A Lazy Concurrent List-Based Set Algorithm

Steve Heller[1], Maurice Herlihy[2], Victor Luchangco[1], Mark Moir[1], William N. Scherer III[3], and Nir Shavit[1]

[1] Sun Microsystems Laboratories
[2] Brown University
[3] University of Rochester

Abstract. List-based implementations of sets are a fundamental building block of many concurrent algorithms. A skiplist based on the lock-free list-based set algorithm of Michael will be included in the Java™ Concurrency Package of *JDK 1.6.0*. However, Michael's lock-free algorithm has several drawbacks, most notably that it requires all list traversal operations, including membership tests, to perform cleanup operations of logically removed nodes, and that it uses the equivalent of an atomically markable reference, a pointer that can be atomically "marked," which is expensive in some languages and unavailable in others.
We present a novel "lazy" list-based implementation of a concurrent set object. It is based on an optimistic locking scheme for inserts and removes, eliminating the need to use the equivalent of an atomically markable reference. It also has a novel wait-free membership test operation (as opposed to Michael's lock-free one) that does not need to perform cleanup operations and is more efficient than that of all previous algorithms.
Empirical testing shows that the new lazy-list algorithm consistently outperforms all known algorithms, including Michael's lock-free algorithm, throughout the concurrency range. At high load, with 90% membership tests, the lazy algorithm is more than twice as fast as Michael's. This is encouraging given that typical search structure usage patterns include around 90% membership tests. By replacing the lock-free membership test of Michael's algorithm with our new wait-free one, we achieve an algorithm that slightly outperforms our new lazy-list (though it may not be as efficient in other contexts as it uses Java's RTTI mechanism to create pointers that can be atomically marked).

1 Introduction

Lists are a fundamental building block for concurrent data structures, both in their own right, and as the basis for many types of search and dictionary data types [12]. We consider three kinds of list operations: inserting a list entry, removing a list entry, and testing whether an entry is in the list.

This paper introduces the *lazy list*, a simple new concurrent *list-based set* algorithm with a number of novel concurrency-related properties. To explain the novel aspects of lazy lists, we start with an overview of different ways to synchronize lists. *Coarse-grained* locking, which uses a single lock to protect the entire list, has the advantage of simplicity, but provides no concurrency. With *lock coupling* (sometimes called "hand-over-hand" locking) [1], a thread acquires the lock for each successive entry before releasing the lock for its predecessor. Lock coupling provides more concurrency than coarse-grained locking, but threads may acquire many successive locks, which is undesirable because lock acquisition typically involves expensive atomic operations (such as compare-and-swap). Moreover, concurrent threads moving through the list may contend for locks even if they are searching for unrelated list entries. Valois [14] was the first to suggest a non-blocking implementation of a concurrent list-based set. Harris [3] and later Michael [10], presented highly efficient lock-free algorithms for list-based sets. Fomitchev and Ruppert [10] present more complex algorithms that provide an amortized cost guarantee for all operations that is provably linear in the length of the list. Michael's algorithm is the basis for a concurrent skip-list data structure in the Java™ Concurrency Package of JDK 1.6.0.

As in most previous list-based set algorithms, we represent a set as a sorted linked list. In our new lazy list algorithm, insertion and removal operations are *optimistic*: each operation searches the list without acquiring any locks or interfering with other threads. When an operation locates the entry it is seeking it locks that entry and its predecessor and checks for synchronization conflicts. If no conflict is detected, an entry is inserted or removed, and otherwise the locks are released and the operation is restarted.

This optimistic approach to insertion and removal has the advantage that insert and remove calls that access non-adjacent list entries never interfere. In the absence of synchronization conflicts, these operations acquire only a constant number of locks. Entries are removed from the list in a *lazy* manner: the entry is first marked as removed (the "logical" removal), and then it is physically unlinked from the list (the "physical" removal). The simplifying power of lazy techniques has been exploited by Harris [3] and Michael [10] for concurrent lists, and by Maier [9] in more general contexts. Nevertheless, the algorithms of Harris and Michael require the ability to perform an atomic compare-and-swap on two fields at once: a Boolean marked field and a reference field to the next entry in the list (the equivalent of an `AtomicMarkableReference` in the Java Programming Language). Since in many systems it is unacceptable to "steal a bit" from a reference, one must use alternative techniques. In modern object oriented languages, one can have two trivial (empty) subclasses of a node object and use a *run time type identification* (RTTI) mechanism [2] to determine which subclass the current instance belongs to, where each subclass represents a state of the bit. In languages without RTTI support, one can use an additional level of indirection, adding a pointer to a special dummy node to signify that the bit is set. This is the mechanism used to implement `AtomicMarkableReference` in the Java Concurrency Package, which unfortunately can introduce significant performance penalties.

Perhaps the most substantial advantage of the new algorithm is that membership test operations are *wait-free* [4]. The lock-freedom progress property of the membership test in Michael's algorithm guarantees that if some threads are executing method calls, and at least one thread continues taking steps, then at least one thread will complete its call, but makes no progress guarantee for any individual thread. Wait-freedom is a stronger progress property that guarantees that any thread that continues taking steps in executing a method call, will eventually complete the call.

The membership test of our algorithm acquires no locks, requires no synchronization, and never interferes with any concurrent operations. This last property is particularly important because it is reasonable to expect that in most real-world applications, membership tests are by far the most common operations. In Michael's lock-free list algorithm, and unlike in ours, if a thread traversing the list encounters an entry that has been logically but not physically removed, then the thread must stop to complete the physical removal. Physical removal requires calling a compare-and-swap operation, and if several concurrent threads attempt to remove the same entry, then only one will succeed, and the rest will be forced to abandon their traversals and start over. While the number of such removals is likely to be small, our empirical testing shows that when there is a high level of concurrent traversals, contention among threads competing to perform the removal causes a large number of traversals to be abandoned and restarted.

By contrast, in the new lazy list algorithm, only the remove operations are required to perform physical removals, while the insertion and (more importantly) membership query traversals are not delayed by physical removals. The wait-free nature of the membership operation means that ongoing changes to the list cannot delay even a single thread from deciding membership. We note that our wait-free membership test is of independent value: one can readily replace the membership test in Michael's algorithm with the lazy list's new membership test, allowing it to obtain improved performance by eliminating the need for physical removals.

To evaluate our new lazy list algorithm, we implemented it in the Java™ programming language and conducted a series of benchmarks comparing our new algorithm to known algorithms on a 16 node SunFire™ 6800 cache coherent bus-based multiprocessor machine. We found that when there is a high fraction of membership tests (as in search structures) the new lazy list algorithm and a new version of Michael's algorithm that uses our wait-free membership test, outperform all others by a factor of two or more. The good performance of our new version of Michael's lock-free list depended on the use of Java's RTTI mechanism. We also found that as the fraction of membership queries dropped, the relative performance advantage of the lazy list disappeared, and the new version of Michael's list with our wait-free membership test showed the best performance.

In summary, we conclude that adding the new wait-free membership test always offers a performance advantage and has no performance penalties. For applications with a high fraction of membership tests, one should definitely use the new algorithms, while the choice of which algorithm to use—the new lazy list, or Michael's lock-free list with our new wait-free membership test—seems to depend on the cost and availability of mechanisms for implementing the equivalent of `AtomicMarkableReference` in a given system and language.

Following our initial presentation of the algorithms in this paper, a complete formal treatment was provided by Vafeiadis et al in [13]. We therefore focus on providing an informal and easily accessible explanation of why our new algorithm works, and refer the interested reader to [13] for the detailed correctness proofs.

2 The New Algorithm

We present our concurrent linked-list implementation in the context of a list-based set object. For our purposes, a *Set* provides three methods:

– The `add`(x) method adds x to the set, returning *true* if and only if x was not already in the set.
– The `remove`(x) method removes x from the set, returning *true* if and only if x was in the set.
– The `contains`(x) method returns *true* if and only if the set contains x.

For each method, we say that a call is *successful* if it returns *true*, and *unsuccessful* otherwise.

Linearizability [6] is a standard correctness condition for concurrent data structures. The list-based set implementation that we present is a linearizable implementation of a set object. To prove this it is enough to identify, for each method call in each possible execution history, a *linearization point*, a single operation when the method call "takes effect". For example, the linearization point defines exactly when `add`(a) adds an entry, a point during the execution of the method immediately before which a is not in the set, and immediately after which a is in the set.

Lock-freedom is a progress property that guarantees that if some threads are executing method calls, and at least one thread continues taking steps, then at least one thread will complete its call. It guarantees that the system as a whole continues to make progress, but makes no progress guarantee for any individual thread. *Wait-freedom* is a stronger progress property that guarantees that any thread that continues taking steps in executing a method call, will eventually complete the call.

As noted earlier, following our initial presentation of the algorithms in this paper, a complete formal treatment was provided by Vafeiadis et al in [13]. We therefore focus here on giving an informal and easily readable explanation of why our new algorithm works.

We represent the set as a sorted list of entries. As shown in Figure 1, the `Entry` class has four fields. The `key` field is the set element. Our algorithm works for any ordered set of keys that has maximum and minimum values and is well-founded, that is, for any given key, there are only finitely many smaller keys. This is trivially satisfied by most real-world key types because the size of the `key` is fixed; for simplicity, we present our algorithm assuming that the keys are integers. We will use the well-foundedness assumption to technically capture

```
private class Entry {
  int key;
  Entry next;
  boolean marked;
  lock lock;
}
```

Fig. 1. List entry: an entry keeps track of the set element itself (the key), the next entry in the list, a marked field to denote logical removal of the entry, and a `lock` field for synchronization.

the notion that the progress of a membership query in Michael's algorithm is lock-free while the new algorithm's membership query is wait-free.

The list is maintained in `key` order, providing an efficient way to determine whether a given key is in the list. We sometimes abuse notation slightly and use the same symbol to refer to an entry and its associated key (entry a will have key a and so on.). The `next` field is a reference to the next entry in the list, the `marked` field indicates if its associated key is logically removed or still in the data structure, and the `lock` field is a lock used for synchronization.

We assume that the `add`(), `remove`(), and `contains`() methods are the *only* ones that modify entries, a property sometimes called *freedom from interference.* We require freedom from interference even for entries that have been removed from the list, since a thread may unlink an entry while it is being traversed by others. In a language such as Java, we can rely on the garbage collector to recycle unreachable entries. In a programming language without garbage collection, this property can be maintained by using methods like ROP [5] or SMR [11].

The list has two kinds of entries. In addition to *regular* entries that hold elements (keys) in the set, we use two *sentinel* entries, called `head` and `tail`,

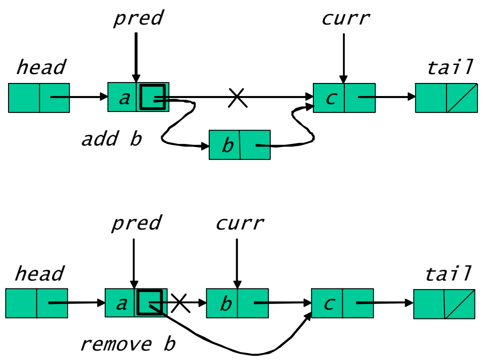

Fig. 2. Insertion and removal of list entries.

```
 public boolean remove(int key) {
    while (true) {
      Entry pred = this.head;
      Entry curr = head.next;
      while (curr.key < key) {
        pred = curr; curr = curr.next;
      }
      pred.lock();
      try {
        curr.lock();
        try {
          if (validate(pred, curr)) {
            if (curr.key != key) {     // present
              return false;
            } else {                   // absent
LR:           curr.marked = true;    // logically remove
              pred.next = curr.next;  // physically remove
              return true;
            }
          }
        } finally {                   // always unlock curr
          curr.unlock();
        }
      } finally {                     // always unlock pred
        pred.unlock();
      }
    }
  }
```

Fig. 3. The lazy `remove`() method: removes entries in two steps, logical and physical.

as the first and last list entries. The sentinel entries contain the minimum and maximum key values, respectively; we assume that these values are never added, removed or searched for. Ignoring the details of synchronization for the moment, the top part of Figure 2 shows a schematic description of how a key is added to the set. Each thread has two local variables used to traverse down the list: `curr` is the current entry and `pred` is its predecessor.

To add a new key to the set, a thread sets the local variable `pred` to `head` and `curr` to `head`'s successor, and moves down the list, comparing `curr`'s key to the key being added. If they match, the key is already present in the set, so the thread returns *false*. If `pred` precedes `curr` in the list, `pred`'s key is lower than the inserted key, and `curr`'s key is higher, then the key is not present in the list. Therefore, the thread creates a new entry b to hold the key, sets b to point to `curr`, and then sets `pred` to point to b. The key is now a member of the set.

```
private boolean validate(Entry pred, Entry curr) {
  return  !pred.marked && !curr.marked && pred.next == curr;
}
```

Fig. 4. The lazy lists validation.

Removing a key is similar: we scan the list to find the relevant adjacent pair of entries. The target entry is removed from the list in two steps: first, its `marked` field is set to *true*, indicating that the entry has been *logically* removed from the list, and second, the predecessor entry's `next` field is redirected to point to the successor entry, *physically* removing the entry from the list. As discussed more precisely later, the removal "actually happens" when an entry is marked, and the physical removal is just a way to clean up.

2.1 The `remove()` method

As shown in Figure 3, when the `remove`() method attempts to remove the entry with key k, it scans through the list without acquiring any locks, traversing both marked and unmarked entries. The `remove`() method uses two local variables: `curr` is the current entry and `pred` is its predecessor. When `curr` is set to the first entry with a key greater than or equal to k, the traversal stops, and the method locks `curr` and `pred`. Because there is a gap between the unsynchronized traversal and the lock acquisition, it is necessary to *validate* that the method has locked the correct entries. What can go wrong? There are three obvious problems: the `curr` entry could have been removed, the `pred` entry could have been removed, or another entry may have been inserted between `pred` and `curr`. Surprisingly, perhaps, these are the *only* things that can go wrong, and moreover, they can be detected very efficiently. It is enough to check that `curr` and `pred` are both unmarked, and that `pred`'s `next` pointer points to `curr` (see Figure 4). If these conditions hold, the entries are adjacent and present in the list. If the validation succeeds, the `remove`() method logically removes the entry, physically removes the entry, releases both locks and returns *true*. If the entry with key k is absent, the method unlocks the entries and returns *false*. If the validation fails, the thread restarts the method.

For an unsuccessful `remove`() call, the linearization point is the point at which it finds (reads the pointer to) a marked entry with the same key or the first unmarked entry with a larger key. For a successful `remove`() method call, the linearization point is the moment the entry is marked (line `LR` of Figure 3).

2.2 List traversal

We pause momentarily to discuss list traversal. The list traversal in the `remove`() method in Figure 3 seems straightforward: simply follow the list pointers. The same approach is used in the `add`() and `contains`() methods. It is important to note that this traversal differs from those of other concurrent list-based set algorithms in the literature in two important ways:

- it requires no additional synchronization (such as acquiring locks [1] or cleaning up logically removed nodes [10]), and
- it traverses both logically and physically removed nodes.

This latter property, which allows us to achieve the former, is the key to our algorithm's good performance. Figure 7 shows how a concurrent physical removal of a node during thread A's traversal can cause it to traverse a physically removed part of the list. The traversal works correctly because we assume the freedom from interference property which implies that nodes, even if they are removed from the list, are not recycled (freed back to the available memory pool) as long as they are reachable. Thus, if a node is removed while it is being traversed, the traversing thread will continue to follow the list of pointers and eventually reach its target node. Our algorithm maintains the property that if an entry was in the list when a given thread started searching for it, it will remain reachable from this thread's `curr` pointer as long as it is not removed.

2.3 The `add`() method

Like the `remove`() method, the `add`() method (Figure 5) scans the list without acquiring locks, until `curr` is set to the first entry with a key greater than or equal to the key to be inserted. The method locks both entries, validates them, and if an entry with the specified key is not already present in the list, inserts a new entry, unlocks the entries, and returns *true*. The remaining cases are just as in the `remove`() method. For an unsuccessful `add`() method call, the linearization point is the moment at which the entry is observed to be unmarked in the list. For a successful `add`() method call, it is the moment when `pred.next` is set. We note that one can make the `add`() method more efficient by locking only the `pred` node, but for the sake of keeping our algorithm simple, we omit this optimization here.

2.4 The wait-free `contains`() method

The key to the performance of our algorithm is the new wait-free `contains`() method. This method is of independent interest. For example, we show in Section 3 that it can readily replace the lock-free `contains`() method in the algorithm of Michael [10] to provide improved performance.

The `contains`() method scans the list, just like the `remove`() and `add`() methods, ignoring whether nodes are marked or not, until `curr` is set to the first entry with a key greater than or equal to the sought-after key. Instead of locking the entry, however, it simply returns *true* if and only if the `curr` entry is unmarked with the desired key. This is correct since the list is ordered and so, if a node is removed, it must be marked or not present in the list.

It is easy to see that this method is wait-free. First, notice that because the universe of keys is well-founded there are only a finite number of keys that are

```
public boolean add(int key) {
  while (true) {
    Entry pred = this.head;
    Entry curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) { // present
            return false;
          } else {               // not present
            Entry entry = new Entry(key);
            entry.next = curr;
            pred.next = entry;
            return true;
          }
        }
      } finally { // always unlock
        curr.unlock();
      }
    } finally { // always unlock
      pred.unlock();
    }
  }
}
```

Fig. 5. The `add()` method.

```
public boolean contains(int key) {
  Entry curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

Fig. 6. The lazy list's wait-free `contains()` method.

smaller than the one being searched for. According to the algorithm, entries with lower or equal keys to a given entry will never be added ahead of it (i.e. so that they are reachable from it) even if the entry points into the list but is logically and physically removed from the list. Thus, each time the traversal moves to a

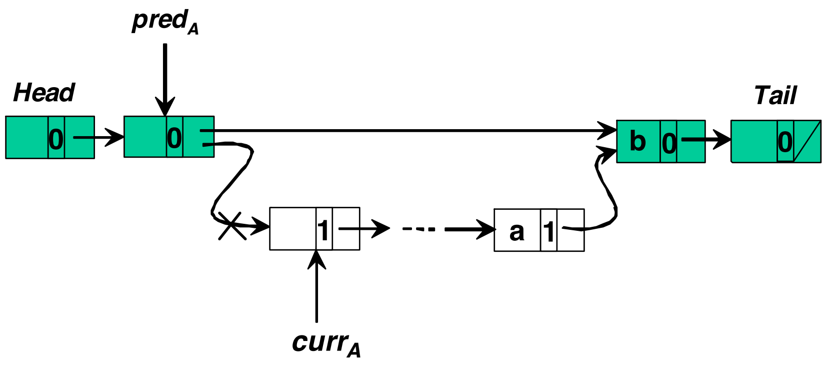

Fig. 7. Linearizing an unsuccessful `contains()` method calls is a bit tricky. Dark nodes are physically in the list and white nodes are physically removed. During a traversal of the list by thread A, the sublist starting at the node pointed to by `curr` (and schematically represented by "...") may be disconnected from the main list by a concurrent `remove()` method execution. Both nodes with items a and b can still be reached, and the determination if an item is in the list is based solely on the mark-bit.

new node, the new node has a larger key than the previous one, and this can happen only finitely many times, which implies that traversal is wait-free. This contrasts with Michael's membership test [10] which is only lock-free [4], since it can be forced to restart its traversal from the beginning of the list infinitely often if the same item is re-inserted and removed, and it fails each time when attempting to clean it up.

A successful `contains()` method call is linearized when the marked field of a matching entry is observed to be false. Linearizing an unsuccessful `contains()` method call is a bit tricky, and is a good example showing that it is not always possible to define a single linearization point for each method that works for all method calls in all executions. In particular, simply choosing the linearization point for an unsuccessful `contains()` as the point at which a marked entry with the sought-after key or an entry greater than the sought-after key is found is incorrect. Consider the following scenario. Assume that entry `a` is marked and thread A is attempting to find the entry matching `a`'s key. While A is traversing the list, $\mathtt{curr}_A$ and all entries between $\mathtt{curr}_A$ and `a` including `a` are removed logically and physically. Thread A would still proceed to the point where $\mathtt{curr}_A$ points to `a`. It would then detect that `a` is marked and therefore no longer in the list. Linearizing at this point is correct in this case. However, consider what happens if while thread A is traversing the removed section of the list leading to `a`, and before it reaches the removed `a`, another thread adds a new entry with a key `a` to the reachable part of the list. Linearizing the unsuccessful `contains()` method at the point at which it observed the marked entry `a` would be wrong, since it occurs *after* the insertion of the new entry with key `a` to the list.

We therefore linearize an unsuccessful `contains`() method call within its execution interval at the earlier of the following points: (1) the point where a removed matching entry is found and (2) the point immediately before a new matching entry is added to the list. As can be seen, this linearization point is determined by the ordering of events in the execution, and not predetermined as a specific point in the method execution.

3 Performance

We evaluated our new algorithm on a SunFire™ 6800 cache coherent bus-based multiprocessor machine with 16 1.2 GHz processors. The algorithms were implemented in Java 1.5.0. We varied the percentage of `contains`() method calls and the percentage of `add`() and `remove`() method calls. Each thread randomly selected both the type of call to make (respecting the given percentages) and the operand for it; operands are integers in the range 0..1023. We repeated this test suite both with and without an additional load of 16 threads performing computation in order to evaluate the sensitivity of our results to background load, but do not report the additional load tests here as there were no significant differences noted. In all our benchmarks, we measured *throughput*: the total number of calls completed over the course of 8 seconds, averaged across three runs. We tested six different list algorithms in all.

– *Coarse* – We use a single `java.util.concurrent.ReentrantLocks` lock to protect all access to the list.
– *Fine* – This is a *fine grained* hand-over-hand locking (lock-coupling) [8, 1] list-based implementation using a lock per list entry. Threads traverse down the list holding multiple locks at a time, releasing the earlier acquired entry's lock only after acquiring the next one in the list.
– *LockFree* – This is a *lock-free* list implemented according to Michael's algorithm [10], using the `AtomicMarkableReference` of JDK 1.5.0 to allow a markable next pointer per entry. As in our algorithm, the mark is used to denote that an entry is logically removed. Unlike in our algorithm, the `contains`() method is lock-free and not wait-free as calls do not traverse marked entries, instead, they clean them up before continuing traversal down the list.
– *LockFreeRTTI* – This is the lock-free list of Michael's algorithm [10] using the Java RTTI mechanism to distinguish marked entries. Such mechanisms are not available in all languages. Achieving the effect of marking a bit in the `next` pointer is done more efficiently than with `AtomicMarkableReference` by having two trivial (empty) subclasses of each entry object and using RTTI to determine at runtime which subclass the current instance is, where each subclass represents a state of the mark bit.
– *NewLockFreeRTTI* – This is *LockFreeRTTI* with Michael's lock-free `contains`() method directly replaced by the new wait-free `contains`() method of this paper, one that does not clean up marked entries and instead traverses them in a wait-free manner.

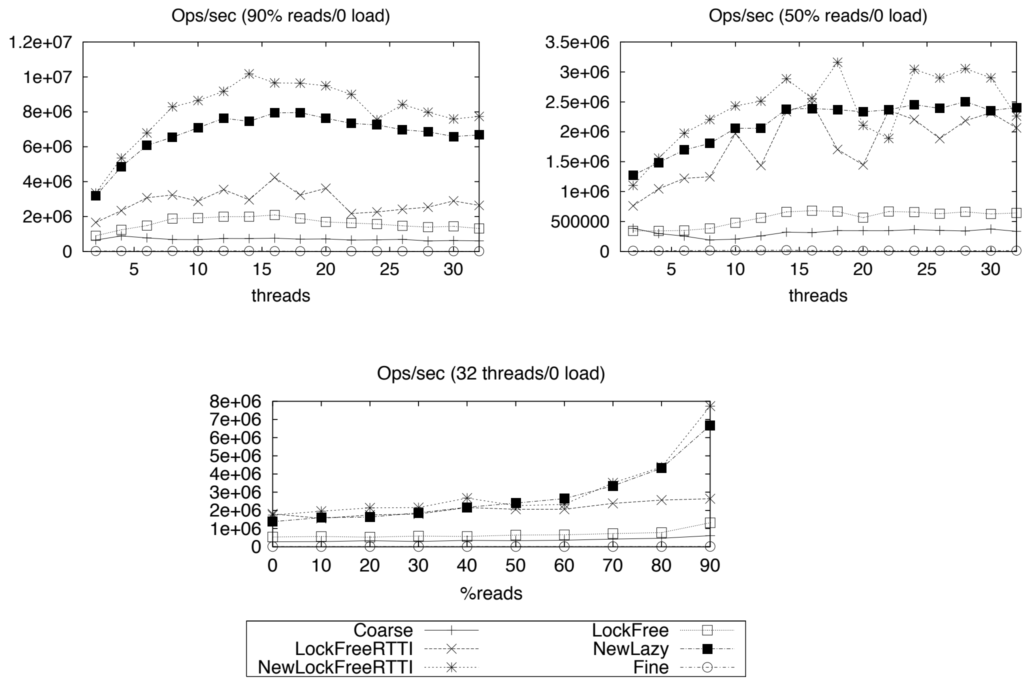

Fig. 8. The top two graphs show the change in throughput as concurrency increases to 32 threads with 60% and 90% of the operations being `contains`() method calls, and a 9/1 ratio of `add`() to `remove`() method calls. The bottom graph shows the change in throughput for the case of 32 threads as the fraction of `contains`() calls increases to 90%

– *NewLazy* – This is the new lazy list algorithm of this paper, with its new wait-free `contains`() and an optimization of the `add`() method to use only a single lock.

The top of Figure 8 shows the results of running a benchmark with 90% `contains`() method calls, 9% `add`() method calls and 1% `remove`() method calls (left) and another benchmark with 50% `contains`() method calls, 45% `add`() method calls and 5% `remove`() method calls (right). The 90/9/1 ratio and the high fraction of `add`() method calls to `remove`() method calls are considered typical of search structures, a common application of linked-lists [7].

If we look at the graph of the 90% test on the lefthand side of Figure 8, we see that the two new algorithms, the lazy list and the new lock-free list with a wait-free `contains`() method, outperform all others by a factor of two or more, including both versions of Michael's lock-free list, the one implemented with `AtomicMarkableReference` and the one implemented with the RTTI mechanism. The reason for this is as follows: even though there is a very small fraction of remove() method calls, there are many concurrent contains() method traversals, and in both of the original versions of Michael's algorithm they all compete to clean up the same small set of logically removed entries. All traversals that fail must restart, leading to a significant overhead. The new version of Michael's algorithm with RTTI and our wait-free contains() method performs slightly better than the lock-based lazy list. However, the reader is reminded that many languages do not have the equivalent of RTTI.

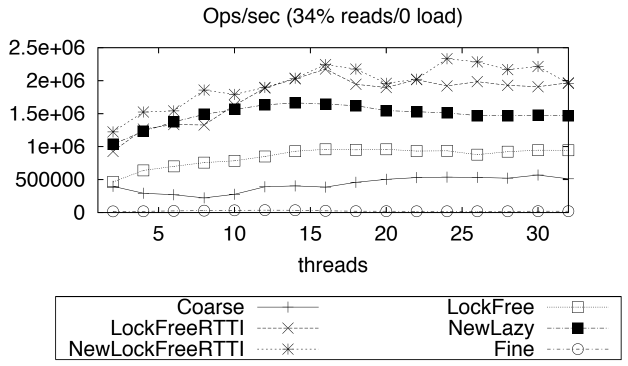

Fig. 9. The graph shows throughput as concurrency increases with a 34%, 33% and 33% ratio respectively of contains(), add(), and remove() method calls.

The graph of the 50% test on the righthand side of Figure 8 shows what happens when we drop the fraction of contains() method calls. As can be seen, the lock-free RTTI-based implementation of Michael's algorithm stays at about the same throughput level, yet the performance of the two new algorithms deteriorates because (1) the large number of additional add() method calls in the new version of Michael's algorithm incur cleanup contention (they fail attempts at cleaning up the same entries) and must restart their traversals, and (2) the add() method calls in the lazy list acquire more costly locks and fail validation at a much higher rate, forcing them to restart their traversals.

The bottom graph of Figure 8 shows the change in throughput for the case of 32 threads as the fraction of contains() method calls increases (maintaining the 9/1 ratio of add() and remove() method calls). As can be seen, from 50% and onward the two new algorithms outperform all others, and have more than twice their throughput at 90%. The choice of which algorithm to use, the new lazy list, or Michael's lock-free list with our new wait-free membership test, for typical search applications with a high fraction of memberships tests, seems to depend on the cost of implementing the equivalent of AtomicMarkableReference in a given system and language.

The graph in Figure 9 shows the change in throughput when running a benchmark with 34% `contains`() method calls, 33% `add`() method calls and 33% `remove`() method calls. Though this is not a typical search structure access pattern, we present it here to explore how the algorithms compare across a wider range of loads. As can be seen, the throughput of the lock-free RTTI based implementations drops slightly, and the performance of the lazy list drops more significantly. As before, this is due to the further increase in the number of costly lock acquisitions and of failed validations.

We conclude that even with higher `add`() and `remove`() method call rates than we expect in many applications, our results show how to improve on the performance of previous algorithms. Furthermore, without using any nonstandard language tricks, our new algorithms soundly beat previous ones.

4 Conclusions

We introduced the *lazy list*, a simple new concurrent list algorithm based on lazy marking and deletion of nodes. Perhaps the most substantial advantage of the new algorithm is a *wait-free* membership test operation, an operation that can readily replace membership tests in other list-based set algorithms such as Michael's lock-free lists [10].

Various optimizations to our algorithm are possible. As noted earlier, one can make the `add`() method more efficient by locking only the `pred` node. One can also add an optimization whereby threads "prevalidate" the state of an entry before acquiring the entry locks, thereby saving the cost of acquiring them upon failure.

Most importantly, we believe the algorithmic approach introduced in this paper, the combination of lazy lock-based list manipulation coupled with wait-free traversal, can lead to simpler and possibly more efficient algorithms for related data structures such as concurrent skip-lists and other search structures.

References

1. R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.
2. B. Eckel. *Thinking in Java (2nd Edition)*. Pearson Education, 2000.
3. T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.
4. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
5. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.
6. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

7. D. Lea. Personal communication.
8. D. Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns.* Addison-Wesley, second edition edition, 1999.
9. Corinne Maier. *Hello Laziness: Why Hard Work Doesn't Pay.* Orion, London, 2005. ISBN: 0752871862.
10. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.
11. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *The 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.
12. M. Moir and N. Shavit. *Chapter 47 – Concurrent Data Structures – Handbook of Data Structures and Applications.* Chapman and Hall/CRC, first edition edition, 2004.
13. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. Technical report, Microsoft Research, Cambridge, UK, 2005.
14. J. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.