Wait-Free Linked-Lists[⋆]

Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank

Dept. of Computer Science, Technion, Israel.
Email: {stimnat,anastas,sakogan,erez}@cs.technion.ac.il

Abstract. *Wait-freedom* is the strongest and most desirable progress guarantee, under which any thread must make progress when given enough CPU steps. Wait-freedom is required for hard real-time, and desirable in many other scenarios. However, because wait-freedom is hard to achieve, we usually settle for the weaker *lock-free* progress guarantee, under which one of the active threads is guaranteed to make progress. With lock-freedom (and unlike wait-freedom), starvation of all threads but one is possible.
The linked-list data structure is fundamental and ubiquitous. Lock-free versions of the linked-list are well known. However, whether it is possible to design a practical wait-free linked-list has remained an open question. In this work we present a practical wait-free linked-list based on the CAS primitive. To improve performance further, we also extend this design using the fast-path-slow-path methodology. The proposed design has been implemented and measurements demonstrate performance competitive with that of Harris's lock-free list, while still providing the desirable wait-free guarantee, required for real-time systems.

1 Introduction

A linked-list is one of the most commonly used data structures. The linked-list seems a good candidate for parallelization, as modifications to different parts of the list may be executed independently and concurrently. Indeed, parallel linked-lists with various progress properties are abundant in the literature. Among these are lock-free linked-lists. A lock-free data structure ensures that when several threads access the data structure concurrently, at least one makes progress within a bounded number of steps. While this property ensures general system progress, it does not prevent starvation of a particular thread, or of several threads. Wait-free data structures ensure that each thread makes progress within a bounded number of steps, regardless of other threads' concurrent execution. Wait-free data structures are crucial for real-time systems, where a deadline may not be missed even in a worst-case scenario. To allow real-time systems and other systems with critical worst-case demands make use of concurrent data structures, we must provide the strong wait-free guarantee. Furthermore, wait-freedom is a desirable progress property for many systems, and in particular operating systems, interactive systems, and systems with service-level guarantees. For all those, the elimination of starvation is highly desirable.

Despite the great practical need for data structures that ensure wait-freedom, almost no practical wait-free data structure is known, because data structures that ensure wait-freedom are notoriously hard to design. Recently, wait-free designs for the simple stack

[⋆] This work was supported by the Israeli Science Foundation grant No. 283/10.

and queue data structures appeared in the literature [7, 2]. Wait-free stack and queue structures are not easy to design, but they are considered less challenging as they present limited parallelism, i.e., a limited number of contention points (the head of the stack, and the head and the tail of the queue). We are not aware of any practical wait-free design for any other data structure that allows multiple concurrent operations to occur simultaneously. In particular, to the best of our knowledge, there is no wait-free linked-list algorithm available in the literature except for algorithms of universal constructions, which do not provide practical efficiency.

The main contribution of this work is a practical, linearizable, fast and wait-free linked-list. Our construction builds on the lock-free linked-list of Harris [4], and extends it using a helping mechanism to become wait-free. The main technical difficulty is making sure that helping threads perform each operation correctly, apply each operation exactly once, and return a consistent result (of success or failure) according to whether each of the threads completed the operation successfully. This task is non-trivial and it is what makes wait-free algorithms notoriously hard to design. Our design deals with several races that come up, and a proof of correctness makes sure that no further races exist. Some of our techniques may be useful in future work, especially the *success bit* introduced to determine the owner of a successful operation. Next, we extend our design using the fast-path-slow-path methodology of Kogan and Petrank [8], in order to make it even more efficient, and achieve performance that is almost equivalent to that of the lock-free linked-list of Harris. Here, the idea is to combine both lock-free and wait-free algorithms so that the (lock-free) fast path runs with (almost) no overhead, but is able to switch to the (wait-free) slow path when contention interferes with its progress. It is also important that both paths are able to run concurrently and correctly. Combining the newly obtained wait-free linked-list with the existing lock-free linked-list of Harris is an additional design challenge that is, again, far from trivial.

We have implemented the new wait-free linked-list and compared its efficiency with that of Harris's lock-free linked-list. Our first design(slightly optimized) performs worse by a factor of 1.5 when compared to Harris's lock-free algorithm. This provides a practical, yet not optimal, solution. However, the fast-path-slow-path extension reduces the overhead significantly, bringing it to just 2-15 percents. This seems a reasonable price to pay for obtaining a data structure with the strongest wait-free guarantee, providing non-starvation even in worst-case scenarios, and making it available for use with real-time systems.

We begin in Section 2 with an overview of the algorithm and continue in Section 3 with a detailed description of its most complex operation and crucial parts. Highlights of the correctness proof appear in Section 4. The linearization points of the algorithm are specified in Section 5. We give an overview of the fast-path-slow-path extension of the algorithm in Section 6, and Section 7 presents the performance measurements. In a full version of this work [11] we also provide details about the fast-path-slow-path implementation, the entire pseudo-code, and a full correctness proof for the algorithm.

1.1 Background and Related Work

The first lock-free linked-list was presented by Valois [12]. A simpler and more efficient lock-free algorithm was designed by Harris [4], and Michael [9] added a hazard-pointers mechanism to allow lock-free memory management for this algorithm. Fomitchev and Rupert achieved better theoretical complexity in [3]. Herlihy and Shavit implemented a variation of Harris's algorithm [6], and we used this implementation both for comparison and as the basis for the Java code we developed.

Wait-free queues were presented in [7, 2]. A different approach for building concurrent lock-free or wait-free data structures is the use of universal constructions [5, 6, 1]. However, universal constructions (at least for the linked-list) are not efficient enough to be applied in practice, and are often non-scalable.

Recently, Kogan and Petrank [8] presented the fast-path-slow-path technique mentioned above. We use the fast-path-slow-path methodology in this work to achieve an efficient and wait-free linked-list.

Our wait-free linked-list design follows the traditional practice, in which concurrent linked-list data structures realize a sorted list, where each key may only appear once in the list [3, 4, 6, 12]. A brief announcement of this work appeared in [10].

2 An Overview of the Algorithm

Before getting into the technical details (in Section 3) we provide an overview of the design. The wait-free linked-list supports three operations: INSERT, DELETE, and CONTAINS. All of them run in a wait-free manner. The underlying structure of the linked-list is depicted in Figure 2. Similarly to Harris's linked-list, our list contains sentinel `head` and `tail` nodes, and the `next` pointer in each node can be marked using a special `mark bit`, to signify that the entry in the node is logically deleted.

To achieve wait-freedom, our list employs a helping mechanism. Before starting to execute an operation, a thread starts by publishing an *Operation Descriptor*, in a special `state` array, allowing all the threads to view the details of the operation it is executing. Once an operation is published, all threads may try to help execute it. When an operation is completed, the result is reported to the state array, using a CAS which replaces the existing operation descriptor with one that contains the result.

A top-level overview of the insert and delete operations is provided in Figure 1. When a thread wishes to INSERT a key k to the list, it first allocates a new node with

```
boolean insert(key)
  Allocate a new node (without help)
  Publish the operation (without help)
  Search for a place to insert the node
    If key already exists, return with failure
  Direct the new node's next pointer
  Insert the node(by modifying its predecessor)
  Return with Success
```

```
boolean delete(key)
  Publish the operation (without help)
  Search for the victim node to delete
    If key doesn't exist, return with failure
  Announce the victim node in the state array
  Mark the victim's pointer to logically delete it
  Physically remove the victim node
  Report that the victim node has been removed
  Compete for success (without help)
```

Fig. 1. Insert and delete overview

key k, and then publishes an operation descriptor with a pointer to the new node. The rest of the operation can be executed by any of the threads in the system, and may also be run by many threads concurrently. Any thread that executes this operation starts by searching for a place to insert the new node. This is done using the search method, which, given a key *k*, returns a pair of pointers, *prev* and *curr*. The prev pointer points to the node with the highest key smaller than *k*, and the curr pointer points to the node with the smallest key larger than or equal to *k*. If the returned curr node holds a key equal to the key on the node to be inserted, then failure is reported. Otherwise the node should be inserted between prev and curr. This is done by first updating the new node's `next` pointer to point to curr, and then updating prev's `next` field to point to it. Both of these updates are done using a CAS to prevent race conditions, and the failure of any of these CASes will cause the operation to restart from the search method. Finally, after that node has been inserted, success is reported.

While the above description outlines the general process of inserting a node, the actual algorithm is a lot more complex, and requires care to avoid problematic races that can make things go wrong. In addition, there is also a potential ABA problem that requires the use of a version mark on the `next` pointer field[1]. We discuss these and other potential races in Section 3.4.

When a thread wishes to DELETE a key *k* from the list, it starts by publishing the details of its operation in the `state` array. The next steps can be then executed by any of the threads in the system until the last step, which is executed only by the thread that initiated the operation, denoted the *owner thread*. The DELETE operation is executed (or helped) in two stages. First, the *victim* node to be deleted is chosen. To do this, the search method is invoked. If no node with the key *k* is found, failure is reported. Otherwise, the victim node is *announced* in the `state` array. This is done by replacing the state descriptor that describes this operation to a state descriptor that has a pointer to the victim node. This announcement helps to ascertain that concurrent helping threads will not delete two different nodes, as the victim node for this operation is determined to be the single node that is announced in the operation descriptor. In the second stage, deletion is executed similarly to Harris's linked-list: the victim node's `next` field is marked, and then it is physically removed from the list. The victim node's removal is then reported back to the `state` array.

However, since multiple threads execute multiple operations, and as it is possible that several operations attempt to DELETE the same node, it is crucial that exactly one operation be declared as successfully deleting the node's key and that the others return failure. An additional (third) stage is required in order to consistently determine which operation can be considered successful. This step is executed only by the owner threads, and is given no help. The threads that initiated the concurrent delete operations compete among themselves for the ownership of the deletion. To this end, an extra `success-bit` designated for this purpose is added to each node in the list. The thread that successfully CASes this bit from false to true is the only one that reports success for this deletion. We believe that using an extra bit to determine an ownership of an operation is a useful mechanism for future wait-free constructions as well. This mechanism is further explained in Section 3.5.

[1] The versioning method provides a simple solution to the ABA problem. A more involved solution that does not require a versioned pointer appears in the full version of this paper [11].

The CONTAINS operation is much simpler than the other two. It starts by publishing the operation. Any helping thread will then search for it in the list, reporting success (on the operation record) if the key was found, or failure if it was not.

3 The Algorithm

In this section we present the details of the algorithm. We fully describe the list structure, the helping mechanism, and the SEARCH and INSERT operations. The INSERT operation is the most complicated part of the algorithm. A detailed description of the DELETE and CONTAINS operations appears in the full version of this paper [11]. We also include in this section a detailed description of the `success-bit` technique used in the DELETE operation, as we believe this mechanism can be useful for future work.

3.1 The Underlying Data Structures

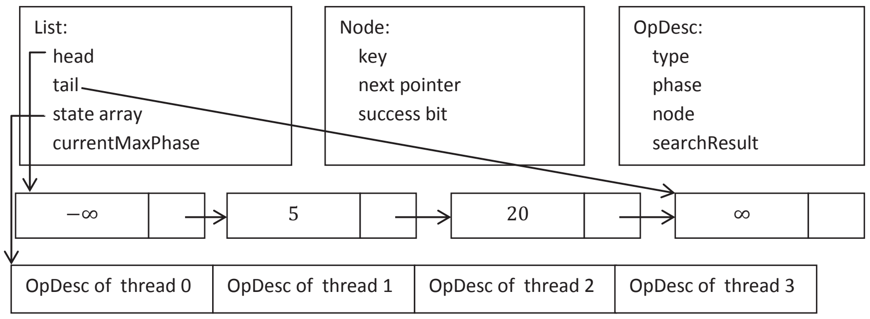

Fig. 2. General structure

The structure of the linked-list is depicted in Figure 2. A node of the linked list consists of three fields: a key, a `success bit` to be used when deleting this node, and a special pointer field. The special pointer field has its least significant bit used by the algorithm for signaling between threads. In addition, this pointer is versioned, in the sense that there is a counter associated with it (in an adjacent word) and each modification of it (or of its special bit) increments the counter. The modification and counter increment are assumed to be atomic. This can be implemented by squeezing all these fields into a single word, and limiting the size of the counter and pointer, or by using a double-word compare-and-swap when the platform allows. Alternatively, one can allocate a “pointer object” containing all these fields and bits, and then atomically replace the existing pointer object with a new one. The latter approach is commonly used with Java lock-free implementations, and we use it as well.

In addition to the nodes of the list, we also maintain an array with an operation-descriptor for each thread in the system. The OpDesc entry for each thread describes its current state. It consists of a phase field `phase`, the `OpType` field signifying which operation is currently being executed by this thread, a pointer to a node, denoted `node`, which serves the insert and delete operations, and a pair of pointers $(prev, curr)$, for recording the result of a search operation. Recall that the result of a SEARCH operation of a key, *k*, is a pair of pointers denoted *prev* and *curr*, as explained in Section 2 above.

The possible values for the operation type (OpType) in the operation descriptor state are:

insert	asking for help in inserting a node into the list.
search_delete	asking for help in finding a node with the key we wish to delete.
execute_delete	asking for help in marking a node as deleted (by tagging its next pointer) and unlinking it from the list.
contains	asking for help in finding out if a node with the given key exists.
success	operation was completed successfully.
failure	operation failed (deletion of a non-existing key or insertion of an existing key).
determine_delete	decide if a delete operation completed successfully.

The first four states in the above list are used to request help from other threads. The last three states indicate steps in the executions in which the thread does not require any help. The linked-list also contains an additional long field, `currentMaxPhase`, to support the helping mechanism, as described in Subsection 3.2.

3.2 The Helping Mechanism

Before a thread starts executing an operation, it first selects a phase number larger than all previously chosen phase numbers. The goal of assigning a phase number to each operation is to let new operations make sure that old operations receive help and complete before new operations are executed. This ensures non-starvation. The phase selection mechanism ensures that if operation O_2 arrives strictly later than operation O_1, i.e., O_1 receives a phase number before O_2 starts selecting its own phase number, then O_2 will receive a higher phase number. The phase selection procedure is executed in the MAXPHASE method depicted in Figure 3. Note that although a CAS is used in this method, the success of this CAS is not checked, thus preserving wait-freedom. If the CAS fails, it means that another thread increased the counter concurrently, which is sufficient for the phase numbering. After selecting a phase number, the thread publishes the operation by updating its entry in the `state` array. It then goes through the array, helping all operations with a phase number lower than or equal to its own. This ensures wait-freedom: a delayed operation eventually receives help from all threads and soon completes. See Figure 3 for the pseudo-code.

3.3 The Search Methods

The CONTAINS method, which is part of the data structure interface, is used to check whether a certain key is a part of the list. The SEARCH method is used (internally) by the INSERT, DELETE, and CONTAINS methods to find the location of a key and perform

```
private long maxPhase() {
  long result = currentMaxPhase.get();
  currentMaxPhase.compareAndSet
      (result, result+1);
  return result; }

private void help(long phase) {
  for (int i = 0; i < state.length(); i++) {
    OpDesc desc = state.get(i);
    if (desc.phase <= phase) {        ▷ help older op
      if (desc.type == OpType.insert) {
        helpInsert(i, desc.phase);
      } else if
      (desc.type == OpType.search_delete
      || desc.type == OpType.execute_delete) {
        helpDelete(i, desc.phase);
      } else if (desc.type == OpType.contains) {
        helpContains(i, desc.phase);
      } } } }

private boolean isSearchStillPending(int tid,
long ph) {
  OpDesc curr = state.get(tid);
  return (curr.type == OpType.insert ||
      curr.type == OpType.search_delete ||
      curr.type == OpType.execute_delete ||
      curr.type==OpType.contains) &&
    curr.phase == ph; }

private Window search(int key, int tid, long
phase) {
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false}; boolean snip;
  retry : while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
                    ▷ Reading both the ref and the mark:
      succ = curr.next.get(marked);
      while (marked[0]) {            ▷ logically deleted
                    ▷ Attempt to physically remove curr:
        snip = pred.next.compareAndSet
            (curr, succ, false, false);
        if (!isSearchStillPending(tid,phase))
          return null;          ▷ to ensure wait-freedom.
        if (!snip) continue retry;        ▷ list changed
        curr = succ;                      ▷ advance curr
        succ = curr.next.get(marked);     ▷ and succ
      }
      if (curr.key >= key)                ▷ window found
        return new Window(pred, curr);
      pred = curr; curr = succ;           ▷ advance both
    }
  }
}
```

Fig. 3. The help and search methods

some maintenance during the search. It is actually nearly identical to the original lock-free SEARCH method. The SEARCH method takes a key and returns a pair of pointers denoted *window*: `pred`, which points to the node containing the highest key less than the input key, and `curr`, which points to the node containing the lowest key higher than or equal to the requested key. When traversing through the list, the SEARCH method attempts to physically remove any node that is logically deleted. If the remove attempt fails, the search is restarted from the head of the list. This endless attempt to fix the list seems to contradict wait-freedom, but the helping mechanism ensures that these attempts eventually succeed. When an operation delays long enough, all threads reach the point at which they are helping it. When that happens, the operation is guaranteed to succeed. The SEARCH operation will not re-iterate if the operation that executes it has completed, which is checked using the ISSEARCHSTILLPENDING method. If the associated operation is complete, then the SEARCH method returns a null. The pseudo-code for the search method is depicted in Figure 3.

3.4 The Insert Operation

Designing operations for a wait-free algorithm requires dealing with multiple threads executing each operation, which is substantially more difficult than designing a lock-free operation. In this section, we present the insert operation and discuss some of the races that occur and how we handle them. The basic idea is to coordinate the execution of all threads using the operation descriptor. But more actions are required, as explained below. Of-course, a proof is required to ensure that all races have been handled. The pseudo-code of the INSERT operation is provided in Figure 4. The thread that initiates the operation is denoted *the operation owner*. The *operation owner* starts the INSERT method by selecting a phase number, allocating a new node with the input key, and installing a link to it in the `state` array.

Next, the thread (or any helping thread) continues by searching the list for a location where the node with the new key can be inserted (Line 17 in the method HELPINSERT). In the original lock-free linked-list, finding a node with the same key is interpreted as failure. However, in the presence of the helping mechanism, it is possible that some other thread that is helping the same operation has already inserted the node but has not yet reported success. It is also possible that the node we are trying to insert was already inserted and then deleted, and then a different node, with the same key, was inserted into the list. To identify these cases, we check the node that was found in the search. If it is the same node that we are trying to insert, then we know that success should be reported. We also check if the (`next` field of the) node that we are trying to insert is *marked* for deletion. This happens if the node was already inserted into the list and then removed. In this case, we also report success. Otherwise, we attempt to report failure. If there is no node found with the same key, then we can try to insert the node between `pred` and `curr`. But first we check to see if the node was already inserted and deleted (line 35), in which case we can simply report success.

The existence of other threads that help execute the same operation creates various races that should be properly handled. One of them, described in the next paragraph, requires the INSERT method to proceed with executing something that may seem redundant at first glance. The INSERT method creates a state descriptor identical to the existing one and atomically replaces the old one with the new one (Lines 42–45). The replacement foils all pending CAS operations by other threads on this state descriptor, and avoids confusion as to whether the operation succeeds or fails. Next, the method executes the actual insertion of the node into the list (Lines 46–48) and it attempts to report success (Lines 49–52). If any of the atomic operations fail, the insertion starts from scratch. The actual insertion into the list (Lines 46–48) is different from the insertion in the original lock-free linked-list. First, the `next` pointer in the new node is not privately set, as it is now accessible by all threads that help the insert operation. It is set by a CAS which verifies that the pointer has not changed since before the search. Namely, the old value is read in Line 16 and used as the expected value in the CAS of Line 46. This verification avoids another race, which is presented below. Moreover, the atomic modification of the `next` pointer in the previous node to point to the inserted node (Lines 47–48) uses the version of that `next` pointer to avoid the ABA problem. This is also justified below.

Let us first present the race that justifies the (seemingly futile) replacement of the state descriptor in Lines 42–45. Suppose Thread T_1 is executing an INSERT operation of a key k. T_1 finds an existing node with the key k and is about to report failure. T_1 then gets stalled for a while, during which the other node with the key k is deleted and

```
public boolean insert(int tid, int key) {
    long phase = maxPhase();                                   ▷ getting the phase for the op
    Node newNode = new Node(key);                              ▷ allocating the node
    OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
    state.set(tid, op);                                        ▷ publishing the operation
    help(phase);        ▷ when finished - no more pending operation with lower or equal phase
    return state.get(tid).type == OpType.success;
}

private void helpInsert(int tid, long phase) {
    while (true) {
        OpDesc op = state.get(tid);
        if (!(op.type == OpType.insert && op.phase == phase))
            return;                                            ▷ the op is no longer relevant, return
        Node node = op.node;                                   ▷ getting the node to be inserted
        Node node_next = node.next.getReference();
        Window window = search(node.key,tid,phase);
        if (window == null)                                    ▷ operation is no longer pending
            return;
        if (window.curr.key == node.key) {                     ▷ chance of a failure
            if ((window.curr==node)||(node.next.isMarked())){  ▷ success
                OpDesc success =
                    new OpDesc(phase, OpType.success, node, null);
                if (state.compareAndSet(tid, op, success))
                    return;
            }
            else {                                             ▷ the node was not yet inserted - failure
                OpDesc fail=new OpDesc(phase,OpType.failure,node,null);
                                        ▷ the following CAS may fail if search results are obsolete:
                if (state.compareAndSet(tid, op, fail))
                    return;
            }
        }
        else {
            if (node.next.isMarked()){                         ▷ already inserted and deleted
                OpDesc success =
                    new OpDesc(phase, OpType.success, node, null);
                if (state.compareAndSet(tid, op, success))
                    return;
            }
            int version = window.pred.next.getVersion();       ▷ read version.
            OpDesc newOp=new OpDesc(phase,OpType.insert,node,null);
                                        ▷ preventing another thread from reporting a failure:
            if (!state.compareAndSet(tid, op, newOp))
                continue;                      ▷ operation might have already reported as failure
            node.next.compareAndSet(node_next,window.curr,false,false);
            if (window.pred.next.compareAndSet
                    (version, node.next.getReference(), node, false, false)) {
                OpDesc success =
                    new OpDesc(phase, OpType.success, node, null);
                if (state.compareAndSet(tid, newOp, success))
                    return;
            }
        }
    }
}
```

Fig. 4. The insert operation

a different thread, T_2, helping the same INSERT operation that T_1 is executing, does find a proper place to insert the key k, and does insert it, but at that point T_1 regains control and changes the descriptor state to erroneously report failure. This sequence of events is bad, because a key has been inserted but failure has been reported. To avoid such a scenario, upon finding a location to insert k, T_2 modifies the operation descriptor to ensure that no stalled thread can wake up and succeed in writing a stale value into the operation descriptor.

Next, we present a race that justifies the setting of the `next` pointer in the new node (Line 46). The INSERT method verifies that this pointer has not been modified since it started the search. This is essential to avoid the following scenario. Suppose Thread T_1 is executing an INSERT of key k and finds a place to insert the new node N in between a node that contains $k-1$ and a node that contains $k+2$. Now T_1 gets stalled for a while and T_2, helping the same INSERT operation, inserts the node N with the key k , after which it also inserts another new node with key $k+1$, while T_1 is stalled. At this point, Thread T_1 resumes without knowing about the insertion of these two nodes. It modifies the next pointer of N to point to the node that contains $k+2$. This modification immediately foils the linked-list because it removes the node that contains $k+1$ from the list. By making T_1 replace the `next` field in N atomically only if this field has not changed since before the search, we know that there could be no node between N and the node that followed it at the time of the search.

Finally, we justify the use of a version for the `next` pointer in Line 47, by showing an ABA problem that could arise when several threads help executing the same insert operation. Suppose Thread T_1 is executing an INSERT of the key k into the list. It searches for a location for the insert, finds one, and gets stalled just before executing Line 47. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 47 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. This is a classical ABA problem, and we solve it using versioning of the next pointer. The version is incremented each time the `next` pointer is modified. Therefore, the insertion and deletion of a different k key while T_1 is stalled cannot go unnoticed.

3.5 The Success Bit Technique

Helping DELETE is different from helping INSERT in the sense that the help method in this case does not execute the entire DELETE operation to its completion. Instead, it stops before determining the success of the operation, and lets the operation owner decide whether its operation was successful. Note that this does not foil wait-freedom, as the operation owner will never get stuck on deciding whether the operation was successful. When the help method returns, there are two possibilities. The simpler possibility is that the requested key was not found in the list. Here it is clear that the operation failed and in that case the state is changed by the helper to a failure and the operation

```
 1: public boolean delete(int tid, int key) {
 2:    long phase = maxPhase();                        ▷ getting the phase for the op
 3:    state.set(tid, new OpDesc
 4:    (phase, OpType.search_delete, new Node(key),null));             ▷ publishing
 5:    help (phase);     ▷ when finished - no more pending operation with lower or equal phase
 6:    OpDesc op = state.get(tid);
 7:    if (op.type == OpType.determine_delete)
 8:                            ▷ Need to compete on the ownership of deleting this node:
 9:       return op.searchResult.curr.success.compareAndSet(false, true);
10:    return false;
11: }
```

Fig. 5. The delete method

can terminate. The other possibility is that the requested key was found and deleted. In this case, it is possible that several DELETE operations for the same key were run concurrently by several operation owners and by several helping threads. As the delete succeeded, it has to be determined which operation owner succeeded. In such a case there are several operation owners for the deletion of the key *k* and only one operation owner can return success, because a single DELETE has been executed. The others operation owners must report failure. This decision is made by the operation owners (and not by the helping threads) in Line 9 of the DELETE method itself, depicted in Figure 5. It employs a designated `success bit` in each node. Whoever sets this bit becomes the owner of the deletion for that node in the list and can report success. We believe that this technique for determining the success of a thread in executing an operation in the presence of helping threads can be useful in future constructions of wait-free algorithms.

3.6 Memory management

The algorithm in this work relies on a garbage collector (GC) for memory management. A wait-free GC does not currently exist. This is a common difficulty for wait-free algorithms. A frequently used solution, which suits this algorithm as well, is Michael's Hazard Pointers technique [9]. Hazard pointers can be used for the reclamation of the operation descriptors as well, and not only for the reclamation of the list nodes themselves.

4 Highlights of the Correctness Proof

We now briefly explain how this algorithm is proven correct. A full proof appears in the full version of this paper [11]. A full proof is crucial for a parallel algorithm as without it, one can never be sure that additional races are not lurking in the algorithm.

Basic Concepts and Definitions. The *mark bit*, is the bit on the next field of each node, and it is used to mark the node as logically deleted. A node can be marked or unmarked according to the value of this bit. We define the nodes that are *logically in the list* to be the unmarked nodes that are reachable from the list's head. Thus, a *logical change* to the list, is a change to the set of unmarked nodes reachable from the head. We say that a node is an *infant node* if it has never been reachable from the head. These are nodes that have been prepared for insertions but have not been inserted yet.

In the proof we show that at the linearization point of a successful insert, the inserted value becomes logically in the list and that at a linearization point of a successful delete, a node with the given value is logically deleted from the list. To show this, we look at the actual *physical* modifications that may occur to the list.

Proof Structure. One useful invariant is that a *physical change* to the list can only modify the node's next field, as a node's key is final and never changes after the initialization of a node. A second useful invariant is that a marked node is never unmarked, and that it's next field never changes (meaning, it will keep pointing to the same node). This is ascertained by examining all the code lines that change a node's next field, and noting that all of them do it using a CAS which prevents a change from taking effect if the node is marked. We next look at all possible physical changes to a node's next field, and show that each of them falls in one of the following four categories:

* Marking: changing the mark bit of a node that is *logically in the list* to true.
* Snipping: physically removing a *marked* node out of the list.
* Redirection: a modification of an infant node's next pointer (in preparation for its insertion).
* Insertion: a modification of a non-infant node to point to an infant node (making the latter non-infant after the modification).

Proving that every *physical change* to a node's next field falls into one of the four categories listed above, is the most complicated part of the formal proof, and is done by induction, with several intermediate invariants. Finally, it is shown that any operation in the *marking* category matches a successful delete operation and any operation in the *insertion* category matches a successful insert operation. Thus, at the proper linearization points the linked list changes according to its specification. Furthermore, it is shown that physical operations in the *Redirection* and *Snipping* categories cause no *logical* changes to the list, which completes the linearizability proof.

To show wait-freedom, we claim that the helping mechanism ensures that a limited number of concurrent operations can be executed while a given insert or delete execution is pending. At the point when this number is exhausted, all threads will help the pending operation, and then it will terminates within a limited number of steps.

5 Linearization Points

In this section we specify the linearization point for the different operations of the linked-list. The SEARCH method for a key k returns a pair of pointers, denoted *pred* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. The linearization point of the SEARCH method is when the pointer that connects *pred* to *curr* is read. This can be either at Line 36 or 45 of the SEARCH method. Note that *curr*'s `next` field will be subsequently read, to make sure it is not *marked*. Since it is an invariant of the algorithm that a marked node is never unmarked, it is guaranteed that at the linearization point both *pred* and *curr* nodes were unmarked.

The linearization point for a CONTAINS method is the linearization point of the appropriate SEARCH method. The appropriate SEARCH method is the one called by the thread that subsequently successfully reports the result of the same CONTAINS operation. The linearization point of a *successful insert* is in Lines 47-48 (together they are a single instruction) of the *helpInsert* method. This is the CAS operation that physically links the node into the list. For a *failing insertion*, the linearization point is the linearization point of the SEARCH method executed by the thread that reported the failure.

The linearization point of a *successful delete* is at the point where the node is *logically* deleted, which means successfully marked. Note that it is possible that this is executed by a helping thread and not necessarily by the operation owner. Furthermore, the helping thread might be trying to help a different thread than the one that will eventually own the deletion. The linearization point of an *unsuccessful delete* is more complex. A delete operation may fail when the key is properly deleted, but a different thread is selected as the owner of the delete. In this case, the current thread returns failure, because of the failure of the CAS of the DELETE method (at Line 9). In this case, the linearization point is set to the point when the said node is logically deleted (marked). The linearization point of an *unsuccessful delete*, originating from simply not finding the key, is the linearization point of the SEARCH method executed by the thread that reported the failure.

6 The Fast-Path-Slow-Path Variation

The idea behind the fast-path-slow-path [8] approach is to combine a (fast) lock-free algorithm with a (slower) wait-free one. The lock free algorithm provides a basis for a fast path and we use Harris's lock-free linked-list for this purpose. The execution in the fast path begins by a check whether a help is required for any operation in the slow path. Next, the execution proceeds with running the fast lock-free version of the algorithm while counting the number of contentions that end with a failure (i.e., failed CASes). Typically, few failures occur and help is not required, and so the execution terminates after running the faster lock-free algorithm. If this fast path fails to make progress, the execution moves to the slow path, which runs the slower wait-free algorithm described in Section 3, requesting help (using an operation descriptor in its slot in the state array) and making sure the operation eventually terminates.

The number of CAS failures allowed in the fast path is limited by a parameter called MAX_FAILURES. The help is provided by threads running both the fast and slow path, which ensures wait-freedom: if a thread fails to complete its operation, its request for help is noticed both in the fast and in the slow path. Thus, eventually all other threads help it and its operation completes. However, help is not provided as intensively as described in Section 3. We use the *delayed help* mechanism, by which each thread only offers help to other threads once every several operations, determined by a parameter called HELPING_DELAY.

Combining the fast-path and the slow-path is not trivial, as care is needed to guarantee that both paths properly run concurrently. On top of other changes, it is useful to note that the DELETE operation must compete on the `success-bit` even in the fast-path, to avoid a situation where two threads running on the two different paths both think they were successful in deleting a node. The full implementation of the fast-path-slow-path variation of the linked-list is described in [11].

7 Performance

Implementation and platform. We compared four Java implementations of the linked-list. The first is the lock-free linked-list of Harris, denoted *LF*, as implemented by Herlihy and Shavit in [6]. (This implementation was slightly modified to allow nodes with user-selected keys rather than the object's hash-code. We also did not use the *item* field.)

The basic algorithm described in Section is denoted *WF-Orig* in the graphs below. A slightly optimized version of it, denoted *WF-Opt*, was changed to employ a delayed help mechanism, similar to the one used in the fast-path- slow-path extension. This means that a thread helps another thread only once every *k* operations, where *k* is a parameter of the algorithm set to 3. The idea is to avoid contention by letting help arrive only after the original thread has a reasonable chance of finishing its operation on its own. This optimization is highly effective, as seen in the results. Note that delaying help is not equivalent to a fast-path-slow-path approach, because all threads always ask for help (there is no fast path). All the operations are still done in the *helpInsert* and *helpDelete* methods.

The fast-path-slow-path algorithm, denoted *FPSP*, was run with the HELPING_DELAY parameter set to 3, and MAX_FAILURES set to 5. This algorithm combines the new wait-free algorithm described in this paper with Harris's lock-free algorithm, to achieve both good performance and the stronger wait-freedom progress guarantee.

We ran the tests in two environments. The first was a SUN's Java SE Runtime, version 1.6.0 on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores). The second was a SUN FIRE machine with an UltraSPARC T1 8 cores each running four hyper-threads.

Workload and methodology. In the micro-benchmarks tested, we ran each experiment for 2 seconds, and measured the overall number of operations performed by all the threads during that time. Each thread performed 60% CONTAINS, and 20% INSERT and DELETE operations, with keys chosen randomly and uniformly in the range $[1, 1024]$. The number of threads ranges from 1-16 (in the Intel(R) Xeon(R)) or from 1-32 (In the UltraSPARC). We present the results in Figure 6. The graphs show the total number of operations done by all threads in thousands for all four implementations, as a function of the number of threads. In all the tests, we executed each evaluation 8 times, and the averages are reported in the figures.

Results. It can be seen that the fast-path-slow-path algorithm is almost as fast as the lock-free algorithm. On the Intel machine, the two algorithms are barely distinguishable; the difference in performance is 2-3%. On the UltraSPARC the fast-path-slow-path suffers a noticeable (yet, reasonable) overhead of 9-14%. The (slightly optimized) basic wait-free algorithm is slower by a factor of 1.3–1.6, depending on the number of threads. Also, these three algorithms provide an excellent speed up of about 7 when working with 8 threads (on both machines), and about 24 when working with 32 multi-threads on the UltraSPARC. The basic non-optimized version of the wait-free algorithm doesn't scale as well. There, threads often work together on the same operation, causing a deterioration in performance and scalability. The simple delayed-help optimization enables concurrency without foiling the worst-case wait-freedom guarantee.

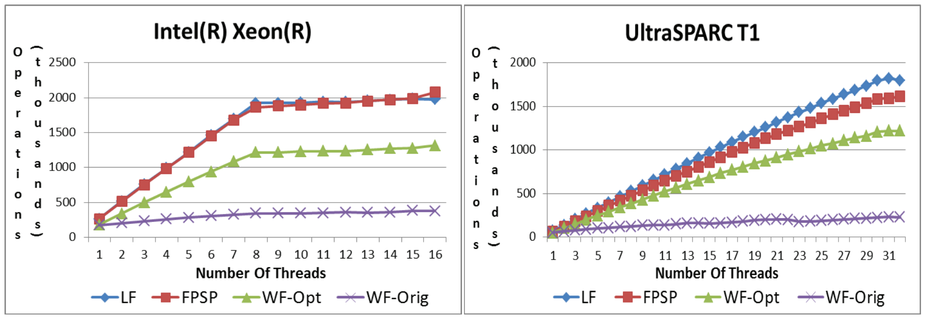

Fig. 6. The number of operations done in two seconds as a function of the number of threads

References

1. Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *SPAA*, pages 335–344, 2010.
2. Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334, 2011.
3. Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, New York, NY, USA, 2004. ACM.
4. Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, London, UK, UK, 2001. Springer-Verlag.
5. Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.
6. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
7. Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPOPP*, pages 223–234, 2011.
8. Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, pages 141–150, 2012.
9. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
10. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *PPOPP*, pages 309–310, 2012.
11. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. http://www.cs.technion.ac.il/%7eerez/%50apers/wfll-full.pdf. 2012.
12. John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, New York, NY, USA, 1995. ACM.