LCD: Local Combining on Demand*

Dana Drachsler-Cohen and Erez Petrank

Computer Science Department, Technion, Israel
{ddana,erez}@cs.technion.ac.il

Abstract. Combining methods are highly effective for implementing concurrent queues and stacks. These data structures induce a heavy competition on one or two contention points. However, it was not known whether combining methods could be made effective for parallel scalable data structures that do not have a small number of contention points. In this paper, we introduce *local combining on-demand*, a new combining method for highly parallel data structures. The main idea is to apply combining locally for resources on which threads contend. We demonstrate the use of local combining on-demand on the common linked-list data structure. Measurements show that the obtained linked-list induces a low overhead when contention is low and outperforms other known implementations by up to 40% when contention is high.

Keywords: Concurrent Data-Structures, Multiprocessors, Synchronization.

1 Introduction

In the era of multi-core architectures, there is a growing need for scalable concurrent data structures, which are fundamental building-blocks in a wide range of algorithms. A common approach to design scalable concurrent algorithms is to let each thread execute as independently as possible of other threads, while making its own progress as fast as possible. This approach is often highly effective, especially when contention is low. When resources become contended it is beneficial to consider *combining* techniques in which threads help each other to complete operations.

Combining techniques entail overhead and thus one may expect to integrate them only in highly-contended data structures. Much work follows this guideline and focuses on techniques designated for data structures which have few contention points, and thus are contention-prone [1,5,6,7,12,13,18].

We present *local combining on-demand* (*LCD*), a combining technique for data structures with unbounded number of contention points. We show combining contributes even to such data structures for which contention occurs for short periods.

For such data structures, applying one of the general combining techniques (e.g., [7,12,13]) often results in a high overhead. General techniques apply combining globally and involve all threads accessing the data structure. *LCD* is applied *locally*, namely only in contended sections, and does not introduce any overhead to threads accessing other sections. This is achieved by applying *LCD* independently for each resource. This approach is beneficial for data structures with an unbounded number of contention points which typically observe contention on small sections.

* This work was supported by the Israeli Science Foundation grant No. 275/14.

M.K. Aguilera et al. (Eds.): OPODIS 2014, LNCS 8878, pp. 355–371, 2014.

In addition, *LCD* applies combining *on-demand* when contention is observed and it applies the proper combining routine. Examples for combining routines are executing operations on behalf of other threads, eliminating complementary operations without affecting the data structure, and notifying threads waiting to lock a resource which has become irrelevant to their operations due to concurrent updates. Thereby, *LCD* reduces the number of accesses to the data structure and the overall waiting time of threads.

We demonstrate the *LCD* methodology by incorporating it into a fundamental data structure, the *linked-list*, for which various concurrent implementations were suggested (e.g., [8,10,11,20]). The linked-list is a simple data structure that enables us to present the main challenges and solutions of an *LCD* design and evaluating its efficiency on standard workloads. We introduce the *LCD-list* which is an extension of the *lazy-list*. The lazy-list [11] is a lock-based implementation which is arguably the most efficient and scalable implementation of the linked-list on most workloads. We implemented the *LCD*-list in Java and we show it improves the lazy-list performance. While the application of *LCD* to data structures whose operations acquire multiple locks is not trivial, we believe it can be beneficial for other concurrent data structures as well.

We consider one implementation choice as an additional contribution of this work. We integrated *LCD* into the *Java reentrant lock*. A thread acquiring this lock first attempts to obtain it using a single CAS operation. If the attempt succeeds, it usually implies that contention is low. If this attempt fails, the thread is added to a waiting queue. *LCD* is triggered only for threads added to this queue, and thus redundant combining overhead is avoided. Furthermore, *LCD* leverages the lock queue to detect the operations to combine. Utilizing one queue for serving threads waiting for the lock and detecting threads for combining reduces space and maintenance overhead. Measurements show that performance is improved when integrating *LCD* within the Java lock.

The main contributions of this paper are:

- A novel combining methodology adequate for data structures with an unbounded number of contention points which is triggered locally on contended sections only.
- An application of the methodology to the linked-list data structure.
- Implementation and evaluation of the *LCD* methodology and its integration into the Java lock. Results show that they perform well especially when contention is high and introduce negligible overhead under little or no contention.
- Implementation and evaluation of the *LCD*-list. Results show that it outperforms the lazy-list and other linked-list implementations on most workloads.

Related Work. The technique of combining operations first appears in *combining trees* [21]. In combining trees each hot-spot is associated with a tree whose leaves are pre-assigned to threads. Threads traverse upwards in the tree to gain exclusive access to the hot-spot which is assigned to threads that reached the root. If during the traversal two threads access concurrently the same tree node, one thread collects the other's operations, and the other thread ceases its traversal and waits until its operations are completed. Several enhancements of this technique have been presented, such as adaptive combining tree [18], barriers implementations [9,16], and counting networks [19].

A different approach lets threads waiting to acquire a global lock to append their operation details to a list of requests [17]. This list is collected and executed by threads that acquire the lock. The *flat combining* technique [12] enhances this method by eliminating

the hot spot caused by threads contending on appending requests to the list. Flat combining was implemented for stacks, queues, and priority queues and showed excellent performance. Later work showed that its application to skip-lists did not improve performance [3]. Various extensions were suggested to improve performance. Hendler et al. [13] extend the method to support multiple locks and delegation of requests from one thread to another. Fatourou and Kallimanis [7] used one queue to implement the lock and maintain the request list for combining. These approaches were evaluated for data structures with a small number of contention points, whereas *LCD* is effective (and was evaluated) for data structures with an unbounded number of contention points. Flat combining was also extended for skip-lists by allowing combiners to access non-intersected sections concurrently [3]. In this technique, combiners are pre-assigned to static sections on which they operate exclusively. In contrast, our approach reduces overhead by triggering combining locally and dynamically only when required.

In addition, *LCD* differs from previous work in its integration of the combining structure into the Java lock. Previous work either maintained designated data structures for combining (e.g., [12]) or presented new locks supporting combining (e.g., [5,7]).

Paper Organization. The rest of the paper is organized as follows. Section 2 provides background. Section 3 overviews the *LCD* methodology and its application to the linked-list. Section 4 describes the algorithm details. Section 5 discusses the *LCD*-list correctness. Section 6 reports performance evaluation, and Section 7 concludes.

2 Background

Here, we provide the background on the lazy-list algorithm [11], which we extend in this work, and the Java reentrant lock, in which we embed our combining technique.

2.1 The Lazy-List

The lazy-list algorithm is a concurrent sorted linked-list implementation. It consists of nodes, each storing: (i) a data object and its unique key, (ii) a lock, (iii) a *marked* flag, and (iv) a pointer to the next node in the list. The *marked* flag signifies whether the node has been *logically* removed from the list, even if it has not been *physically* unlinked from it. For simplicity, the data object is ignored in the sequel.

Throughout the execution, the list contains two sentinel nodes, denoted by *head* and *tail*, where *head*'s key is $-\infty$ and *tail*'s key is ∞. Initially, *head*'s next node is *tail*.

The list supports three operations:

- `insert(k)` – inserts k if it is not in the list; returns `true` if it inserted k, and `false` if not.
- `remove(k)` – removes k if it is in the list; returns `true` if it removed k, and `false` if not.
- `contains(k)` – checks if k is in the list; returns `true` if so, and `false` otherwise.

Algorithm 1. Search(n, k)

```
1 prev = n
2 curr = prev.next
3 while curr.key < k do
4       prev = curr; curr = curr.next
5 return prev, curr
```

Algorithm 2. Contains(k)

```
1 prev, curr = search(head, k)
2 if curr.marked then return false
3 return curr.key == k
```

All operations begin with a traversal along the list to find the correct location for the operation invocation (Alg. 1). This location is captured by two consecutive nodes, *prev* and *curr*, such that k is greater than *prev*'s key and smaller or equal to *curr*'s key. After the correct location is reached, the operations proceed differently.

Algorithm 3. Insert(k)

```
1  prev = head
2  while true do
3      prev, curr = search(prev, k)
4      prev.lock()
5      next = prev.next
6      if !prev.marked && next.key ≥ k then
7          if next.key == k then
8              prev.unlock()
9              return false
10         new = Node(k)
11         new.next = next
12         prev.next = new
13         prev.unlock()
14         return true
15     prev.unlock()
16     if prev.marked then prev = head
```

Algorithm 4. Remove(k)

```
1  prev = head
2  while true do
3      prev, curr = search(prev, k)
4      prev.lock()
5      next = prev.next
6      if !prev.marked && next.key ≥ k then
7          if next.key != k then
8              prev.unlock()
9              return false
10         next.lock()
11         next.marked = true
12         prev.next = next.next
13         next.unlock(); prev.unlock()
14         return true
15     prev.unlock()
16     if prev.marked then prev = head
```

In `contains(k)` (Alg. 2), *curr*'s *marked* is examined and if it is `true`, `false` is returned (since *curr* was removed). If *marked* is `false`, *curr*'s key is examined and if it is k, `true` is returned; otherwise, `false` is returned. Note no locks are acquired.

In `insert(k)` and `remove(k)` (Alg. 3 and Alg. 4) *prev* is locked, its successor (which may be different from *curr*) is stored in *next*, and then the location is checked to meet two conditions: (i) *prev*'s *marked* is `false` (it was not removed), and (ii) *next*'s key is greater or equal to k . If condition (i) fails, the lock is released and the operation restarts. If condition (ii) fails, the lock is released and the traversal for the correct location resumes from *prev*. If both conditions are met, `insert` and `remove` take place.

In `insert(k)`, *next*'s key is examined and if it is k, `false` is returned. Otherwise, a new node with key k is inserted between *prev* and *next* and `true` is returned.

In `remove(k)`, *next*'s key is examined and if it is greater than k, then k is not in the list and `false` is returned. Otherwise, *next*'s key is k and the removal begins. First, *next*'s lock is acquired, then its *marked* is set to `true` (the *logical* removal), and finally *prev*'s next node is set to *next*'s next node (the *physical* removal) and `true` is returned.

This description is a slightly optimized version of the lazy-list, in which unnecessary lock acquires and restarts were removed. In the original lazy-list, `insert` and `remove` lock both *prev* and *curr*, check that their *marked* flags are `false` and that *prev* points to *curr*. If any condition fails, the operation restarts. Here, *curr* is not locked, instead, *next* (*prev*'s successor after locking *prev*) is examined. While holding *prev*'s lock, its successor cannot be concurrently changed, not by adding nodes between *prev* and *next*, nor by removing *next*. Thus, reading *next*'s key is safe and there is no need to lock it.

The presented description also avoids restarts, namely, a new traversal from the head of the list. In the original lazy-list, a restart is triggered if any of the conditions following the lock acquisition has failed. Here, restarts occur only if condition (i) fails (i.e., *prev* is marked as removed). If only condition (ii) fails, the traversal resumes from *prev*. This is safe, since if only condition (ii) fails, then the key of *prev*'s successor is smaller than k, and thus the correct location must appear after *prev*.

2.2 The Java Reentrant Lock

The Java reentrant lock, available at Java's concurrent package, is a variant of the CLH lock [4]. For simplicity, the below description omits some details.

The Java lock is a semi-honest lock which provides fair access but allows opportunistic attempts to acquire it unfairly. To provide fair access, a queue of pending threads is maintained using a doubly-linked list. The lock's main fields are *head*, *tail*, and *owner*. The *head* and *tail* point to the queue head and tail. The *owner* points to the thread holding the lock or to `null` if the lock is free.

Algorithm 5. Lock()

```
1 if CAS(owner, null, thread) then return
2 enqueue(thread)
3 while true do
4     if thread == next(head) then
5         if CAS(owner, null, thread) then
6             head = thread
7             return
8     sleep()
```

Algorithm 6. Unlock()

```
1 owner = null
2 wake up next(head)
```

The lock supports the `lock` and `unlock` operations. The `lock` operation (Alg. 5) first attempts to acquire the lock unfairly by updating *owner* to the current thread via the `CAS` operation[1]. This may succeed only if *owner* is `null`, namely no other thread holds it. If the `CAS` operation fails, the thread is added to the end of the queue. It is then allowed to attempt acquiring the lock only when the queue head is its predecessor (i.e., the thread is the second in line). In this case, attempting to acquire the lock may fail if its predecessor has not released the lock yet, or if another thread has acquired it unfairly. If the acquisition fails, the thread yields and attempts again later. After acquiring the lock, the queue head is updated and the operation terminates.

The `unlock` operation (Alg. 6) sets *owner* to `null` and notifies the successor of the queue head it can acquire the lock.

3 Overview

In this section, we provide an overview of the local combining on-demand methodology.

LCD is executed independently for each lock, thus allowing multiple combining threads to execute concurrently (one for each lock). *LCD* is executed by threads that acquired the lock fairly, and thus waited for permission to lock in the lock queue. In *LCD*, the permission to lock is granted to the *newest* thread in the queue (and not to the oldest one, as in the Java lock). When a thread acquires the lock fairly, it becomes a *combiner* and collects the operation *requests* of threads preceding it in the queue. The combiner examines the collected requests. Requests not requiring this lock are removed (and their owners are notified), and identical or complementary requests are eliminated. Then, the remaining requests are executed. Finally, the owner threads of the combined requests are reported. Fig. 1 illustrates a flow example of *LCD* in the lazy-list.

LCD is embedded in the Java lock but requires cooperation from the data structure operations. The lock is responsible for the combining logic, e.g., picking the next combiner and collecting requests. The data structure operations are responsible for searching and updating the data structure, and reporting to the owners of combined operations.

[1] `CAS`(*field*, *old*, *new*) is an atomic operation that updates *field* to *new* if *field* stores *old*. If *field* is updated, the operation succeeded and `true` is returned; otherwise `false` is returned.

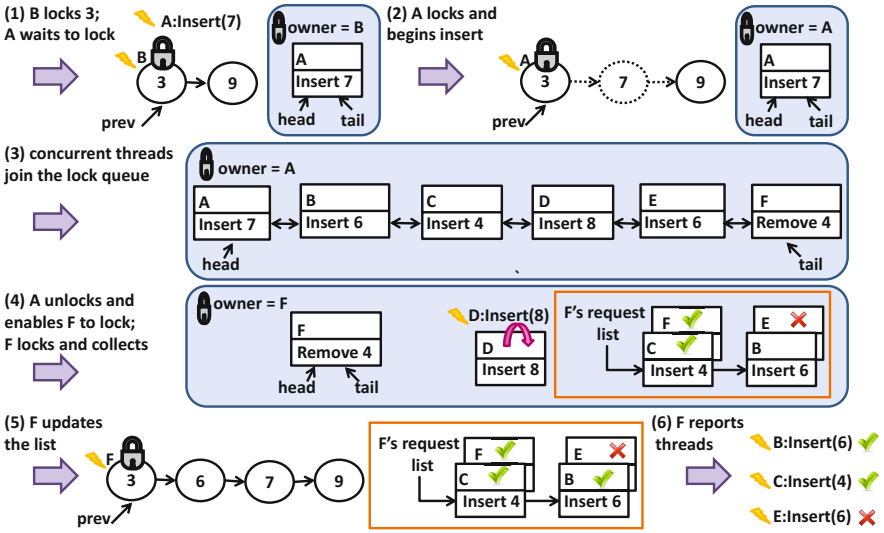


Fig. 1. *LCD* in the lazy-list. (1) A invokes `insert(7)`, attempts to lock 3, observes B has acquired it, and waits in the lock queue, (2) B unlocks and A begins to insert 7, (3) concurrent threads join the lock queue, (4) A unlocks and wakes the queue tail, F; F locks and collects requests: D is notified to search for another lock for its `insert(8)` request, C's `insert(4)` and F's `remove(4)` are grouped and marked as successful, and B's and E's `insert(6)` are grouped and E is marked as failed. (5) F inserts 6, and marks B's request as successful, and finally, (6) F reports B, C, and E, and terminates.

The *LCD* Lock. The Java lock is extended with *LCD* via `LCDlock` and `LCDunlock`. The `LCDlock` operation begins with an attempt to lock unfairly. If it fails, the thread joins the lock queue, and waits for permission to lock or for a notification that its request was combined. If it was granted permission and acquired the lock, it becomes a combiner and collects requests of threads preceding it in the lock queue. During the collection, the requests are examined. Identical requests are grouped so eventually only one may update the data structure (e.g., B's and E's requests in Fig. 1). Complementary requests are grouped and completed without affecting the data structure (e.g., C's and F's requests in Fig. 1). Threads whose requests require a different lock are notified to search again. This case may arise since threads choose a lock after observing a certain state of the data structure, however, this state may change before they acquire the lock (e.g., D's request in Fig. 1). If the combiner itself requires a different lock, it delegates the collected requests and the combiner role to another thread that requires this lock.

The `LCDunlock` operation releases the lock, and if it is invoked by a combiner, a new combiner is chosen, and it is chosen to be the last thread that joined the lock queue.

Data Structure Adaptation. The data structure operations are adapted to cope with the `LCDlock` results. If the lock was acquired unfairly, the operation proceeds without any *LCD* overhead. If the lock was acquired fairly, the operation executes the collected requests and reports to the request owners. If the request was combined, the operation terminates; and if the lock was unsuitable, the operation searches for the correct lock.

Operations acquiring several locks require a special care. *LCD* is applied independently for each lock and thus such operations need to be split into sub-operations that each require one lock for executing its updates safely. We denote such sub-operations as *single-lock operations*. Operations are split into single-lock operations as follows.

First, all possible executions are represented as series of steps of the following form:

1 . execute s_0, acquire lock l_1, and execute s_1,
2 . acquire lock l_2 and execute s_2,
...
k . acquire lock l_k, execute s_k, and release l_k,
...
2'. execute s'_2 and release l_2,
1'. execute s'_1, release l_1, and execute s'_0.

Where s_0, s'_0 are sequences of computational operations and for $i > 0$, s_i, s'_i are sequences of computational or update operations which require the lock l_i.

Next, we divide such execution into k phases, where the i^{th} phase is:
(i) execute step i,
(ii) invoke `LCDlock(`l_{i+1}`)` to initiate the next phase and upon its completion,
(iii) execute step i'.
Each phase assumes previous steps have acquired locks and completed successfully.

Finally, phases requiring the same lock are grouped into a single-lock operation, and some criteria are used to decide which phase to execute.

The benefit of single-lock operations is that different combiners may help separately and independently. Note that combiners may execute several consecutive single-lock operations provided they gained access to the required locks.

We exemplify this technique on the `remove(k)` operation of the lazy-list.

A successful removal execution requires the locks of two nodes: *prev* (the node preceding k) and *next* (the node storing k), and it can be split into the following steps:

1 . locate *prev* (s_0), lock it (l_1), verify it is not marked and that *next*'s key is k (s_1),
2 . lock *next* (l_2), set its *marked* flag to `true` (s_2), and release the lock (l_2),
1'. update *prev*'s next to *next*'s next (s'_1), release *prev*'s lock (l_1), and return `true` (s'_0).

An unsuccessful removal execution requires *prev*'s lock and consists of two steps:
1 . locate *prev*, lock it, verify it is not marked and that *next*'s key is greater than k,
1". release *prev*'s lock and return `false`.

A restart removal execution requires *prev*'s lock and consists of two steps:

1 . locate *prev*, lock it, verify it is marked or that *next*'s key is smaller than k,
1"'. release *prev*'s lock and repeat step 1.

Thus, `remove(k)` is split into two single-lock operations:

- *remove* – execute step 1. Then, if *prev* is not marked and *next*'s key is k, initiate step 2 and upon its completion execute step 1'; if *prev* is not marked and *next*'s key is greater than k, execute step 1"; else, execute step 1"'.
- *mark* – execute step 2.

This separation allows *remove* and *mark* to be executed by different threads.

4 Implementation Details

In this section we present the implementation details and provide pseudo-code. We first describe the request object, next the *LCD* lock, and finally the *LCD* linked-list.

4.1 The Request Object

Algorithm 7. Request

```
1 Operation op
2 Key key
3 Boolean result
4 Thread threadID
5 State state
6 Request head
7 Request next
8 Set combined
```

To enable *LCD*, each thread is equipped with a unique *request* object which remains throughout its execution. The request object (Alg. 7) stores all information required for the combining. `Op` and `key` contain the operation details. `Op` may be *insert*, *remove*, or *mark*. `Result` stores the operation outcome after it is completed, and initially it is set to `null`. `ThreadID` stores the owner thread ID to notify upon events, e.g., operation completion.

The request status is stored in `state` and may be either *none*, *locked*, *pending*, *completed*, or *search*. *None* is the initial state. *Locked* indicates that the request owner acquired the lock fairly and acts as a combiner (if the lock is acquired unfairly, the request fields are ignored). *Pending* indicates that a combiner has collected this request, and *completed* indicates that a combiner has executed it. *Search* signals that the request owner is waiting for a lock unsuitable for its request.

The `head` field points to the request list of the request owner. The owner is responsible for executing these requests. Initially, each thread is responsible only to its own request and thus its request list is of size one. The request list may become longer if its owner thread becomes a combiner and adds requests to its list. The request list may become empty if a combiner thread collects this request. An invariant of the execution is that uncompleted requests always belong to a single request list. As a result, each uncompleted request has at most one successor in the request list it belongs to, and this successor is stored in `next`. Thus, the request list can be implemented as a linked-list whose elements are connected via the `next` fields. When registering a new operation to a request, the `head` is set to this request and `next` is set to `null`.

The `combined` field stores requests which were eliminated or combined by this request. These requests' owners are reported after this request completes its execution.

4.2 The LCD Lock

The *LCD* lock applies combining via the `LCDlock` and `LCDunlock` operations.

The `LCDlock` (Alg. 8) attempts to acquire the lock and returns `false` if the lock was acquired unfairly and `true` otherwise (i.e., `true` indicates that local combining was initiated). `LCDlock` begins with an attempt to acquire the lock unfairly by attempting to set the `owner` to be the invoking thread. If it fails, fair locking begins by initializing the given request, registering the operation details, and adding the thread to the lock queue. An attempt to acquire the lock occurs only when the thread is the queue head's successor. If the lock is acquired, the queue head is advanced to point to the thread that acquired the lock, the request `state` is set to *locked*, and requests are collected.

If the lock is not acquired, the `state` is examined. If it is *pending*, the thread yields until `state` is updated. If `state` is *none*, the thread yields and attempts to acquire the lock later. Otherwise, if `state` is *completed*, *search*, or *locked* (in case a combiner thread delegated it the lock), `LCDlock` returns since the lock is not required anymore.

The `LCDunlock` operation (Alg. 9) releases the lock, and picks a new combiner if it is invoked by a combiner thread and there are threads waiting in the lock queue. The new combiner is chosen to be the current queue tail. To allow it acquire the lock, the queue head is set to be its predecessor in the lock queue. Requests of threads preceding the new combiner in the queue join the `reqList`, and they will collected by the new combiner after it acquires the lock.

After choosing the new combiner (if required), the lock is released by clearing the `owner` field and notifying the queue head successor.

Algorithm 8. LCDlock(req, op, key)

```
1  if CAS(owner, null, thread) then return false
2  req.state = none; res.result = null
3  req.head = req; res.next = null
4  req.op = op; req.key = key
5  enqueue(thread)
6  while true do
7      if thread == next(head) then
8          if CAS(owner, null, thread) then
9              head = thread
10             req.state = locked
11             collect(req)
12             return true
13     while req.state == pending do sleep()
14     if req.state != none then return true
15     sleep()
```

Algorithm 9. LCDunlock(req)

```
1  reqList = new List()
2  if acquired fairly && tail != head then
3      for t = next(head); t != tail; t = next(t) do
4          reqList.add(t.request)
5      head = prev(tail)
6  owner = null
7  wake next(head)
```

Collecting Requests. Combiners collect requests via the `collect` operation (Alg. 10). `Collect` begins by checking whether the combiner needs the current lock and if so, the requests will be collected to its request list (lines 2-4). If the combiner does not need this lock, another thread will be selected and requests will be added to its request list. The request whose request list is extended with the collected requests is stored in `dest`. To check whether the combiner needs this lock, its request key is compared to the key of the locked list node's successor. If the request key is smaller or equal to the successor's key, the combiner needs this lock and thus its request is stored in `dest`. Otherwise, the combiner's `state` is set to *search* (instead of *locked*) indicating that after collecting requests, it passes the lock and the combiner role to another thread, and searches for the lock required for its own request.

Next, the combiner collects the requests from the `reqList` (constructed by the `LCDunlock` operation) (lines 5-14). At each iteration, one request, denoted by r, is processed. First, its `state` is set to *pending*. Then, if `dest` is `null` and `r` requires this lock, `dest` is set to `r` (lines 7-8). Otherwise, `r`'s request list is added to `dest`'s list via `addReqs` (line 11). Then, if `r`'s `result` was not `null` before calling `addReqs`, then `r`'s owner is a combiner that completed its own request and proceeded executing other requests. Thus, after collecting its requests, `r` completes its execution (lines 12-14).

After all requests were collected, `dest` is examined. If it is the combiner's request, the operation terminates. If `dest` is `null`, the lock is released as no thread requires it (line 15). If `dest` is a different request, the lock is delegated to `dest`'s owner by updating the lock's `owner` and `dest`'s `state` (lines 16-19). The lock queue head remains unchanged since its task is to signal to `LCDunlock` the starting point of waiting threads.

Algorithm 10. Collect(req)

```
1  k = ownerNode.next.key
2  dest = null
3  if req.head.key ≤ k then dest = req
4  else req.state = search
5  for r in reqList do
6      r.state = pending
7      if dest == null && r.head.key ≤ k then
8          dest = r
9      else
10         rResult = r.result
11         addReqs(dest.head, rHead, k)
12         if rResult != null then
13             r.locked = completed
14             notify r.threadID
15 if dest == null then LCDunlock(req)
16 if dest != req && dest != null then
17     owner = dest.thread
18     dest.state = locked
19     notify dest.threadID
```

Algorithm 11. AddReqs(dst, src, k)

```
1  d = p = dst
2  for s = src; s != null; s = s.next do
3      if s.key > k then
4          s.head = s
5          s.state = search
6          notify s.theadID
7          return
8      while d.key < s.key do
9          p = d
10         d = d.next
11     if d.key > s.key then
12         s.next = d
13         p.next = s
14     else
15         if d.op == s.op || d.result then
16             s.result = false
17         else
18             d.result = s.result = true
19         d.combined.add(s)
```

The `addReqs` operation (Alg. 11) is invoked after a list node was locked, and it transfers requests from a source list, `src`, to a destination list, `dst`, provided that the requests in `src` require this lock. To verify this, `addReqs` receives the key of the list node's successor, k, and only requests whose keys are no greater than k are transferred. The `src` and `dst` lists are sorted by the operation keys in an ascending order.

The `addReqs` operations iterates the `src` list and at each iteration examines one request, denoted by `s`. If `s` has a key greater than k, `s`'s `state` is set to *search* and `s`'s `head` is set to `s`. Thereby, `s` becomes responsible to all requests succeeding it in `src`, whose keys are also greater than k as `src` is sorted (lines 3-7).

If `s`'s key is not greater than k, the combiner looks for two requests in `dst`, `p` and `d`, such that `s`'s key is greater than `p`'s key and not greater than `d`'s key (lines 8-10)[2]. If `d`'s key is greater than `s`'s key, `s` is inserted between `p` and `d` (lines 11-13). Otherwise, if the keys are equal, elimination is applied (lines 14-19). If `d` and `s` are identical operations, `s` is eliminated by setting its `result` to `false`. This is correct since if the combiner were to execute `d` and `s`, then `s` would fail. To illustrate, consider two requests of `insert(4)`: the first may succeed if 4 is not in the list, but the second will observe 4 and fail.

If `d` and `s` are complementary operations (i.e., an insert-remove pair) their `result` fields are set to `true`. If the combiner were to execute them, it could choose an ordering that would not affect the data structure. To illustrate, assume `d` and `s` are `insert(4)` and `remove(4)`. If 4 is not in the list, the combiner can execute `d` and then `s`; otherwise it can execute `s` and then `d`. Such elimination can be applied once for each request and thus if `d` is discovered to be eliminated, then `s` is eliminated by the request that eliminated `d` (and is identical to `s`), and thus `s`'s `result` is set to `false`.

After eliminating `s`, it is added to `d`'s `combined` set (line 19). After `d`'s operation is completed, the `state` of these requests is set to `completed`. This is required for

[2] For simplicity's sake, we omit the special treatment required if `s` becomes `dst`'s head or tail.

Algorithm 12. LCDRemove(k,req)

```
1  prev = head
2  while true do
3      prev, curr = search(prev, k)
4      if prev.LCDlock(req, remove, k) then
5          return combine(req, prev)
6      next = prev.next
7      if !prev.marked && next.key ≥ k then
8          if next.key != k then
9              prev.LCDunlock(req)
10             return false
11         if next.LCDlock(req, mark, k) then
12             combine(req, next)
13         else
14             next.marked = true
15             next.LCDunlock(req)
16         prev.next = next.next
17         prev.LCDunlock(req)
18         return true
19     prev.LCDunlock(req)
20     if prev.marked then prev = head
```

Algorithm 13. Combine(req, n)

```
1  while true do
2      if req.state == completed then
3          return complete(req)
4      if req.state == locked then
5          dne = execute(req, req.head, n)
6          n.LCDunlock(req)
7          if dne then
8              return complete(req)
9      if n.marked then
10         n = list.head
11     curr = n.next
12     while curr.key < req.head.key do
13         n = curr; curr = curr.next
14     n.LCDlock(req, req.op, req.key)
```

Algorithm 14. Complete(req)

```
1  for r in req.combined do
2      r.state = completed
3      notify r.threadID
4  return req.result
```

operations eliminated by identical operations, since their results are valid only after the eliminating requests were executed. In addition, it reduces the number of updates to the list. For example, delaying the completion of the complementary requests `insert(4)` and `remove(4)` enables to eliminate additional `insert(4)` with the first `insert(4)`.

4.3 LCD-List

The extended lazy-list `insert` and `remove` operations, denoted by `LCDInsert` and `LCDRemove`, access locks via `LCDlock` and `LCDunlock` (Alg. 12 shows `LCDRemove` which extends `remove`, and `LCDInsert` extends `insert` similarly). If `LCDlock` returns `false`, *LCD* was not initiated and the operation proceeds as in the lazy-list. If `LCDlock` returns `true`, then the invoking thread is either a combiner or its request was combined, and in any case, the operation is completed via the `combine` operation.

The `combine` operation (Alg. 13) receives the thread request, `req`, and the node which `LCDlock` attempted to lock, `n`. It begins by reading `req`'s `state` which may be *completed*, *locked*, or *search*. If `state` is *completed*, `req` was combined, and the operation completes (via the `complete` operation). If `state` is *locked*, `execute` is called and `n`'s lock is released. If `execute` returned `true`, then all requests were executed and `combine` completes. If there are remaining requests or if `state` is *search*, a new `n` node is located and attempted to be locked, and the loop begins again (lines 9-14).

The `complete` operation (Alg. 14) receives a request, reports threads whose requests were combined or eliminated by this request, and returns the request `result`.

The `execute` operation (Alg. 15) receives the combiner request, `req`, the head of its request list, `r`, and a locked node, `n`. It executes all requests requiring `n` and returns `true` if all the requests in the list were executed. It begins by finding the first request not eliminated in the request list, namely, a request whose `result` field is `null` (lines 1-2). If all requests were eliminated, it returns `true`.

If a non-eliminated request was found, two conditions are checked similarly to the lazy-list (line 7). If any condition fails, `execute` terminates (lines 31-36). Note that if the second condition fails, namely, `r`'s `key` is greater than `next`'s key, then also all requests succeeding `r` require a different lock since the list is sorted by the operation keys.

If both conditions are met, `r` begins execution. First, it is added to `req`'s `combined` set (line 8) so it will be reported after `req` is completed[3]. Then, `r`'s operation type is examined.

If `r`'s operation is *insert* (lines 9-19), and it is a successful insert, a new node is created and added to a temporary sublist consisting of all nodes created by insertion requests. The sublist is connected to the list when `execute` terminates and only then these requests' `result` fields are set to `true` (lines 31-34).

If `r`'s operation is *remove* (lines 20-26), and it is a successful removal, the `removeNode` operation is invoked. Note that the sublist cannot contain `r`'s `key`, as complementary requests are eliminated.

If `r`'s operation is *mark*, it is delayed until all other requests are completed (lines 27, 35). If `r` would have been served immediately by marking `n`, then no subsequent request could have been executed as the check in line 7 would fail. `Execute` may encounter at most one *mark* request, since a thread initiating a *mark* request locks the node preceding `n` until it is completed.

After serving `r`, the next request to execute is searched for (lines 28-30).

The `removeNode` operation (Alg. 16) receives two consecutive nodes, `p` and `n`, and it removes `n` from the list. It begins by invoking `LCDlock` to lock `n` with a fresh request, `frq` (to avoid overriding

Algorithm 15. Execute(req, r, n)

```
1  while r != null && r.result != null do
2      complete(r); r = r.next
3  if r == null then return true
4  insReqs = new List()
5  subHead = null; subTail = null
6  next = n.next
7  while !n.marked && r.key ≤ next.key do
8      req.combined.add(r)
9      if r.op == Insert then
10         if r.key == next.key then
11             r.result = false
12         else
13             new = Node(r.key)
14             if subHead == null then
15                 subHead = new
16             else
17                 subTail.next = new
18             subTail = new
19             insReqs.add(r)
20     if r.op == Remove then
21         if r.key < next.key then
22             r.result = false
23         else
24             removeNode(req, n, next)
25             next = n.next
26             r.result = true
27     if r.op == Mark then mark = r
28     while r != null && r.result != null do
29         complete(r); r = r.next
30     if r == null then break
31 if subHead != null then
32     subTail.next = n.next
33     n.next = subHead
34 for r in insReqs do r.result = true
35 if mark != null then  n.marked = true
36 return r == null
```

Algorithm 16. RemoveNode(req, p, n)

```
1  n.LCDlock(frq, mark, n.key)
2  if frq.state == locked then
3      n.marked = true
4      n.LCDunlock(frq)
5      k = n.next.key
6      addReqs(req.head, frq.head.next, k)
7  p.next = n.next
```

[3] This is a simplification, requests are actually notified sooner.

req fields). Then, frq's state is examined, and it may be either *locked* or *completed* (it cannot be *search*, as p is locked and thus n's lock must be the required lock). If state is *completed* then p's next is set to n's next. Otherwise, if state is locked, n's marked flag is set to true and n's lock is released. Then, collected requests are transferred from frq to req, starting from the second request in frq's request list (the first one is frq since the list is sorted). Finally, p's next is set to n's next.

5 Correctness

Here, we provide the linearization points [15] of the *LCD*-list operations.

Unsuccessful inserts and removes (which were not eliminated) are linearized when discovered to be unsuccessful in LCDInsert (similarly to the discovery in insert, line 7), LCDRemove (line 8), or execute (lines 10, 21).

A successful insert is linearized in LCDInsert when prev is set to point to the new node (similarly to the update in insert, line 12), or in execute when the sublist containing the new node is appended to the linked-list (line 33). This linearization point linearizes "atomically" all successful inserts whose nodes share the same sublist. That is, these inserts are linearized at a single point, and no other operation is linearized between them. The linearization order between the inserts is by their keys.

A successful remove is linearized when the node's marked flag is set to true, which is either in LCDRemove (line 14), execute (line 35), or removeNode (line 3).

Eliminated operations are grouped into disjoint sets as follows. Two complementary operations which were eliminated (addReqs, line 18), belong to the same set. Operations that were eliminated by an identical operation or by a complementary eliminated operation (addReqs, line 16) belong to the set of that operation. The operations of each disjoint set are linearized together "atomically" in one of the following points. If the operation set contains two complementary operations, then the linearization point is upon the first update to a state field of any of these operations (complete, line 2). If the operation set does not contain complementary operations, then it consists of identical operations and one of them is executed in execute. In this case, the linearization point of the operation set is the linearization point of the executed operation.

The linearization point ordering between operations belonging to the same operation set is as follows. If the operation set contains two complementary operations, i.e., insert(k) and remove(k), then the ordering is one of the following. If k is present in the linked-list, then first appears the remove(k) whose result is true. Next, all the other remove(k) appear in some order. Then, the insert(k) whose result is true appears; and finally, all the other insert(k) appear in some order. If k is not present in the linked-list, the ordering is similar only that inserts are linearized before removes. If the operation set contains only identical operations then the order is first the operation that was executed (in execute) and then the other operations in some order.

The linearization point of the contains operation is set similarly to the original lazy-list algorithm. A successful contains is linearized when the marked flag is found false. An unsuccessful contains has two possible linearization points. The first case is when k is indeed not in the linked-list and curr's key is found to be greater than k, or curr is found to be marked. Here, the linearization point is when curr is examined. The second case is similar to the first case only that k is concurrently inserted. Since contains returns false, it is linearized just before the concurrent insert. Note that this

linearization point occurs after `contains` begins, since otherwise `contains` would have found k. Also note that the linearization points of `contains` are not affected by inserts which were eliminated by complementary removes, since these operations do not affect the list and are linearized "atomically" with their complementary removes.

6 Performance Evaluation

We implemented the *LCD*-list in Java and ran experiments on an AMD Opteron Processor 6376 with 128GB RAM and 64 cores: four processors with sixteen cores each. We used Ubuntu 12.04 LTS and OpenJDK Runtime version 1.7.0_65 using the HotSpot 64-bit Server VM (build 24.45-b08, mixed mode).

We compared the *LCD*-list to the following list implementations:

- The Lazy-List – the optimized lazy-list algorithm as presented in Section 2.1.
- The Flat Combining List [12] – this list is protected by a global lock and the lock owner executes operations of contending threads. We used the authors' code [2].
- The Lock Free List [10] – the lock-free list by Harris. We used the code provided with the book "The Art of Multiprocessor Programming" [14].

We also consider two variations of the *LCD*-list to evaluate two of its main features:

- *LCD* without elimination – the *LCD*-list without eliminating operations.
- *LCD* without integration – the *LCD*-list without the Java lock integration.

We ran two workloads:

(I) 0% `contains`, 50% `insert`, and 50% `remove`.
(II) 60% `contains`, 20% `insert`, and 20% `remove`.

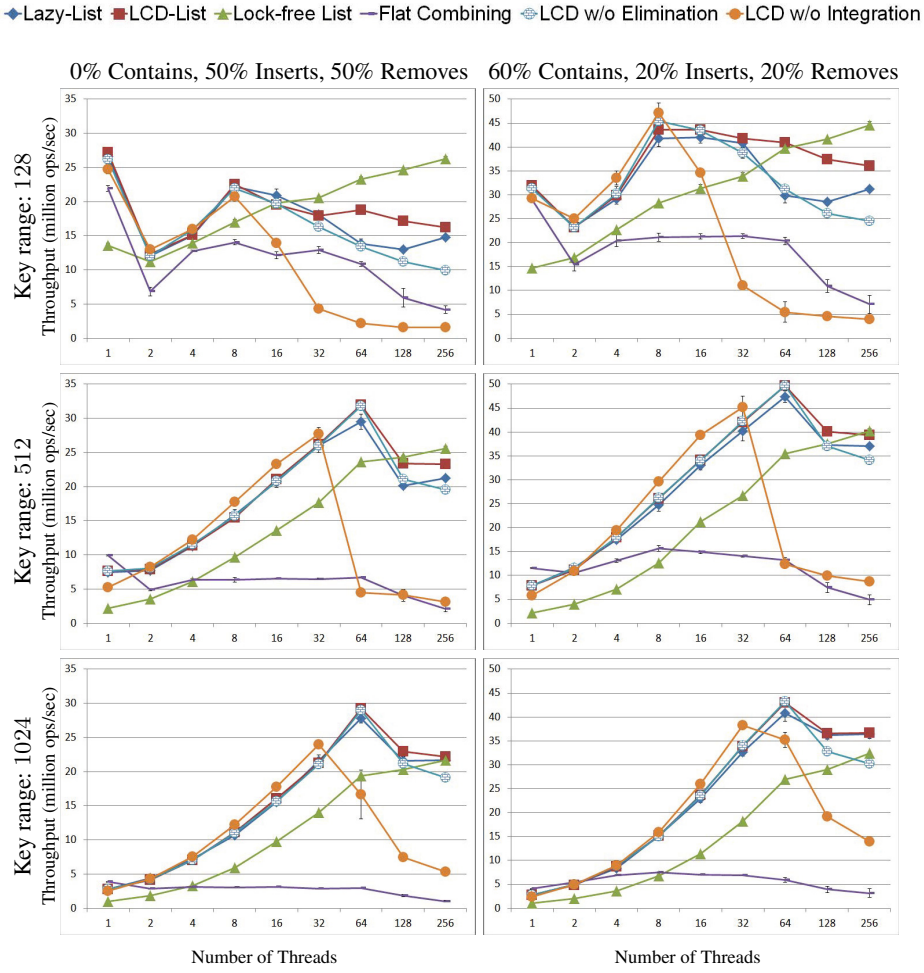
We ran five-second trials, where each thread reported the number of operations it completed. We report the total throughput, namely the total number of completed operations. The number of threads is 2^i where i varies between 0-8.

During the trial, each thread randomly chooses a type of operation according to the workload distribution and then randomly chooses the key for that operation from a given range. The examined range sizes were: 128, 512, and 1024. Before each trial, the list was prefilled to a size of half of the key range. Every experiment was run 8 times and the arithmetic average is reported along with the 95% confidence interval. Each batch of 8 trials was run in its own JVM instance, and a warm-up phase was run before to avoid HotSpot effects. Threads were not bound to processors in our experiments.

Table 1 reports the results. The experimental results show that *LCD* improves the lazy-list performance on most workloads, mostly by 5%-15% and up to 40%. Under low contention, the *LCD*-list performance is similar to the lazy-list and its overhead is negligible. The lock-free list performance is at best under heavy contention and it decreases significantly under lower contention. The flat combining performs poorly as it blocks all concurrent accesses to the list even if they do not contend.

The *LCD* features appear to improve the *LCD* performance. The performance of the *LCD*-list without the operation elimination is mostly similar to the *LCD*-list, but as contention increases and more operations can be eliminated its performance is reduced. The *LCD*-list without the Java lock integration performs better than the *LCD*-list under low contention and performs poorly under heavy contention. The integration into the Java lock introduces overhead since volatile fields of the Java lock are updated (e.g., the queue head and tail), and this overhead is significant under low contention. However, the lock integration becomes very beneficial as contention increases.

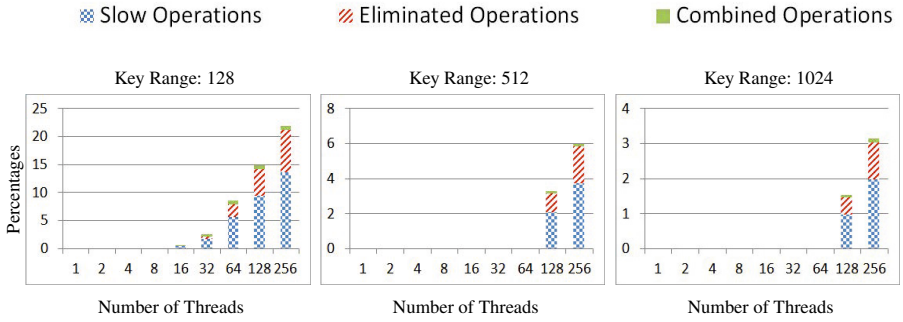
Table 1. Throughput of linked-list implementations (64 h/w threads)



We next study how often *LCD* is applied in the *LCD*-list. There are four avenues for an operation to be executed: (i) a fast execution without *LCD*, (ii) a slow execution when the operation owner is a combiner, (iii) operation elimination, or (iv) combining.

Table 2 reports the percentage of operations completed in each *LCD* avenue ((ii)–(iv)). The summation of each column is the percentage of operations completed in an *LCD* avenue. The two workloads yielded similar results, thus we present only the results of the second workload. The results show that as contention increases, more operations use *LCD* (up to 22% of the operations). As the number of threads increases, many operations are eliminated. The number of combined operations is low on all workloads.

Table 2. Distribution of *LCD* avenue types



7 Summary

We presented the *local combining on-demand* methodology (*LCD*), a combining technique for data structures with an unbounded number of contention points. We designed and implemented the *LCD*-list, an extension of the lazy-list that provides *LCD*. Evaluation shows that the *LCD*-list outperforms the lazy-list typically by 7% and up to 40%.

References

1. Bar-Nissan, G., Hendler, D., Suissa, A.: A dynamic elimination-combining stack algorithm. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 544–561. Springer, Heidelberg (2011)
2. Bronson, N.: The flat combining project, `http://mcg.cs.tau.ac.il/projects/flat-combining`
3. Budovsky, V.: Combining techniques application for tree search structures, m.sc. thesis. Tel-Aviv University, Israel (2010)
4. Craig, T.: Building fifo and priority-queuing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Dept. of Computer Science (1993)
5. Dice, D., Marathe, V.J., Shavit, N.: Flat-combining numa locks. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011, San Jose, California, USA, pp. 65–74. ACM, New York (2011)
6. Dice, D., Marathe, V.J., Shavit, N.: Lock cohorting: A general technique for designing numa locks. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, Louisiana, USA, pp. 247–256. ACM, New York (2012)
7. Fatourou, P., Kallimanis, N.D.: Revisiting the Combining Synchronization Technique. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, New Orleans, Louisiana, USA, pp. 257–266. ACM, New York (2012)
8. Fomitchev, M., Ruppert, E.: Lock-free Linked Lists and Skip Lists. In: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, pp. 50–59. ACM, New York (2004)
9. Gupta, R., Hill, C.R.: A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. International Journal of Parallel Programming 18(3), 161–180 (1990)
10. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)

11. Heller, S., Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
12. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat Combining and the Synchronization-parallelism Tradeoff. In: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, Thira, Santorini, Greece, pp. 355–364. ACM, New York (2010)
13. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Scalable flat-combining based synchronous queues. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 79–93. Springer, Heidelberg (2010)
14. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
15. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (1990)
16. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (1991)
17. Oyama, Y., Taura, K., Yonezawa, A.: Executing parallel programs with synchronization bottlenecks efficiently. In: Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, PDSIA 1999, pp. 182–204 (July 1999)
18. Shavit, N., Zemach, A.: Combining Funnels: A Dynamic Approach to Software Combining. J. Parallel Distrib. Comput. 60(11), 1355–1387 (2000)
19. Shavit, N., Zemach, A.: Diffracting Trees. ACM Trans. Comput. Syst. 14(4), 385–428 (1996)
20. Valois, J.D.: Lock-free Linked Lists Using Compare-and-swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, Ottowa, Ontario, Canada, pp. 214–222. ACM, New York (1995)
21. Yew, P.-C., Tzeng, N.-F., Lawrie, D.H.: Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. IEEE Trans. Comput. 36(4), 388–395 (1987)