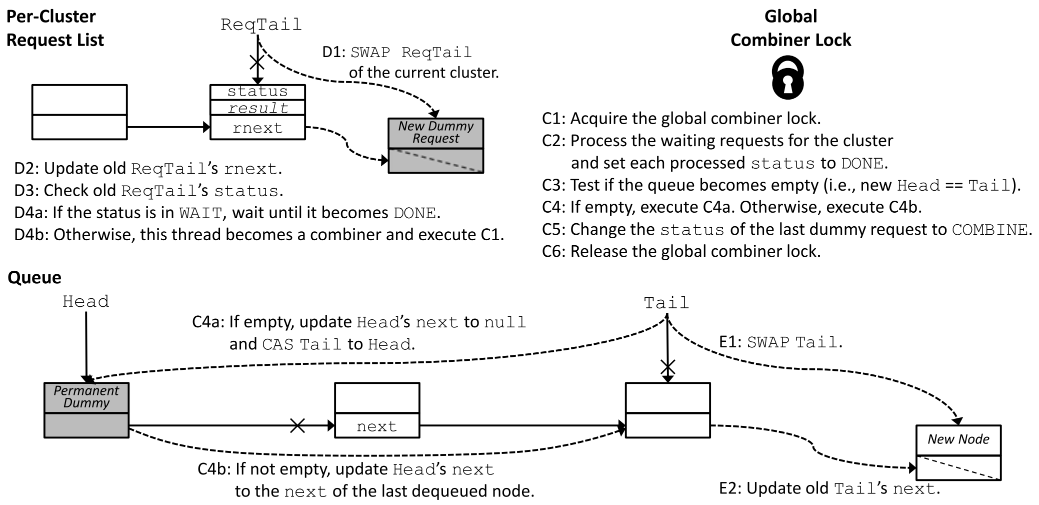
Per-Cluster Request List
ReqTail
D1: SWAP ReqTail of the current cluster.
status
result
rnext
New Dummy Request
D2: Update old ReqTail's rnext.
D3: Check old ReqTail's status.
D4a: If the status is in WAIT, wait until it becomes DONE.
D4b: Otherwise, this thread becomes a combiner and execute C1.
Global Combiner Lock
C1: Acquire the global combiner lock.
C2: Process the waiting requests for the cluster and set each processed status to DONE.
C3: Test if the queue becomes empty (i.e., new Head == Tail).
C4: If empty, execute C4a. Otherwise, execute C4b.
C5: Change the status of the last dummy request to COMBINE.
C6: Release the global combiner lock.
Queue
Head
Tail
C4a: If empty, update Head's next to null and CAS Tail to Head.
E1: SWAP Tail.
Permanent Dummy
next
New Node
C4b: If not empty, update Head's next to the next of the last dequeued node.
E2: Update old Tail's next.

Fig. 1. The overall flow of the LECD queue algorithm. `Enqueue` operation and `dequeue` operation are shown in E1-E2 and in D1-D4, respectively. Combining `dequeue` operation is shown in C1-C6.

2.4 Memory Management

Including list-based queues, dynamic-sized concurrent data structures that avoid locking face the problem of reclaiming nodes that are no longer in use and the ABA problem [7], [8], [9]. In case of concurrent queue algorithms, before releasing a node in a `dequeue` operation, we must ensure that no thread will subsequently access the node. When a thread releases a node, some other contending thread, which has earlier read a reference to that node, is about to access its contents. If the released node is arbitrarily reused, the contending thread might corrupt the memory, which was occupied by the node, return the wrong result, or suffer an access error by dereferencing an invalid pointer.

In garbage-collected languages such as Java, those problems are subsumed into automatic garbage collectors, which ensures that a node is not released if any live reference to it exists. However, for languages like C, where memory must be explicitly reclaimed, previous work proposed various techniques including managing a freelist with an ABA tag [10], [11], [13], [14], [24], lock-free reference counting [21], [22], quiescent-state-based reclamation [30], [31], [32], [33], epoch-based reclamation [34], and hazard-pointer-based reclamation [7], [8], [23].

A common approach is to tag values stored in nodes and access such values only through `CAS` operations. In algorithms using this approach [10], [11], [13], [14], [24], a `CAS` applied to a value after the node has been released will fail, so the contending thread detects whether the node is already released or not. However, since a thread accesses a value which is in previously released memory, the memory used for tag values cannot be used for anything else. To ensure that the tag memory is never reused for other purposes, a freelist explicitly maintains released nodes. An important limitation of using a freelist is that queues are not truly dynamic-sized: if the queues grow large and subsequently shrink, then the freelist contains many nodes that cannot be reused for any other purposes. Also, a freelist is typically implemented using Treiber's stack [24] — a `CAS`-based lock-free stack, and its scalability is fundamentally limited by high contention on updating the stack top [35].

Another approach is to distinguish between removal of a node and its reclamation. Lock-free reference counting [21], [22] has high overhead and scales poorly [9]. In epoch-based reclamation [34] and hazard-pointer-based reclamation [7], [8], [23], readers explicitly maintain a list of currently accessing nodes to decide when it is safe to reclaims nodes. In quiescent-state-based reclamation [30], [31], [32], [33], safe reclamation times (i.e., quiescent states) are inferred by observing the global system state. Though these algorithms can reduce space overhead, they impose additional overhead, such as atomic operations and barrier instructions. Hart et al. [9] show that the overhead of inefficient reclamation can be worse than that of locking and, unfortunately, there is no single optimal scheme: data structure, workloads, and execution environments can dramatically affect the memory reclamation performance.

In practice, since most memory reclaim algorithms are patented or under patent application [21], [22], [23], [30], most implementations [2], [3] written in C/C++ rely on freelists based on a Treiber's stack.

3 The LECD Queue

In this section, we elaborate the LECD queue algorithm and its memory management in Sections 3.1 and 3.2, respectively, and then discuss its linearizability and progress property in Section 3.3.

The LECD queue is a list-based concurrent queue which supports concurrent enqueuers and dequeuers. The LECD queue performs `enqueue` operations in a lock-free manner and performs `dequeue` operations in a combining manner in order to achieve high scalability at a high degree of concurrency without manual parameter tuning and dedicated memory management schemes. We illustrate the overall flow of the LECD queue in Fig. 1. In the `enqueue` operation, we first update `Tail` to a new node using `SWAP` (E1) and update the old `Tail`'s `next` to the new node (E2). In the `dequeue` operation, we first enqueue a request into a request list using `SWAP` (D1, D2) and then determine whether the current thread takes the role of a combiner (D3). If it does not take the role of a combiner, it waits until the enqueued request is processed by the

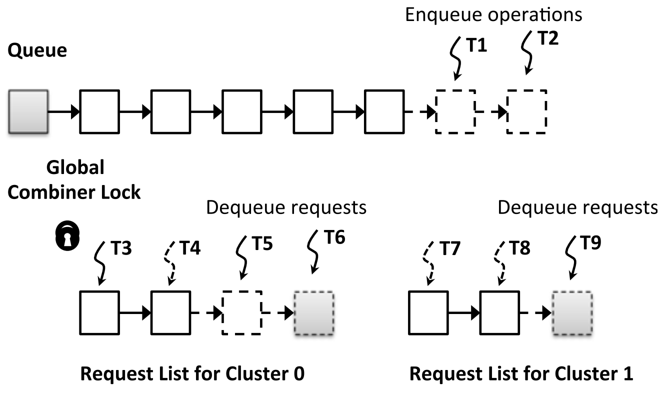

Fig. 2. An illustrative example of the LECD queue, where nine concurrent threads (T1-T9) are running. A curved arrow denotes a thread; a solid one is in a running state and a dotted one is in a blocked state. A gray box denotes a dummy node and a dotted box denotes that a thread is under operation. T1 and T2 are enqueuing nodes. T3 and T7 take a combiner role for cluster 0 and 1, respectively. T3 acquires the lock and performs dequeue operations. In contrast, T7 is waiting for the lock. Non-combiner threads, T4 and T8, are waiting for the completion of operations by the combiner. T5, T6, and T9 are adding their dequeue requests to the list.

combiner (D4a). Otherwise (D4b), it, as a combiner, processes pending requests in the list (C1-C6). Many-core systems are typically built with clusters of cores such that communication among the cores of the same cluster is performed much faster than that among cores residing in different clusters. Intel Nehalem and Sun Niagara 2 are examples of cluster architecture. To exploit the performance characteristics of cluster architecture, we manage a request list for each cluster, similar to H-Synch [19]. The execution of per-cluster combining threads is serialized by the global combiner lock (C1, C6). The combiner thread processes pending requests for the cluster (C2) and checks whether the queue has become empty (C3). Similar to the MS queue, we use a dummy node pointed by `Head` to check whether the queue is empty. In the MS queue, the last dequeued node is recycled into a dummy node, while our dummy node is allocated at queue initialization and *permanently* used. Since this makes the life cycle of a dequeued element and its queue node the same, more flexible memory management, such as embedding a queue node to an element, is possible. To this end, we update `Head`'s `next` instead of `Head` when the queue is not empty (C4b). When the queue becomes empty, we update `Tail` to `Head` and `Head`'s `next` to `null` using `CAS` (C4a). Since the `CAS` operation is used to handle concurrent updates from enqueuers, no retry loop is needed. In this way, since the LECD queue is based on a `SWAP` atomic primitive, which always succeeds, unlike `CAS`, and a `CAS` with no retry loop, parameter tuning is not required to limit the contention level. We illustrate an example of the LECD queue running in nine concurrent threads (T1-T9). As Fig. 2 shows, the LECD queue concurrently performs three kinds of operations: `enqueue` operation (T1, T2), `dequeue` combining operation (T3), and adding new `dequeue` requests (T5, T6 and T9).

There are interesting interactions between `enqueue` operations and `dequeue` operations. Since `enqueue` operations are performed in parallel using a lock-free manner, threads spend most of their time executing `dequeue` operations. In this circumstance, a combiner can process a longer list of requests in a single lock acquisition. This results in more efficient combining operations by reducing locking overhead and context switching overhead among combiners. Consequently, the LECD queue significantly outperforms the previous combining-based queues.

3.1 Our Algorithm

We present the pseudo code of the LECD queue in Fig. 3. The LECD queue is based on widely supported `SWAP` and `CAS` instructions. `SWAP(a,v)` atomically writes the value of `v` into `a` and returns the previous value of `a`. `CAS(a,o,v)` atomically checks whether the value of `a` is `o`, and if they are the same, it writes the value of `v` into `a` and returns `true`, otherwise it returns `false`. Though `SWAP` is a less powerful primitive than `CAS`, it always succeeds, so neither the retry loop nor contention management, such as a backoff scheme [10], [11], [14] is needed. `CAS` is the most widely supported atomic instruction, and `SWAP` is also supported in most hardware architectures, including $\times$86, ARM, and Sparc. Thus, the LECD queue can work on various hardware architectures.

3.1.1 *Initialization*

In the LECD queue, `Head` points to a dummy node and `Tail` points to the last enqueued node. Nodes in the queue are linked via `next` in the order of `enqueue` operations. When a queue is empty with no element, both `Head` and `Tail` point to a dummy node. When initializing a queue, we set `Head` and `Tail` to a newly allocated permanent dummy node (line 23) and initialize per-cluster request lists (lines 24-27).

3.1.2 `Enqueue` *Operation*

We first set the `next` of a new node to `null` (line 31), and then update `Tail` to the new node using `SWAP` (line 32). As a result of the `SWAP` operation, we atomically read the old `Tail`. Finally, we update old `Tail`'s `next` to the new node (line 33). The old `Tail`'s `next` is updated using a simple `store` instruction since there is no contention on updating. In contrast to previous lock-free queues [10], [11], [14], our `enqueue` operation always succeeds regardless of the level of concurrency. Moreover, since any code except `SWAP` can be executed independently with other concurrent threads, an enqueuer can enqueue a new node, even in front of a node being inserted by another concurrent `enqueue` operation (between lines 32 and 33). Thus, we can fully exploit the advantages of parallel execution. However, there could be races between writing the `old_tail`'s `next` in enqueuers and reading the `old_tail`'s `next` in dequeuers. In Section 3.3, we will elaborate on how to handle the races to preserve linearizability.

3.1.3 `Dequeue` *Operation*

Request structure. The per-cluster request list contains dequeue requests from dequeuing threads in a cluster and maintains the last request in the list as a dummy request. An initial dummy request is allocated for each cluster at queue initialization (lines 24-27). A dequeuing thread first inserts a new dummy request at the end of the list (lines 50-57) and behaves according to the status of the previous last request

```
1   struct node_t {
2     data_type value;
3     node_t *next;
4   };
5
6   struct request_t {
7     request_t *rnext;
8     int status; /* {COMBINE, WAIT, DONE} */
9     node_t *node; /* dequeued node */
10    bool ret; /* return code */
11  };
12
13  struct lecd_t {
14    node_t *Tail cacheline_aligned;
15    node_t *Head cacheline_aligned;
16    Lock CombinerLock;
17    request_t *ReqTail[NCLUSTER] cacheline_aligned;
18  };
19
20  void initialize(lecd_t *Q) {
21    node_t *permanent_dummy = new node_t;
22    permanent_dummy→next.ptr = null;
23    Q→Head = Q→Tail = permanent_dummy;
24    for (int i = 0; i < NCLUSTER; ++i) {
25      ReqTail[i] = new request_t;
26      ReqTail[i].status = COMBINE;
27    }
28  }
29
30  void enqueue(lebd_t *Q, node_t *node) {
31    node→next.ptr = null;
32    old_tail = SWAP(&Q→Tail, node);
33    old_tail→next.ptr = node;
34  }
35
36  bool dequeue(lecd_t *Q, node_t **pnode) {
37    bool ret;
38    request_t *prev_req;
39    prev_req = add_request(Q);
40    while(prev_req→status == WAIT) ;
41    if (prev_req→status == COMBINE)
42      combine_dequeue(Q, req);
43    *pnode = prev_req→node;
44    ret = prev_req→ret;
45    delete req;
46    return ret;
47  }
48
49  request_t *add_request(lecd_t *Q) {
50    request_t *new_req, *prev_req;
51    int cluster_id = get_current_cluster_id();
52    new_req = new request_t;
53    new_req→status = WAIT;
54    new_req→rnext = null;
55    prev_req = SWAP(&Q→ReqTail[cluster_id], new_req);
56    prev_req→rnext = new_req;
57    return prev_req;
58  }
59
60  void combine_dequeue(lecd_t *Q, request_t *start_req) {
61    request_t *req, *req_rnext;
62    request_t *last_req, *last_node_req = null;
63    node_t *node = Q→Head, *node_next;
64    int c = 0;
65    lock(CombinerLock);
66    for (req = start_req; true; req = req_rnext, ++c) {
67      do {
68        node_next = node→next;
69      } while (node_next == null && node != Q→Tail);
70      if (node_next == null)
71        req→ret = false;
72      else {
73        if (last_node_req != null)
74          last_node_req→status = DONE;
75        last_node_req = req;
76        req→node = node_next;
77        req→ret = true;
78        node = node_next;
79      }
80      req_rnext = req→rnext;
81      if (req_rnext→rnext == null || c == MAX_COMBINE) {
82        last_req = req;
83        break;
84      }
85      req→status = DONE;
86    }
87    if (last_node_req→node→next == null) {
88      Q→Head→next = null;
89      if (CAS(&Q→Tail, last_node_req→node, Q→Head))
90        goto END;
91      while (last_node_req→node→next == null) ;
92    }
93    Q→Head→next = last_node_req→node→next;
94   END:
95    if (last_node_req != last_req)
96      last_node_req→status = DONE;
97    last_req→next→status = COMBINE;
98    last_req→status = DONE;
99    unlock(CombinerLock);
100 }
```

Fig. 3. The pseudo-code of the LECD queue.

in the list (lines 40-42). The status of a request can be either `COMBINE`, `WAIT`, or `DONE`. The first request is always in the `COMBINE` state (lines 26 and 97), and the rest are initially in the `WAIT` state (line 53). After the combiner processes the requests, the status of each request is changed to `DONE` (lines 74, 85, 96 and 98). At the end of the combining operation, the status of the last dummy request is set to `COMBINE` for a subsequent dequeuer to take the role of a combiner.

Appending a new dummy request. We first insert a new dummy request at the end of the per-cluster request list: we update `ReqTail` to the new dummy request using `SWAP` (line 55) and the old `ReqTail`'s `rnext` to the new dummy request (line 56). The initial status and `rnext` of the new dummy request are set to `WAIT` and `null`, respectively (lines 53-54). Like the `enqueue` operation, since the `SWAP` always succeeds, it does not require parameter tuning. According to the status of old `ReqTail`, a thread takes the role of a combiner when the status is `COMBINE` (line 41) or waits for the completion of its request processing when the status is `WAIT` (line 40).

Combining `dequeue` operation. The execution of combiners is coordinated by the global combiner lock, `CombinerLock` (lines 65 and 99). An active combiner, which acquired the lock, processes pending requests and changes the status of each processed request to `DONE` for the waiting threads to proceed (lines 74, 85, 96 and 98). Though a higher degree of combining contributes higher combining throughput by reducing the locking overhead, there is a possibility for a thread to serve as a combiner for an unfairly long time. Especially, to prevent a combiner from traversing a continuously growing list, we limit the maximum degree of combining to `MAX_COMBINE` similarly to H-Queue [19]. In our experiments, we set `MAX_COMBINE` to three times the number of threads. After processing the requests, we update `Head`'s `next` in contrast to other queues' updating `Head`. If there are at least two nodes in the queue, `Head`'s `next` is set to the

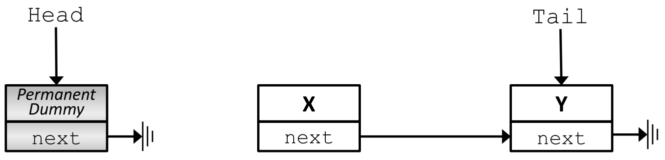

Fig. 4. A second node (Y) is enqueued over the first node (X) which is in a transient state.

next of the last dequeued node (line 93). If the last non-dummy node is dequeued and thus the queue is empty, we update Head's next to null and update Tail to Head (lines 88 and 89). Head's next is updated using a simple store instruction since there is no contention. In contrast, Tail should be updated using CAS due to contention with concurrent enqueuers. However, there is no need to retry the CAS when it fails. Since the CAS failure means that another thread enqueues a node so that the queue is not empty any more, we update Head s next to the next of the last dequeued node (line 93). Also, the LECD queue performs busy waiting to guarantee that all enqueued nodes are dequeued (lines 67-69 and 91). Finally, we set the status of the last dummy request to COMBINE for a subsequent dequeuer of the cluster to take the role of a combiner (line 97).

Our per-cluster combining approach has important advantages, even with the additional locking overhead. First, since all concurrent dequeuers update the local ReqTail, the SWAP of the ReqTail is more efficient and does not generate cross-cluster traffic. Also, since a combiner only accesses local requests in one cluster, fast local memory accesses make the combiner more efficient. Finally, notifying completion by changing the status to DONE also prevents generation of cross-cluster traffic.

3.2 Memory Management

It is important to understand why our dequeue operation updates Head's next instead of Head. As we discussed in Section 2.4, a dequeued node cannot be immediately freed using a standard memory allocator because the node is recycled as a dummy node and there are read/reclaim races. On the other hand, in the use of our *permanent dummy node*, Head invariably points to the permanent dummy node by updating Head's next instead of Head. Also, since the privatization of Tail makes it only touchable by one single thread and Head is accessed by a single threaded combiner, there is no read/reclaim races. Therefore, it is possible to efficiently implement the LECD queue even in C or C++ with no need for a dedicated memory management scheme. Moreover, our caller-allocated node enables application developers' further optimization of memory management including embedding a node into its element.

3.3 Linearizability and Progress Guarantee

As stated earlier, our enqueue operation allows another concurrent thread to enqueue over a node which is in a transient state. For example, in Fig. 4, when thread T1 is preempted after updating Tail to X and before updating old Tail's next (between lines 32 and 33), thread T2 enqueues a node Y over the X. Such independent execution among concurrent enqueuers can maximize advantages of parallel execution. However, if thread T3 tries to dequeue a node at the same moment, the dequeue operation cannot be handled before T1 completes the enqueue operation. If we assume the queue is empty in this case, it would break the linearizability of the queue: its sequential specification should allow it to assume the queue contains at least one node. Since the LECD queue performs busy waiting in this case (lines 67-69 and 91), it is linearizable. The formal proof will be described in Section 5.

Traditionally, most concurrent data structures are designed to guarantee either the progress of all concurrent threads (wait-freedom [25]) or the progress of at least one thread (lock-freedom [25]). The key mechanism to guarantee progress is that threads help each other with their operations. However, such helping could impose large overhead even in the case that a thread can complete without any help [36]. Therefore, many efforts has been made to develop more efficient and simpler algorithms by loosening progress guarantee [37] or not guaranteeing progress at all [12], [18], [19], [38]. Though the LECD queue is not non-blocking, it is starvation-free since our enqueue operation will operate in a bounded number of instructions. Moreover, since the window of blocking is extremely small (between lines 32 and 33), the blocking occurs extremely rarely, and the duration, when it occurs, is very short. We will present a formal proof of our progress guarantee in Section 5 and measure how often such waiting occurs under a high level of concurrency in Section 4.3.2. Finally, it is worth noting that several operating systems allow a thread to hint to the scheduler to avoid preemptions of that thread [39]. Though the most likely use is to block preemption while holding a spinlock, we could use it to reduce the possibility of preemption during enqueue operation.

4 Evaluation

4.1 Evaluation Environment

We evaluated the queue algorithms on a four-socket system with four 10-core 2.0 GHz Intel Xeon E7-4850 processors (Westmere-EX), where each core multiplexes two hardware threads (40 cores and 80 hardware threads in total). Each processor has a 256 KB per-core L2 cache and a 24 MB per-processor L3 cache. It forms a NUMA cluster with an 8 GB local memory and each NUMA cluster communicates through a 6.4 GT/s QPI interconnect. The machine runs 64-bit Linux Kernel 3.2.0, and all the codes were compiled by GCC 4.6.3 with the highest optimization option, -O3.

4.2 Evaluated Algorithms

We compare the LECD queues to the best performing queues reported in recent literature: Fatourou and Kallimanis' H-Queue [19], which is the best performing combining-based queue, and Morrison and Afek's LCRQ and LCRQ+H [16], which are the best performing ×86-dependent lock-free queues. We also evaluated Michael and Scott's lock-free MS queue, which is the most widely implemented [1], [2], [3], and their two-lock queue [10].

Since the performances of the LCRQ and LCRQ+H are sensitive to the ring sizes, as shown in [16], we evaluated the LCRQ and LCRQ+H in two different ring sizes, the largest size with 2^{17} nodes and the medium size with 2^{10} nodes in their experiments. Hereafter, **(L)** represents for the large ring size, and **(M)** is the medium ring size. Also, hazard pointers [7], [8], [23] were enabled for memory reclamation of rings. For the queue algorithms which require locking, CLH locks [40], [41] were used. For the H-Queue, MS queue, and the two-lock queue, we used the implementation from Fatourou and Kallimanis' benchmark framework [42]. For the LCRQ and LCRQ+H, we used the implementation provided by the authors [43]. We implemented the LECD queue in C/C++. In all the queue implementations, important data structures are aligned to the cacheline size to avoid false sharing.

Though interaction between the queue algorithms and memory management schemes are interesting, it is outside the scope of this paper. Instead, we implemented a per-thread freelist on top of the `jemalloc` [44] memory allocator. Our freelist implementation has minimal runtime overhead at the expense of the largest space overhead; it returns none of the allocated memory to the operating system and does not use atomic instructions. For the LECD queue, our benchmark code allocates and deallocates nodes to impose roughly the same amount of memory allocation overhead to the others.

To evaluate the queue algorithms in various workloads, we run the following two benchmarks under two different initial conditions, in which (1) the queue is empty and (2) the queue is not empty (256 elements are in the queue):

- *Enqueue-dequeue pairs*. Each thread alternately performs an `enqueue` and a `dequeue` operation.
- *50 percent enqueues.* Each thread randomly performs an `enqueue` or a `dequeue` operation, generating a random pattern of 50 percent `enqueue` and 50 percent `dequeue` operations.

After each queue operation, we simulate a random workload by executing a random number of dummy loop iterations up to 64. Each benchmark thread is pinned to a specific hardware thread to avoid interference from the OS scheduler. For the queue algorithms, which need parameter tuning, we manually found the best performing parameters for each benchmark configuration. For example, in case of the enqueue-dequeue pairs benchmark where a queue is initially empty, we used the following parameters; in MS queue, the lower bound and upper bound of exponential backoff scheme were set to 1 and 22, respectively; in LCRQ+H, the yielding time was set to 110 microseconds for the large ring and 200 microseconds for the medium ring.

In the rest of this section, we first investigate how effective our design decisions are in Section 4.3 and then compare the LECD queue with other queue algorithms in Section 4.4. Fig. 5 shows the throughputs of the queue algorithms, and Fig. 6 shows the average throughput of the four benchmark configurations on 80 threads. Fig. 6 shows that the LECD queue performs best, followed by LCRQ+H, H-Queue, LCRQ, MS queue, and the two-lock queue. For further analysis, we show the average CPU cycles spent in memory stalls in Fig. 7 and the number of atomic instructions executed per operation in Fig. 8. Finally, to investigate how enqueuers and dequeuers interact in the LECD queue, we compare the degree of combining of the LECD queue with that of the H-Queue in Fig. 9. Also, we compare the total enqueue time and total dequeue time of our queues in Fig. 10. Though we did not show the results of all benchmark configurations, the omitted results also showed similar trends. We measured one million pairs of queue operations and reported the average of ten runs.

4.3 Experimental Results of the LECD Queue

4.3.1 How Critical Is It to Exploit `SWAP` Primitive?

To verify the necessity of `SWAP` for achieving high scalability, we simulated `SWAP` instructions of the LECD queue using `CAS` retry loops in a lock-free manner. In our experimental results, it is denoted as the LECD-CAS queue. As Fig. 7 shows, in 80 threads, the CPU stall cycles for memory are increased by 5.6 times, causing significant performance reduction, as shown in Figs. 5 and 6; on average, performance is degraded by 4.3 times in 80 threads.

4.3.2 Is Our Optimistic Assumption Valid?

As discussed in Section 3.3, we designed the LECD queue based on the optimistic assumption; though an enqueuer could block the progress of concurrent dequeuers, such blocking will hardly occur and the period will be extremely short, when it occurs, since the blocking window is extremely small. To verify whether our assumption is valid in practice, we run our benchmarks in 80 threads and measured the number of blocking occurrences for one million operations and its percentage in CPU time. Excluding the benchmark threads, about 500 threads were running in our test machine. As we expected, Table 1 shows that the blocking is very limited even at a high level of concurrency. Especially, when queues are not empty, we did not observe blocking. Even when queues are empty, the performance impact of such blocking is negligible.

4.3.3 Effectiveness of the Per-Cluster Combining

To verify how our per-cluster `dequeue` combining is effective, we evaluate the performance of the LECD queue that performs single global combining with no per-cluster optimization. We denote its results as the LECD-NO-CLUSTER queue in our experimental results. Since there is only one combining thread in the single global combining, additional locking used in the per-cluster combining is not required. However, since the combiner processes requests from arbitrary clusters, it generates cross-cluster traffic and thus degrades performance. Figs. 6 and 7 confirm this; on average, 3.2-fold the number of stall cycles cause a 2.9-fold performance degradation in 80 thread.

4.4 Comparison with Other Algorithms

4.4.1 The Two-Lock Queue

In all benchmark configurations, the throughput of the two-lock queue is the lowest (Figs. 5 and 6). Though the two-lock queue executes only one atomic instruction per operation, which is the smallest, regardless of the level of concurrency (Fig. 8), and its CPU stall cycles for memory are lower than that of the MS queue, the serialized execution results in the slowest performance.

4.4.2 The MS Queue

In all lock-free queues, the parameter tuned versions are significantly faster. The results of the MS queues show how critical the parameter tuning is. As Figs. 7 and 8 show, the

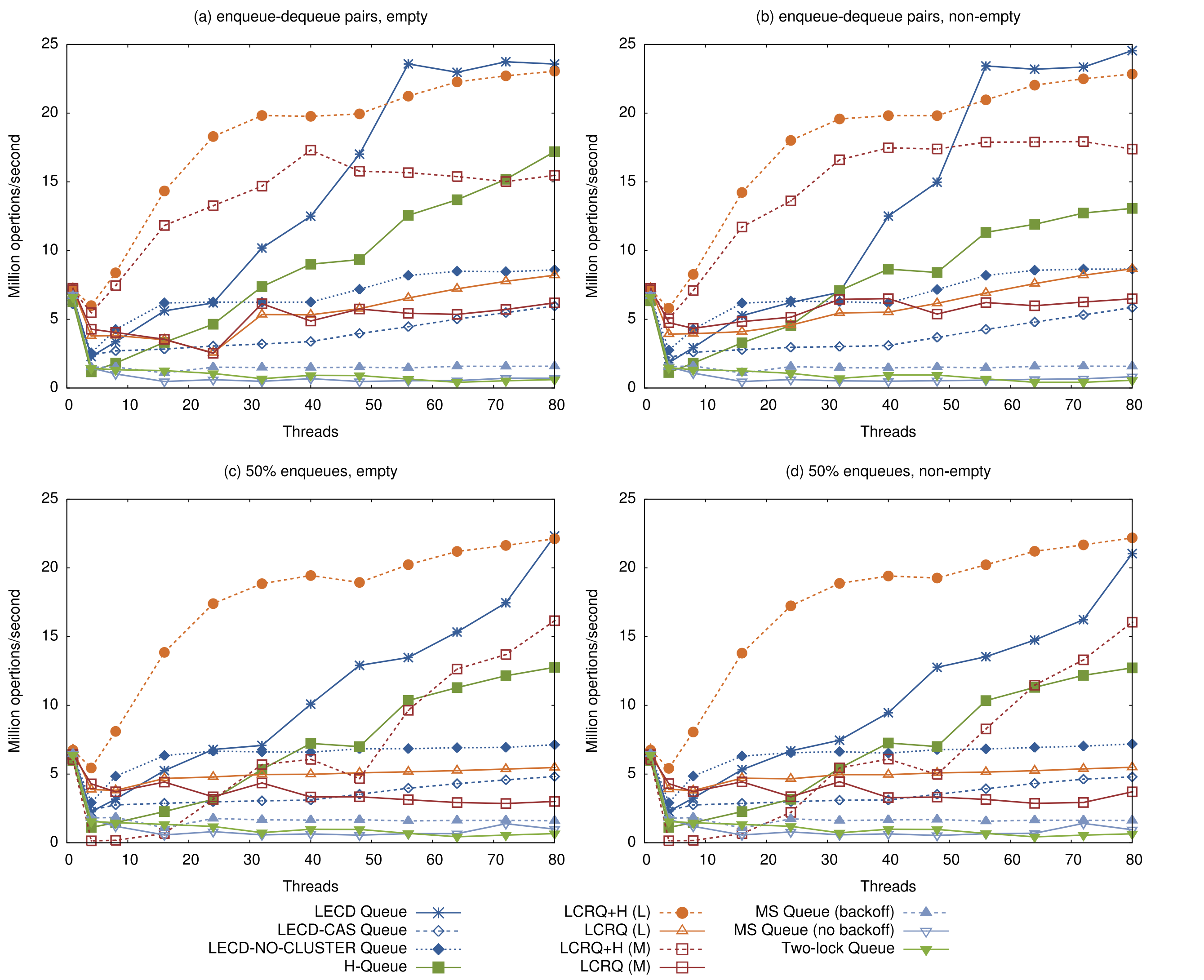

Fig. 5. Throughputs of the queue algorithms for the benchmark configurations. The LECD-CAS queue is the LECD queue in which `SWAP` is simulated using `CAS` in a lock-free way. The LECD-NO-CLUSTER queue is the LECD queue which performs not per-cluster combining but global combining using a single request list. For LCRQ and LCRQ+H, suffix **(L)** denotes that a queue is configured to use the large ring with 2^{17} nodes, and suffix **(M)** denotes that a queue is configured to use the medium ring with 2^{10} nodes.

backoff scheme reduces CPU stall cycles for memory and atomic instructions per operation by 4.2 times and 1.7 times, respectively. As a result, the MS queue with the backoff scheme has a 1.8 times greater performance than one without it. Even using the backoff scheme, since `CAS` retry loops in the MS queue make `Head` and `Tail` contended hot spots, the stall cycles of the MS queue is 16 times larger than that of the LECD queue, and the LECD queue outperforms it by 14.3 times.

4.4.3 The LCRQ and LCRQ+H

The hierarchical optimization of the LCRQ can significantly improve the performance. In LCRQ+H, each thread first checks the global cluster ID. If the global cluster ID is not the

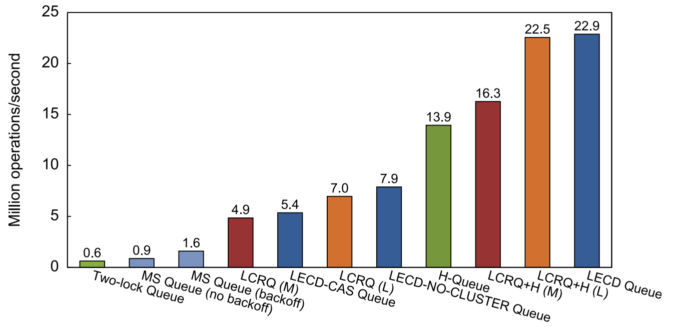

Fig. 6. Average throughputs of the four benchmark configurations on 80 threads.

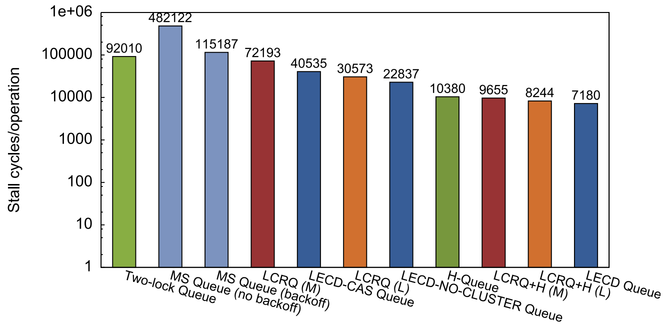

Fig. 7. Average CPU cycles spent in memory stalls of the four benchmark configurations on 80 threads. Y-axis is in log scale.

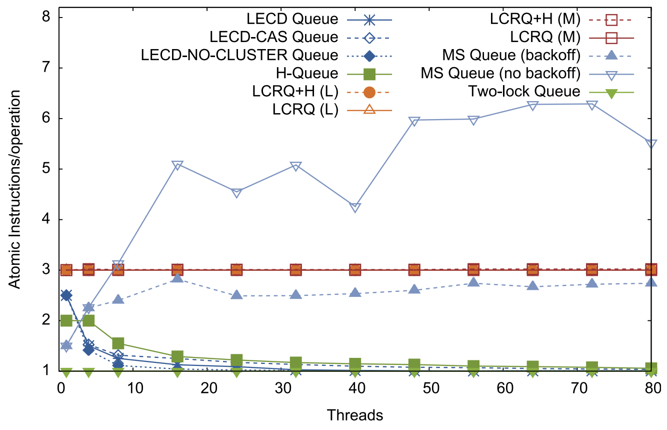

Fig. 8. Number of atomic instructions per operation in the enqueue-dequeue pairs benchmark with an initial empty state.

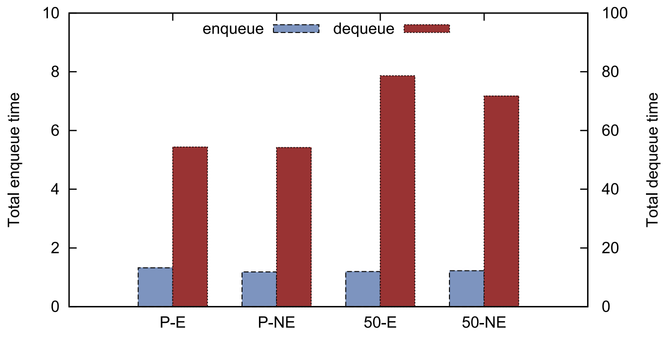

Fig. 10. Total enqueue and dequeue times of the LECD queue on 80 threads. **P** and **50** represent benchmark types of enqueue-dequeue pairs and 50 percent enqueues benchmarks, respectively. **E** and **NE** represent the empty and non-empty initial queue conditions, respectively. The scale of y 2-axis is 10-fold greater than that of y 1-axis.

same as the running cluster ID of the thread, the thread waits for a while (i.e., voluntarily yielding), and then updates the global cluster ID to its cluster ID, using CAS, and executes the LCRQ algorithm. Since it clusters operations from the same cluster for a short period, this process can reduce cross-cluster traffic and improve performance.

In our experimental results, LCRQ+H outperforms LCRQ and the queues with the larger ring are faster. Although the hierarchical optimization and the smaller ring size slightly increase the number of atomic instruction per operation (by up to 7 percent in Fig. 8), we observed that the CPU stall cycles for memory more directly affects performance than the increased atomic instructions (Fig. 7). When a ring is the medium size (M), the whole of the ring fits in a per-core L2 cache. That increases the stall cycles by up to 2.4 times since updating an L2 cacheline is very likely to trigger invalidating shared cachelines in other cores. As a result, performances of LCRQ (M) and LCRQ+H (M) are about 40 percent slower than those of LCRQ (L) and LCRQ+H (L). The hierarchical optimization decreases the stall cycles by 7.5 times and thus improves performance by 3.3 times. However, as Fig. 5 shows, in addition to the drawback of manually tuning the yielding time, the LCRQ+H shows great variance in performance according to the benchmark. In the enqueue-dequeue pairs benchmark, LCRQ+H outperforms LCRQ in all levels of concurrency. But, in the 50 percent enqueues benchmark, LCRQ+H (M) shows performance collapse at a low level of concurrency. If there are many concurrent operations in the currently scheduled cluster, the voluntary yielding helps improve performance by aggregating operations in the same cluster and reducing cross-cluster traffic. Otherwise, it is simply a waste of time. That is why LCRQ+H (M) shows performance collapse at a low level of concurrency. LCRQ+H (M) shows the lowest throughput, 0.17 Mops/sec, in eight threads of the 50 percent enqueues benchmark, and 98.7 percent of CPU time is spent on just yielding. This shows that the best performing yielding time is affected by many factors, such as arrival rate of requests, cache behavior, cost of contention, level of contention, and so on. The LECD queue outperforms LCRQ (L) and LCRQ (M) by 3.3 times and 4.7 times, respectively. Unlike LCRQ+H, LECD queue shows little variance in performance according to the benchmark. Also, in some benchmark configurations, it shows similar or better performance to LCRQ+H without manual parameter tuning.

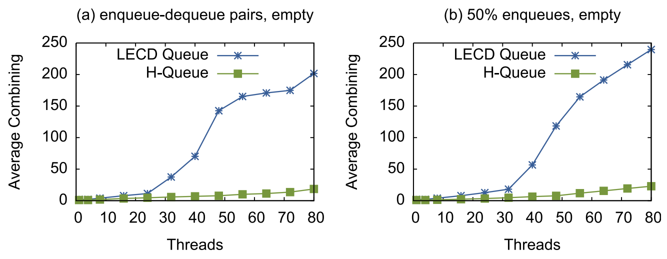

Fig. 9. Average degree of combining per operation.

4.4.4 *The H-Queue*

The LECD outperforms the H-Queue in all cases, and in 80 threads, the average throughput of the LECD queue is 64 percent higher than that of the H-Queue. In the LECD queue, enqueuers can be benefited by parallel execution since only the SWAP operation is serialized by H/W. So, as Fig. 10 shows, enqueuers consume time in only $1/53$ dequeuers. Thus, most threads wait for the completion of dequeue operation by the combiner, as confirmed by the high combining degree in Fig. 9: in 80 threads, the average degree of combining in the LECD queue is ten times higher than that of the H-Queue. The high combining degree eventually reduces the number of lock operations among per-cluster combiners. So, the reduced locking overhead contributes to reduce the number of atomic instructions per operation in Fig. 8, and thus finally contributes to achieve higher performance.

5 Correctness

Our model of multi-threaded computation follows the linearizability model [45]. We treat basic load, store, SWAP, and CAS operations as atomic actions and thus can use the standard approach of viewing them as if they occurred sequentially [46]. We prove that the LECD queue is linearizable to the abstract sequential FIFO queue [47] and it is starvation-free.

5.1 Linearizability of the LECD Queue

Definition 1. *The linearization points for each of the queue operations are as follows:*

TABLE 1
Performance Impact of Preemption in the Middle of `enqueue` Operation

Benchmark	# of blocking/Mops	% of the blocking time
enq-deq pairs, empty	0.53	$1.12 * 10^{-5}\%$
enq-deq pairs, non-empty	0.0	0.0%
50%-enqs, empty	0.07	$6.25 * 10^{-8}\%$
50%-enqs, non-empty	0.0	0.0%

- *`Enqueue` operation is linearized in line 32.*
- *Unsuccessful `dequeue` operation is linearized at the execution of its request in line 71 by the active combiner.*
- *Successful `dequeue` operation is linearized at the execution of its request in line 77 by the active combiner.*

Definition 2. *After executing the `SWAP` in line 32 and before executing line 33, the `node` and `old_tail` are considered to be* involved in the middle of the `enqueue` operation*.*

Definition 3. *When the status of a thread is `COMBINE` so that the thread can execute the `combine_dequeue` method (lines 61-99), the thread is called a* `dequeue` combiner*, or simply a* combiner*.*

Definition 4. *An* active `dequeue` combiner*, or simply an* active combiner*, is a combiner which holds `CombinerLock` in line 65.*

Definition 5. Combined `dequeue` requests*, or simply* combined requests*, are an ordered set of requests processed in lines 66-86 for a single invocation of a `combine_dequeue` method.*

Lemma 1. *`Head` always points to the first node in the linked list.*

Proof. `Head` always points to the initially allocated permanent dummy node and is never updated after initialization. □

Lemma 2. *The `enqueue` operation returns in a bounded number of steps.*

Proof. Since the `enqueue` operation has no loop, it returns in a bounded number of steps. □

Lemma 3. *Nodes are ordered according to the order of the linearization points of the `enqueue` operations.*

Proof. The `SWAP` in line 32 guarantees that `Tail` is atomically updated to a new `node` and only one thread can access its old value, `old_tail`. The privately owned `old_tail`'s `next` is set to the new `node`. Therefore, nodes are totally ordered by the execution order of line 32. □

Lemma 4. *All linearization points are in between their method invocations and their returns.*

Proof. It is obvious for the `enqueue` operation and the `dequeue` operation, where a thread is a combiner. For the `dequeue` operation, where a thread is not a combiner, since the thread waits until the status of its request becomes `DONE` (line 40) and the combiner sets the status of a processed request to `DONE` after passing the linearization point of each request (lines 74, 85, 96 and 98), its linearization point is in between the invocation and the return. □

Lemma 5. *`Tail` always points to the last node in the linked list.*

Proof. In the initial state, `Tail` points to the permanent dummy node, which is the only one node in the list (line 23). In enqueuing nodes, the `SWAP` in line 32 atomically updates `Tail` to a new `node`. When the queue becomes empty, a successful `CAS` in line 89 updates `Tail` to `Head`, which always points to the dummy node by Lemma 1. □

Lemma 6. *An active dequeue combiner deletes nodes from the second node in the list in the order of `enqueue` operations.*

Proof. An active combiner starts node traversal from the second node (i.e., `Head`'s `next`) until all pending requests are processed or the number of processed requests is equal to `MAX_COMBINE` (lines 66-86). Since the combiner traverses via `next` (lines 68 and 78), by Lemma 3, the traverse order is the same as that of `enqueue` operations. If a node in the current iteration is the same as `Tail` (lines 69-70), the queue becomes empty, so `ret` of the request is set to `false` (line 71). Otherwise, the dequeued node is returned with setting `ret` to `true` (lines 72-79). After the iteration, if the queue becomes empty, the active combiner updates the pointer to the second node to `null` in line 88 and tries to move `Tail` to `Head` using `CAS` in line 89. If the `CAS` fails (i.e., another node is enqueued after the iteration), the completion of enqueuing the node waits in line 91. By Lemma 2, the completion of the `enqueue` operation is guaranteed. If the queue does not become empty, the pointer of the second node is updated to the `next` of the last dequeued node in line 93. In all cases, the update of the `Head`'s `next` is protected by `CombinerLock`. □

Lemma 7. *An enqueued node is dequeued after the same number of `dequeue` operations as that of `enqueue` operations is performed.*

Proof. From Definition 2 and Lemmas 1 and 6, when the second node in the list is in the middle of the `enqueue` operation, the queue is not empty but the `Head`'s `next` is `null`. Since the `dequeue` operation waits until the `enqueue` operation completes (lines 67-69), the nodes in the middle of the `enqueue` operation are never dequeued, and an enqueued node is dequeued after the same number of `dequeue` operations as that of `enqueue` operations is performed. □

Theorem 1. *The LECD queue is linearizable to a sequential FIFO queue.*

Proof. From Lemmas 3, 6 and 7, the order of the `enqueue` operations is identical to the order of its `dequeue` operations, so the queue is linearizable. From Lemma 5, `Tail` is guaranteed to be set to `Head` when the queue is empty, so the queue is also linearizable with respect to unsuccessful `dequeue` operations. □

5.2 Liveness of the LECD Queue

Lemma 8. *Adding a new request returns in a bounded number of steps.*

Proof. Since the `add_request` method (lines 50-57) has no loop, it returns in a bounded number of steps. □

Lemma 9. *If a thread first adds a request to the per-cluster request list after the combining operation, the thread becomes a combiner for the cluster.*

Proof. The decision whether or not to take the role of combiner is determined by the status of the previous request (lines 39-41, 55-57). After the initialization, since the status of the first request for each cluster is set to `COMBINE` in line 26, the first dequeuer for each cluster becomes a combiner. Since the status of the next request of combined requests is set to `COMBINE` in line 97 before an active combiner completes its operation, a thread that adds a request for the cluster just after the combining becomes a new combiner. □

Lemma 10. *The combining* `dequeue` *operation (*`combine_-dequeue` *method) is starvation-free.*

Proof. Assuming the `CombinerLock` implementation is starvation-free, acquiring the `CombinerLock` in the active combiner is also starvation-free. In the combining operation, the number of combined requests is limited to `MAX_COMBINE` and the other two loops wait for the completion of the `enqueue` operation, which is wait-free by Lemma 2. By Lemma 9, after the combining operation, the subsequent dequeuer is guaranteed to become the next combiner. Therefore, the combining operation is starvation-free. □

Theorem 2. *The LECD queue is starvation-free.*

Proof. By Lemma 2, the `enqueue` operation is wait-free. From Lemmas 4, 8, and 10, the `dequeue` operation is starvation-free. Therefore, the LECD queue is starvation-free. □

6 Conclusion and Future Directions

We have presented the LECD queue, which is a linearizable concurrent FIFO queue. Neither parameter tuning nor a dedicated memory management scheme is needed. The advantage of these features is that they can be used for libraries in which parameter tuning for particular combinations of workloads and machines are infeasible and more flexible memory management for application developers' further optimization is needed. The key to our design in the LECD queue is to synergistically use lock-free and combining approaches. As our experimental results show, in the 80-thread average throughput, the LECD queue outperforms the MS queue, whose backoff parameters are tuned in advance, by 14.3 times; the H-Queue, which is the best performing combining-base queue, by 1.6 times; and the LCRQ+H, whose design is ×86-dependent and whose parameter is tuned in advance, by 1.7 percent.

Our lessons learned have implications for future directions. First of all, we expect that integrating lock-free approaches and combining approaches could be an alternative way in designing concurrent data structures. As the core count increases, `CAS` failure on hot spots is more likely to happen so the architecture-level support of atomic primitives, such as `SWAP`, that always succeed is critical. Also, though we showed the effectiveness of the LECD queue, the combiner will become a performance bottleneck at the very high concurrency level (e.g., hundreds of cores). To resolve this, we have a plan to extend the LECD queue to have parallel combiner threads.

Acknowledgments

This work was supported by the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform]. This research was supported by the Ministry of Science, ICT & Future Planning (MSIP), Korea, under the Information Technology Research Center (ITRC) support program (NIPA-2014 (H0301-14-1020)) supervised by the National IT Industry Promotion Agency (NIPA). Young Ik Eom is the corresponding author of this paper.

References

[1] D. Lea. JSR 166: Concurrency utilities. (2014) [Online]. Available: http://gee.cs.oswego.edu/dl/jsr166/dist/docs/
[2] Boost C++ Libraries. boost::lockfree::queue. (2014) [Online]. Available: http://www.boost.org/doc/libs/1_54_0/doc/html/boost/lockfree/queue.html
[3] liblfds.org. r6.1.1:lfds611_queue. (2014) [Online]. Available: http://www.liblfds.org/mediawiki/index.php?title=r6.1.1:Lfds61_queue
[4] C. Min and Y. I. Eom, "DANBI: Dynamic scheduling of irregular stream programs for many-core systems," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, 2013, pp. 189–200.
[5] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *Proc. Int. Conf. Parallel Archit. Compilation Tech.*, 2011, pp. 22–32.
[6] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing," *SIGOPS Oper. Syst. Rev.*, vol. 46, vol. 3, pp. 65–81, Dec. 2012.
[7] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, vol. 6, pp. 491–504, Jun. 2004.
[8] M. Herlihy, V. Luchangco, and M. Moir, "The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures," Sun Microsyst., Inc., Mountain View, CA, USA, Tech. Rep. SMLI TR-2002-112, 2002.
[9] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, vol. 12, pp. 1270–1285, Dec. 2007.
[10] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th Annu. ACM Symp. Principles Distrib. Comput.*, 1996, pp. 267–275.
[11] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free FIFO queues," *Distrib. Comput.*, vol. 20, no. 5, pp. 323–341, 2008.
[12] C. Min, H. K. Jun, W. T. Kim, and Y. I. Eom, "Scalable cache-optimized concurrent FIFO queue for multicore architectures," *IEICE Trans. Inf. Syst.*, vol. E95-D, no. 12, pp. 2956–2957, 2012.
[13] 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.
[14] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Proc. 11th Int. Conf. Principles Distrib. Syst.*, 2007, pp. 401–414.

[15] A. Agarwal and M. Cherian, “Adaptive backoff synchronization techniques,” in *Proc. 16th Annu. Int. Symp. Comput. Archit.*, 1989, pp. 396–406.
[16] A. Morrison and Y. Afek, “Fast concurrent queues for x86 processors,” in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 103–112.
[17] Y. Oyama, K. Taura, and A. Yonezawa, “Executing parallel programs with synchronization bottlenecks efficiently,” in *Proc. Int. Workshop Parallel Distrib. Comput. Symbolic Irregular Appl.*, 1999, pp. 182–204.
[18] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” in *Proc. 22nd ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 355–364.
[19] P. Fatourou and N. D. Kallimanis, “Revisiting the combining synchronization technique,” in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 257–266.
[20] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Scalable flat-combining based synchronous queues,” in *Proc. 24th Int. Conf. Distributed Comput.*, 2010, pp. 79–93.
[21] M. S. Moir, V. Luchangco, and M. Herlihy, “Single-word lock-free reference counting,” U.S. Patent 7299242, Nov. 2007.
[22] D. L. Detlefs, P. A. Martin, M. S. Moir, and G. L. Steele, JR., “Lock free reference counting,” U.S. Patent 6993770, Jan. 2006.
[23] M. M. Michael, “Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation,” U.S. Patent 2004/0107227 A1, Jun. 2004.
[24] R. K. Treiber, “Systems programming: Coping with parallelism,” IBM Almaden Res. Center, San Jose, CA, Tech. Rep. RJ-5118, 1986.
[25] M. Herlihy, “Wait-free synchronization,” *ACM Trans. Program. Lang. Syst.*, vol. 13, vol. 1, pp. 124–149, Jan. 1991.
[26] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, “Distributing hot-spot addressing in large-scale multiprocessors,” *IEEE Trans. Comput.*, vol. TC-36, no. 4, pp. 388–395, Apr. 1987.
[27] J. M. Mellor-Crummey, “Concurrent queues: Practical fetch-and-Φ algorithms,” Tech. Rep. 229, Comput. Sci. Dept., Univ. Rochester, Rochester, NY, USA, Nov. 1987.
[28] D. Vyukov. Intrusive MPSC node-based queue. (2014) [Online]. Available: http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue
[29] M. Desnoyers and L. Jiangshan. LTTng Project: Userspace RCU. (2014) [Online]. Available: http://lttng.org/urcu
[30] P. McKenney and J. Slingwine, “Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring,” US Patent 5442758, Aug. 15, 1995.
[31] P. E. McKenney and J. D. Slingwine, “Read-copy update: Using execution history to solve concurrency problems,” in *Proc. Int. Conf. Parallel Distrib. Comput. Syst.*, 1998, pp. 509–518.
[32] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma, “Using read-copy-update techniques for system V IPC in the Linux 2.5 Kernel,” in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, 2003, pp. 297–310.
[33] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, “User-level implementations of read-copy update,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, vol. 2, pp. 375–382, Feb. 2012.
[34] K. Fraser, “Practical lock-freedom,” Ph.D. dissertation, Comput. Laboratory, Cambridge Univ., Cambridge, U.K., 2004.
[35] D. Hendler, N. Shavit, and L. Yerushalmi, “A scalable lock-free stack algorithm,” in *Proc. 16th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2004, pp. 206–215.
[36] A. Kogan and E. Petrank, “A methodology for creating fast wait-free data structures,” in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 141–150.
[37] M. Herlihy, V. Luchangco, and M. Moir, “Obstruction-free synchronization: Double-ended queues as an example,” in *Proc. 23rd Int. Conf. Distrib. Comput. Systems*, 2003, pp. 522–529.
[38] M. Desnoyers, “Proving the correctness of nonblocking data structures,” *Queue*, vol. 11, vol. 5, pp. 30–43, May 2013.
[39] Oracle. Man pages section 3: Basic library functions: Schedctl_start(3C). (2014) [Online]. Available: http://docs.oracle.com/cd/E19253-01/816-5168/6mbb3hrpt/index.html
[40] T. Craig, “Building FIFO and priority-queuing spin locks from atomic swap,” Dept. Comput. Sci., Univ. Washington, Seattle, WA, USA, Tech. Rep. 93-02-02, 1993.
[41] P. S. Magnusson, A. Landin, and E. Hagersten, “Queue locks on cache coherent multiprocessors,” in *Proc. 8th Int. Symp. Parallel Process.*, 1994, pp. 165–171.
[42] N. D. Kallimanis. Sim: A highly-efficient wait-free universal construction. (2014) [Online]. Available: https://code.google.com/p/sim-universal-construction/
[43] LCRQ source code package. (2014) [Online]. Available: http://mcg.cs.tau.ac.il/projects/lcrq/lcrq-101013.zip
[44] J. Evans. Scalable memory allocation using jemalloc. (2014) [Online]. Available: https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919
[45] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, vol. 3, pp. 463–492, Jul. 1990.
[46] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, “Atomic snapshots of shared memory,” *J. ACM*, vol. 40, vol. 4, pp. 873–890, Sep. 1993.
[47] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to Algorithms.* Cambridge, MA, USA: MIT Press, 2001.

Changwoo Min received the BS and MS degrees in computer science from Soongsil University, Korea, in 1996 and 1998, respectively, and the PhD degree from the College of Information and Communication Engineering, Sungkyunkwan University, Korea, in 2014. From 1998 to 2005, he was a research engineer in Ubiquitous Computing Lab (UCL) of IBM Korea. Since 2005, he has been a research engineer at Samsung Electronics. His research interests include parallel and distributed systems, storage systems, and operating systems.

Young Ik Eom received the BS, MS, and PhD degrees from the Department of Computer Science and Statistics, Seoul National University in Korea, in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an associate professor at Dankook University, Korea. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine from September 2000 to August 2001. Since 1993, he has been a professor at Sungkyunkwan University, Korea. His research interests include parallel and distributed systems, storage systems, virtualization, and cloud systems.