Practical Non-blocking Unordered Lists

Kunlong Zhang, Yujiao Zhao, Yajun Yang

Tianjin University

Yujie Liu, and Michael Spear

Lehigh University

Department of Computer Science and Engineering
Lehigh University

Technical Report LU-CSE-13-003

Abstract

This paper introduces new lock-free and wait-free unordered linked list algorithms. The composition of these algorithms according to the fast-path-slow-path methodology, a recently devised approach to creating fast wait-free data structures, is nontrivial, suggesting limitations to the applicability of the fast-path-slow-path methodology. The list algorithms introduced in this paper are shown to scale well across a variety of benchmarks, making them suitable for use both as standalone lists, and as the foundation for wait-free stacks and non-resizable hash tables.

1 Introduction

Linked lists are fundamental data structures that are widely used both on their own and as building blocks for other data structures. While a sequential linked list is easy to implement, concurrent linked lists that achieve both strong progress guarantees and good performance are challenging to design [3, 7–9, 16, 19, 22, 24]. Herlihy [10] demonstrated the existence of universal constructions for wait-free concurrent objects, yet it remains an open problem whether all such objects can be made practical: wait-free data structures implemented from universal constructions [4, 6, 11] tend to incur significant overhead, increased time and space complexity, and/or static bounds on the size of the data structure. Although many lock-free concurrent implementations [5,12,20,21] have been proposed for sequential data structures, practical wait-free versions are relatively rare [14, 23].

We introduce the first practical implementation of an unordered linked list that supports *wait-free* insert, remove, and lookup operations. The implementation is linearizable [13] and uses only a single-word compare-and-swap (CAS) primitive. Furthermore, the implementation does not require marking the lower bits of pointers [8]. Our implementation is built from a novel lock-free unordered list algorithm, where each insert and remove operation first linearizes by appending an intermediate “request” node at the head of the list, followed by a lazy search phase that computes the return value of the operation (which depends on whether the key value is already in the set); lookup operations have no side-effects on the shared memory. The implementation achieves scalable wait-freedom by adapting a technique originally designed for wait-free queues [14], and to further improve performance, we applied a recently-devised fast-path-slow-path methodology [15] to construct adaptive variants of our algorithm.

In this paper, we introduce the first practical wait-free unordered linked list, which is immediately usable in applications as-is, and can be employed in the creation of wait-free non-resizable hash tables and stacks. We discuss our experience and findings in applying the fast-path-slow-path methodology, identifying both strengths and limitations of the approach. In Section 2, we present background and related work. In Section 3 we present a lock-free unordered list algorithm that serves as the basis for the wait-free algorithm discussed in Section 4. We evaluate performance in Section 5. Section 6 concludes with guidelines for using the fast-path-slow-path methodology.

2 Related Work

The first lock-free list to require only atomic compare-and-swap (CAS) operations was developed by Valois [24], who employed a technique in which auxiliary nodes encoded in-progress operations. Harris [8] implemented a lock-free ordered list by using a pointer marking technique, in which a node is logically deleted by marking the least significant bit of its next pointer; the node is then physically removed from the list in a separate phase. Michael [16] improved memory reclamation in the Harris algorithm using hazard pointers [17]. Heller et al. [9] designed a lock-based linked list with wait-free lookup operations. Their wait-free technique can also be incorporated into the Harris-Michael algorithm to improve performance. Kogan and Petrank [14] proposed a wait-free queue implementation and a more efficient variant based on the fast-path-slow-path methodology [15] which composes the slower wait-free algorithm with a faster lock-free implementation [18]. Timnat et al. [23] designed a wait-free ordered linked list based on the fast-path-slow-path methodology, using the Harris-Michael algorithm as its fast path.

Subsequent efforts have contributed to our general understanding of lock-free list implementations, but have neither improved progress guarantees nor delivered superior performance to that attainable by combining the Harris, Michael, and Heller techniques. Fomitchev and Ruppert [7] presented a lock-free list with worst-case linear amortized cost. Attiya and Hillel [1] presented a lock-free doubly-linked list that relies on a double-compare-and-swap (DCAS) operation. Sundell and Tsigas [22] presented a lock-free doubly-linked list using only CAS. Braginsky and Petrank [2] presented the first lock-free unrolled linked list.

Herlihy [10, 11] presented the first universal construction to convert sequential objects to wait-free concurrent implementations. Fatourou and Kallimanis [6] provided a universal construction that can be used to implement highly efficient stacks and queues.

3 A Lock-free Unordered List

We now present a lock-free unordered list algorithm, which serves as the basis for our wait-free implementation. The algorithm implements a set object, where the elements can be compared using an equality operator $(=)$, even if they can not be totally ordered.

The list supports three operations: $\text{INSERT}(k)$ attempts to insert value k into the set and returns true (success) if k was not present in the set, and returns false otherwise. $\text{REMOVE}(k)$ returns true if it successfully removes value k from the set and returns false if k does not exist in the set. $\text{CONTAINS}(k)$ indicates whether k is contained by the set.

3.1 Overview

Figure 1 presents the basic algorithm. The list is comprised of NODE objects, where each NODE stores a key value, a $next$ pointer to the successor node, and a $state$ field for coordinating concurrent operations. The $prev$ and tid fields are reserved for the wait-free algorithm (Section 4). We maintain a global pointer $head$ that points to the first element of the list. Elements are always inserted at the head position.

The key insight of the algorithm is to maintain a refinement mapping function that maps a linked list object (starting from node h) to an abstract set object AbsSet(h):

$$\text{AbsSet}(h) \equiv \begin{cases} \emptyset & \text{if } h = \textbf{nil} \\ \text{AbsSet}(h.next) & \text{if } h.state = \textit{INV} \\ \text{AbsSet}(h.next) \cup \{h.key\} & \text{if } h.state = \textit{INS} \vee h.state = \textit{DAT} \\ \text{AbsSet}(h.next) \setminus \{h.key\} & \text{if } h.state = \textit{REM} \end{cases}$$

To maintain this property, an INSERT or REMOVE operation first places a node with an intermediate state (*INS* or *REM*) at the head of the list. Then it searches the list for the value being inserted or removed, removing logically deleted nodes along the way. Finally, it sets the intermediate node to a final state (*DAT* or *INV*).

In more detail, an INSERT operation allocates an *INS* node (h) and links it to the head of the list by invoking ENLIST (lines 2 - 3). It then invokes HELPINSERT (line 4) to determine whether the insertion is effective, that is, to check whether the key is already present in the set. The return value of HELPINSERT dictates the return value of the INSERT operation, as well as the final state of h (line 5): if the key was absent from the set, $h.state$ is set to *DAT*, and the insertion becomes effective; otherwise, $h.state$ is set to *INV*, indicating that the insertion failed due to the key already being present in the set, and h becomes a garbage node that will be physically removed by some subsequent operation. The update of $h.state$ must use a CAS instruction (line 5), since a concurrent REMOVE that deletes the same key may attempt to change $h.state$ concurrently. If the CAS fails, it means the key was deleted concurrently and the thread will invoke HELPREMOVE (lines 6 - 7) to help the deleting thread to clean up the list.

Similarly, a REMOVE operation starts by inserting a *REM* node at the head position (lines 10 - 11). The real work of removal is delegated to the HELPREMOVE operation (line 12), which traverses the list to delete the specified key and returns a boolean value indicating whether the key was found (and deleted). Then node h is set to the *INV* state (line 13), allowing some subsequent operation to remove it from the list.

The CONTAINS operation has no side effect on shared memory (it is read-only). The operation traverses the list to find the specified key and skips any *INV* nodes (lines 18 - 20). If a non-*INV* node with the specified key is encountered, the operation returns true (found) if the node is in state *DAT* or *INS* (line 21). Otherwise, the node is in *REM* state, which represents a REMOVE operation that can be thought of as having already deleted the key from the suffix of the list, and hence, the CONTAINS operation immediately returns false.

3.2 ENLIST Operation

Both INSERT and REMOVE use the ENLIST operation to insert a node at the head position. In the lock-free algorithm, ENLIST repeatedly performs a CAS operation (line 28), attempting to change $head$ to point to h, until the CAS succeeds. However, this approach fails to provide *wait-freedom*: since the CAS operation at line 28 of a specific thread may fail repeatedly, for an unbounded number of times (due to contention), the thread may starve in the ENLIST operation and make no progress. In Section 4, we introduce a wait-free ENLIST implementation, and show the algorithm can be made wait-free without any change to the other parts.

```
datatype NODE
  $key$    : $\mathbb{N}$      // integer data field
  $state$  : $\mathbb{N}$      // $INS$, $REM$, $DAT$, or $INV$
  $next$   : NODE   // pointer to the successor
  $prev$   : NODE   // pointer to the predecessor
  $tid$    : $\mathbb{N}$      // thread id of the creater

global variables
  $head$   : NODE   // initially nil

1  function INSERT($k : \mathbb{N}$) : $\mathbb{B}$
2    $h \leftarrow$ new NODE$\langle k, INS,$ nil, nil, threadid$\rangle$
3    ENLIST($h$)
4    $b \leftarrow$ HELPINSERT($h, k$)
5    if $\neg CAS(\&h.state, INS, (b ?\ DAT : INV))$ then
6      HELPREMOVE($h, k$)
7      $h.state \leftarrow INV$
8    return $b$

9  function REMOVE($k : \mathbb{N}$) : $\mathbb{B}$
10   $h \leftarrow$ new NODE$\langle k, REM,$ nil, nil, threadid$\rangle$
11   ENLIST($h$)
12   $b \leftarrow$ HELPREMOVE($h, k$)
13   $h.state \leftarrow INV$
14   return $b$

15 function CONTAINS($k : \mathbb{N}$) : $\mathbb{B}$
16   $curr \leftarrow head$
17   while $curr \neq$ nil do
18     if $curr.key = k$ then
19       $s \leftarrow curr.state$
20       if $s \neq INV$ then
21         return $(s = INS) \vee (s = DAT)$
22     $curr \leftarrow curr.next$
23   return false

24 procedure ENLIST($h$ : NODE)
25   while true do
26     $old \leftarrow head$
27     $h.next \leftarrow old$
28     if $CAS(\&head, old, h)$ then
29       return

30 function HELPINSERT($h$ : NODE, $k : \mathbb{N}$) : $\mathbb{B}$
31   $pred \leftarrow h$
32   $curr \leftarrow pred.next$
33   while $curr \neq$ nil do
34     $s \leftarrow curr.state$
35     if $s = INV$ then
36       $succ \leftarrow curr.next$
37       $pred.next \leftarrow succ$
38       $curr \leftarrow succ$
39     else if $curr.key \neq k$ then
40       $pred \leftarrow curr$
41       $curr \leftarrow curr.next$
42     else if $s = REM$ then
43       return true
44     else if $(s = INS) \vee (s = DAT)$ then
45       return false
46   return true

47 function HELPREMOVE($h$ : NODE, $k : \mathbb{N}$) : $\mathbb{B}$
48   $pred \leftarrow h$
49   $curr \leftarrow pred.next$
50   while $curr \neq$ nil do
51     $s \leftarrow curr.state$
52     if $s = INV$ then
53       $succ \leftarrow curr.next$
54       $pred.next \leftarrow succ$
55       $curr \leftarrow succ$
56     else if $curr.key \neq k$ then
57       $pred \leftarrow curr$
58       $curr \leftarrow curr.next$
59     else if $s = REM$ then
60       return false
61     else if $s = INS$ then
62       if $CAS(\&curr.state, INS, REM)$ then
63         return true
64     else if $s = DAT$ then
65       $curr.state \leftarrow INV$
66       return true
67   return false
```

Figure 1: A Lock-free Unordered List

3.3 Coordination Protocol

The core protocol of coordinating concurrency is encapsulated by the HELPINSERT and HELPREMOVE operations. The two operations share a similar code structure: each takes a pointer parameter h, which points to the node inserted by the prior ENLIST operation. In both operations, the thread traverses the list starting from h, and reacts to the different types of nodes it encounters.

As a common obligation of both operations, logically deleted nodes are purged during the traversal (lines 35 - 38 and lines 52 - 55). That is, once an *INV* node is encountered (pointed to by $curr$), the node is physically removed from the list by setting the predecessor's next pointer to the successor of $curr$. Note that since new nodes cannot be added to the list at any point other than the head, the problems that plague node removal in sorted lists do not apply. In particular, it is not possible that removing one node can inadvertently lead to a new arrival disappearing from the list. While it is possible for a removed node to re-appear in the list on account of conflicting writes to the next pointer, such a node will necessarily already be marked *INV*, and thus there will be no impact on the correctness of the list.

During the traversal, the $curr$ node is skipped if $curr.key \neq h.key$ (lines 39 - 41 and 56 - 58). Otherwise, we say the $curr$ node is a "related node" with respect to the current operation. There are three possibilities if $curr$ is a related node: $curr$ is a *DAT* node, an *INS* node, or a *REM* node. In the latter two cases, the related node was created by some concurrent INSERT or REMOVE operation. We call such operations "related operations".

In HELPINSERT, if a related *REM* node is encountered, there is a concurrent REMOVE operation finalizing a removal of the same key. Hence, the HELPINSERT returns true (success) immediately (lines 42 - 43), since the concurrent REMOVE operation ensures that the key is absent in the set. Otherwise (lines 44 - 45), if the related node is an *INS* node, then the related INSERT operation inserted the same key earlier (or is determining that the key already exists in the list) and the HELPINSERT operation must return false. Finally, if the related node is a *DAT* node, HELPINSERT returns false since the key already exists in the set.

In HELPREMOVE, if a related *REM* node is found (lines 59 - 60), the operation returns false immediately since the key was already deleted by a concurrent REMOVE operation. If the related node is an *INS* node (lines 61 - 63), then the key was inserted by a concurrent INSERT operation. In this case, the thread attempts to change the node from *INS* to *REM* (line 62); a CAS instruction is needed to prevent data races on the $state$ field (i.e., line 5). In the last case, the related node is a *DAT* node, meaning that the key is in the set, and the node is deleted by setting its $state$ to *INV* (line 65).

3.4 Lock-freedom

To show that the algorithm is lock-free, we show that *some* operation completes when any thread executes a bounded number of local steps. We first notice that the ENLIST operation is lock-free: a thread's CAS at line 28 may fail only due to another thread performing a CAS and completing its ENLIST operation. Since ENLIST is invoked exactly once in each INSERT and REMOVE, for n threads, at least one list operation will complete if some thread fails the CAS for n times in its ENLIST operation.

To show that every HELPINSERT and HELPREMOVE operation terminates, it is sufficient to show the list is acyclic. There are three places where the $next$ pointer of a node is changed: executing line 27 cannot form a cycle, since the node h is newly allocated and is not reachable from any other node; when a thread executes line 37 or line 54, $pred$ is clearly always a predecessor of $succ$ in some total order R, which can be defined as the order in which nodes are inserted to the list (by the CAS at line 28).

Since the size of the list is bounded by E, the total number of completed ENLIST operations, every HELPINSERT and HELPREMOVE operation finishes in $O(E)$ steps. Note that in HELPREMOVE, a thread never executes the CAS at line 62 twice on the same node: if the CAS failed, the $curr$ node is turned into a final state (*DAT* or *INV*) and will cause the loop to exit or skip the node in the next iteration. Thus, for n threads, either a thread completes its own list operation in $O(n + E)$ local steps, or some other thread completes a list operation during this period of time.

3.5 Linearizability

A complete proof of linearizability is provided in a Appendix A. We define the linearization point for each operation: An $\text{INSERT}(k)$ or $\text{REMOVE}(k)$ operation linearizes at the successful CAS at line 28 in ENLIST. A $\text{CONTAINS}(k)$ linearizes at line 16 if $k \notin \text{AbsSet}(head)$ when p executes this line. In cases where $k \in \text{AbsSet}(head)$ when p executes this line, the $\text{CONTAINS}(k)$ linearizes at line 16 if the operation returns true. If the operation returns false, we show that there exists a concurrent $\text{REMOVE}(k)$ that linearizes after p executes line 16 and before p's $\text{CONTAINS}(k)$ returns. We let p's $\text{CONTAINS}(k)$ linearize *immediately after* the linearization point of this $\text{REMOVE}(k)$. Note that multiple $\text{CONTAINS}(k)$ operations may be required to linearize after the same $\text{REMOVE}(k)$ operation, and any two of these $\text{CONTAINS}(k)$ operations can be ordered arbitrarily.

4 Achieving Wait-freedom

The major challenge of the wait-free list algorithm lies in the implementation of a wait-free ENLIST operation. In this section, we present a wait-free ENLIST implementation adapted from a wait-free enqueue technique introduced by Kogan and Petrank [14]. We also introduce an adaptive wait-free algorithm which allows applications to trade off between average latency and worst-case latency of operations.

4.1 Wait-free ENLIST Implementation

The enqueue technique introduced by Kogan and Petrank [14] provides a wait-free approach to append nodes at the tail of a list, but it is not immediately available as a solution to the ENLIST problem where nodes are appended at the head position. We employ $prev$ fields to solve this problem. The additional code for implementing a wait-free ENLIST is presented in Figure 2.

The basic idea of the wait-free ENLIST algorithm is to let different ENLIST operations help each other to complete. The helping mechanism must ensure that every ENLIST operation reaches the response point in bounded number of steps (wait-freedom). This requires every thread to announce its intention by creating a descriptor entry in a $status$ array before starting an operation. During its operation, the thread must visit each entry in the status array, helping other threads to make progress. To prevent starvation, each operation is assigned a $phase$ number from a strictly increasing counter, and an operation only helps those with smaller phase numbers.

The wait-free ENLIST operation goes through six steps, as depicted in Figure 3:

(a) The thread first announces its operation by creating a descriptor entry in its slot (indexed by its thread id) in the $status$ array (line 70). The descriptor contains the $phase$ number of the operation, a boolean $pending$ field that indicates whether the operation is incomplete, and a pointer to the enlisting node. Once the descriptor is announced, the subsequent steps can be performed by the thread itself or by some helper thread.

(b) The thread finds the node pointed to by $head$, and attempts to change its $prev$ field to the enlisting node h by a CAS instruction (line 85).

```
datatype DESC
  $phase$    : $\mathbb{N}$      // integer phase number
  $pending$  : $\mathbb{B}$      // whether operation is pending
  $node$     : NODE    // pointer to the enqueueing node

global variables
  $head$     : NODE
  $dummy$    : NODE
  $counter$  : $\mathbb{N}$
  $status$   : DESC[$THREADS$]

initially
  $head \leftarrow$ new NODE$\langle -1, REM,$ nil, nil, $-1\rangle$
  $dummy \leftarrow$ new NODE$\langle -, -, -, -, - \rangle$
  $counter \leftarrow 0$
  foreach $d$ in $status$ do
    $d \leftarrow$ new DESC$\langle -1,$ false, nil$\rangle$

68  procedure ENLIST($h$ : NODE)
69    $phase \leftarrow$ F&I(&$counter$)
70    $status$[threadid] $\leftarrow$ new DESC$\langle phase,$ true, $h\rangle$
71    for $tid \leftarrow 0 \ldots (THREADS - 1)$ do
72      HELPENLIST($tid, phase$)
73    HELPFINISH()

74  function ISPENDING($tid : \mathbb{N}$, $phase : \mathbb{N}$) : $\mathbb{B}$
75    $d \leftarrow status[tid]$
76    return $d.pending \wedge (d.phase \leq phase)$

77  procedure HELPENLIST($tid : \mathbb{N}$, $phase : \mathbb{N}$)
78    while ISPENDING($tid, phase$) do
79      $curr \leftarrow head$
80      $pred \leftarrow curr.prev$
81      if $curr = head$ then
82        if $pred =$ nil then
83          if ISPENDING($tid, phase$) then
84            $n \leftarrow status[tid].node$
85            if $CAS(\&curr.prev,$ nil$, n)$ then
86              HELPFINISH()
87              return
88        else
89          HELPFINISH()

90  procedure HELPFINISH()
91    $curr \leftarrow head$
92    $pred \leftarrow curr.prev$
93    if $(pred \neq$ nil$) \wedge (pred \neq dummy)$ then
94      $tid \leftarrow pred.tid$
95      $d \leftarrow status[tid]$
96      if $(curr = head) \wedge (pred = d.node)$ then
97        $d' \leftarrow$ new DESC$\langle d.phase,$ false, $d.node\rangle$
98        CAS(&$status[tid], d, d'$)
99        $pred.next \leftarrow curr$
100       CAS(&$head, curr, pred$)
101       $curr.prev \leftarrow dummy$
```

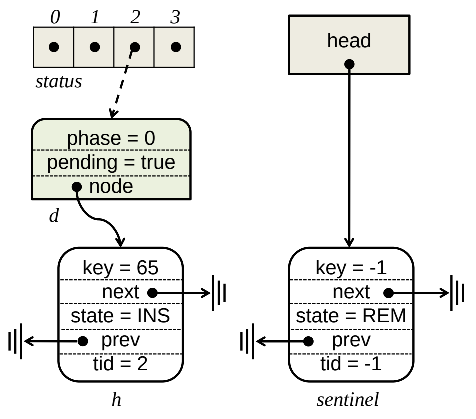
Figure 2: A Wait-free ENLIST Implementation

(c) The thread sets the *pending* flag of the operation descriptor to false by installing a new descriptor (line 98); this prevents concurrent helpers from retrying after the node is enlisted.

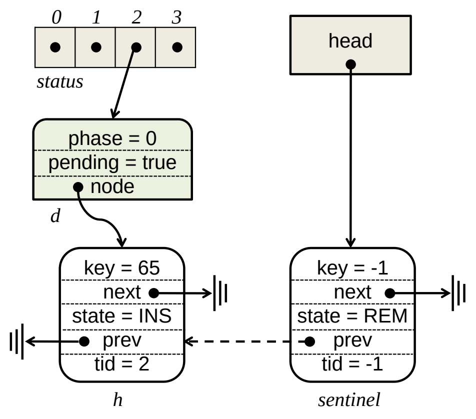
(d) The thread sets $h.next$ to point to the original head node (line 99), which is the linearization point of the ENLIST operation. The ordering of this step is important with respect to steps (b) and (e). That is, the update of $h.next$ must be ordered after $head.prev$ is set to h, since the correct successor of h is "unknown" until then. On the other hand, $h.next$ must be updated before $head$ is changed to h, since otherwise a concurrent CONTAINS operation may start traversing from h and erroneously end by discovering $h.next$ is **nil**.

(e) The thread fixes $head$ by changing it to h using a CAS (line 100).

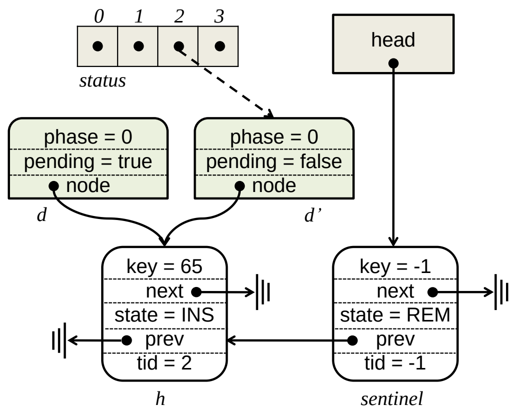
(f) Finally, the thread clears the $prev$ field of the original head by setting it to a $dummy$ state (line 101). This is necessary for allowing the garbage collector to recycle deleted nodes. Since the $prev$ pointers are installed by the wait-free ENLIST implementation, and the lock-free algorithm is unaware of their existence, keeping the $prev$ pointers prevents the garbage collector from reclaiming a node even if the



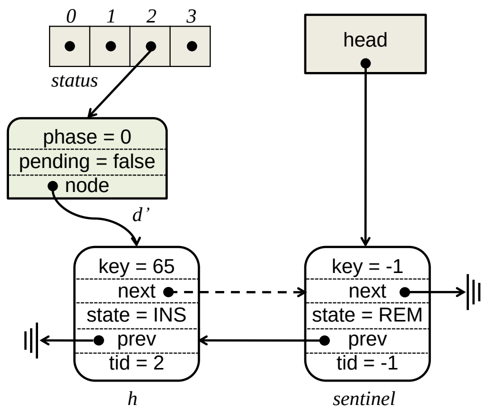
(a) Announce operation



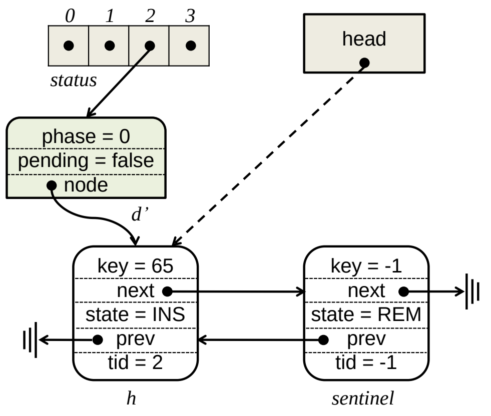
(b) Update prev



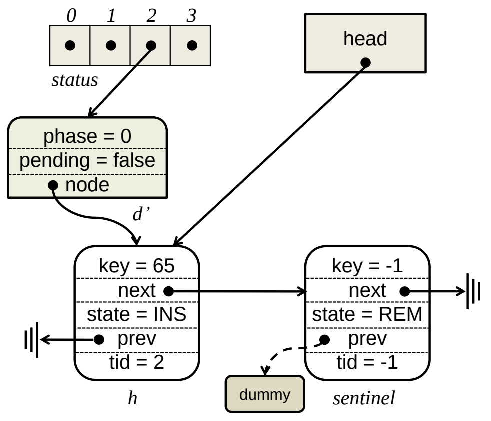
(c) Update descriptor



(d) Update next



(e) Update head



(f) Nullify prev

Figure 3: Wait-free ENLIST Implementation Extended from the Kogan-Petrank Algorithm

node is considered "unreachable" by the lock-free algorithm. It is worth noting that we must invalidate the $prev$ pointer by setting it to a $dummy$ state instead of **nil**, since the latter would admit ABA problems for the CAS instruction (line 85). Once the $prev$ field of a node is set to $dummy$, it never changes.

4.2 An Adaptive Algorithm

Although the wait-free algorithm provides an upper bound on the steps required to complete an operation in the worst case, it imposes overhead in the common cases when contention is low. We employed the "fast-path-slow-path" methodology [15] to construct an adaptive algorithm that performs competitively in the common case while retaining the wait-free guarantee.

In the adaptive algorithm, a thread starts by executing a fast path version of the ENLIST operation, and falls back to the wait-free slow path if the fast path fails too many times (bounded by constant F). To prevent a thread from repeatedly taking the fast path while another thread starves, every thread checks the global status array after completing D operations, and performs helping if necessary. As shown in [15], for n threads, the adaptive algorithm ensures that every ENLIST operation completes in $O(F+D\cdot n^2)$ local steps. The F and D parameters can be adjusted to balance between the worst-case and common-case latency of operations.

It is worth noting that the fast path ENLIST of the adaptive algorithm is *not* equivalent to the lock-free ENLIST implementation in Figure 1. Instead, the fast path algorithm resembles the wait-free protocol, but excluding the announcing and helping steps.

5 Performance Evaluation

We evaluate performance of the lock-free and wait-free list algorithms via a set of microbenchmarks. These experiments allow us to vary the ratio of INSERT, REMOVE and CONTAINS operations, the range of key values, and the initial size of the list. We compare the following list-based set algorithms:

HarrisAMR: Implementation of the Harris-Michael algorithm [16] which also incorporates the wait-free CONTAINS technique introduced in [9]. The implementation uses Java `AtomicMarkableReference` objects to atomically mark deleted nodes.

HarrisRTTI: Optimized implementation of HarrisAMR in which Java run-time type information (RTTI) is used in place of `AtomicMarkableReference`. This is the best-known lock-free list implementation.

LazyList: Lock-based optimistic list implementation proposed by Heller et al [9].

LFList: The lock-free unordered list algorithm discussed in Section 3.

WFList: The basic wait-free unordered list algorithm discussed in Section 4.

Adaptive: The adaptive wait-free unordered list algorithm discussed in Section 4.2.

FastPath: The fast-path portion of the Adaptive algorithm from Section 4.2.

In all implementations (except "HarrisAMR"), we use Java "FieldUpdaters" to perform CAS instructions on object fields. This approach provides better performance than simply using atomic fields (i.e. `AtomicInteger` and `AtomicReference`), which require expensive heap allocation cost and extra indirection overhead.

Experiments were conducted on an HP z600 machine with 6GB RAM and a 2.66GHz Intel Xeon X5650 processor with 6 cores (12 total threads) running Linux kernel 2.6.37 and OpenJDK 1.6.0. Each data point is the median of five 5-second trials. Variance was always below 5%.

	Harris	LazyList	LFList	WFList	Adaptive
INSERT Cost	1 CAS	2 CAS	2 CAS	4 CAS + 1 F&I	3 CAS
REMOVE Cost	2 CAS	2 CAS	1 CAS	3 CAS + 1 F&I	2 CAS
Traverse Distance	$\frac{1}{2}k$		$(1-\frac{\alpha}{2})k$		

Figure 4: Update Cost and Average Traversal Distance (in uncontended cases)

5.1 Expected Overheads

Figure 4 enumerates the expected overheads of each of the algorithms. The cost of a successful list operation is affected by the update cost and the traversal cost. We measure the cost of an update operation (INSERT or REMOVE) by the number of atomic instructions required in the uncontended case. Compared to the Harris algorithm, LFList uses an extra CAS instruction in INSERT and one less in the REMOVE operation. The WFList requires 2 more CAS instructions to provide wait-freedom, though this cost is reduced in the Adaptive algorithm by leveraging the lock-free fast path.

The traversal cost is the average number of nodes that must be accessed. Suppose the list contains k elements uniformly selected from range $[0...M)$ and let $k = \alpha M$ $(0 \leq \alpha \leq 1)$. The average traversal distance for searching a random key value in an ordered list is: $D_o = \frac{1}{2}k$. In unordered lists, the average traversal distance is averaged among successful and unsuccessful search operations: $D_u = \alpha \cdot \frac{1}{2}k + (1-\alpha)k = (1-\frac{\alpha}{2})k$. This suggests that ordered lists have an increasing advantage over unordered lists when the set is sparse. For instance, when $\alpha = \frac{1}{2}$ (half of key space is in the set), the average traversal distance in an unordered list is 50% longer than its ordered permutation. Note too that in the ordered lists, an unsuccessful insert/remove does not perform a CAS, whereas every insert/remove in the unordered list performs a CAS.

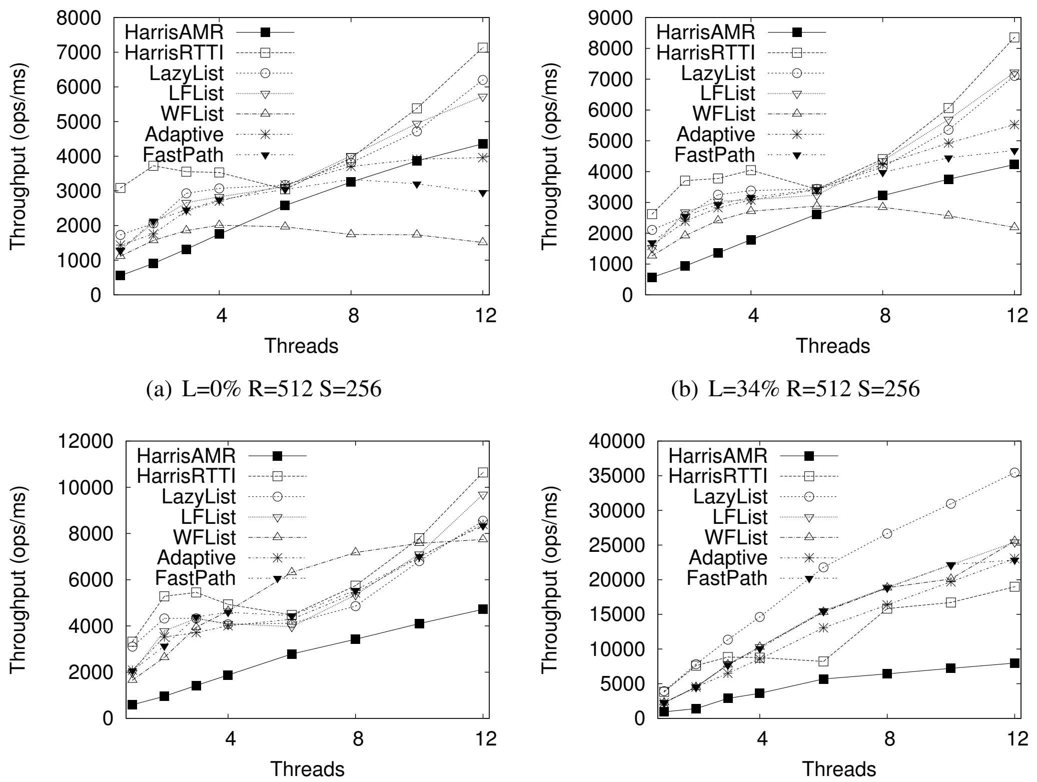
5.2 x86 Performance

In Figures 5–7, we assess the performance of the lists for a variety of workloads. The "L" parameter indicates the percentage of operations that are lookups, with the remainder evenly split between inserts and removals. "R" indicates the key range, and "S" indicates the average size of the list. In every case, the list is pre-populated with a random selection of S unique elements in the range $[0, \mathrm{R})$. These elements are chosen at random, without replacement. Thus in the unordered lists, they will not be ordered.

The x86 processor features an aggressive pipeline, a deep cache hierarchy, and low-latency CAS operations. On this platform, the cost of write-write sharing is high, and thus both the wait-free enlistment mechanism and conflicting CAS operations on the head of the list are potential scalability bottlenecks. Nonetheless, our lock-free and wait-free algorithms scale well in all but a few cases. Indeed, the difference in performance appears to be much more a consequence of the increased traversal distance in the unordered algorithm than a consequence of increased cache misses due to frequent updates to the head of the list.

The most immediate and consistent finding is that the Harris list without RTTI optimizations has substantially higher latency and worse scalability than all other algorithms. We include this result as a reminder that concurrent data structures must be implemented using state-of-the-art techniques. Merely showing improved performance relative to the canonical Harris list presented in [12] does not give any indication of real-world performance. In particular, we caution that a direct comparison between our list and the wait-free ordered list [23] is not possible until that list is redesigned to use these modern optimizations.

We also see that long-running and read-only operations significantly reduce the cost of wait-free enlist-



(a) L=0% R=512 S=256

(b) L=34% R=512 S=256

(c) L=80% R=512 S=256

(d) L=100% R=512 S=256

Figure 5: Microbenchmark - Short Lists (L: Lookup Ratio, R: Key Range, S: List Size)

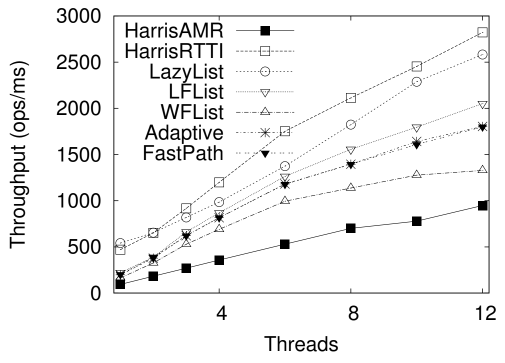
ment. When lists are small and updates are frequent, the enlistment table and counter themselves become a bottleneck. Otherwise, the adaptive algorithm and its FastPath component are nearly identical.

The FastPath lock-free list is always a constant factor slower than the lock-free unordered list, but the Adaptive algorithm remains close to FastPath. This finding confirms Kogan and Petrank's claim [15] that the fast-path-slow-path technique can provide worst-case wait-freedom with lock-free performance. Furthermore, since the average operation in our list accesses many locations, contention on the head node of the list, while significant, does not dominate. Thus we observed that even for small thresholds, the adaptive algorithm rarely fell back to wait-free mode. However, it is important to observe that the lock-free FastPath algorithm itself is slower than our best lock-free unordered list. We shall return to this point in Section 6.

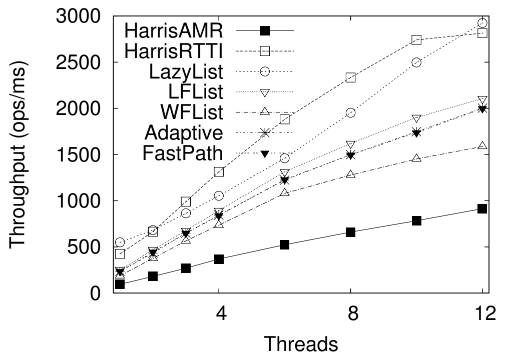
6 Discussion and Future Work

In their paper introducing the fast-path-slow-path methodology, Kogan and Petrank state that "...each operation is built from a fast path and a slow path, where the former is a version of a lock-free implementation of that operation, and the latter is a version of a wait-free implementation. Both implementations are customized to cooperate with each other [15, Sec. 3]."

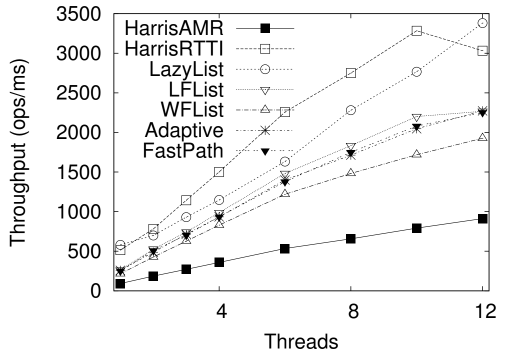
Given a lock-free algorithm *L*, the question then is how to apply the methodology to produce a wait-free algorithm that does not sacrifice performance. We will consider *L* as consisting of three phases: a prefix



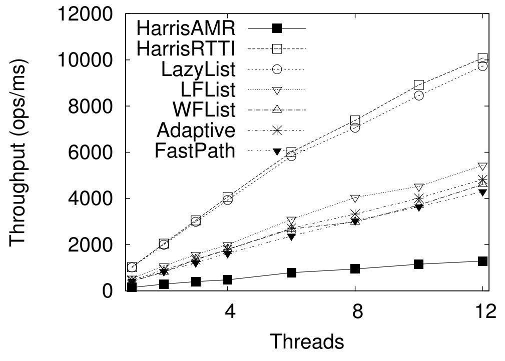
(a) L=0% R=2K S=1K



(b) L=34% R=2K S=1K

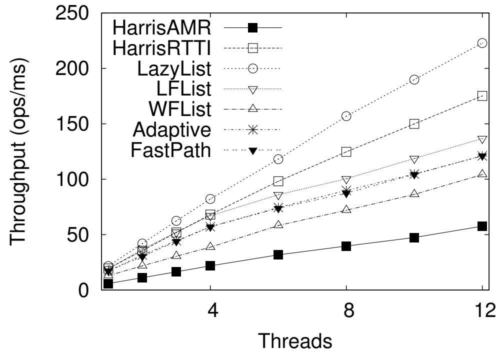


(c) L=80% R=2K S=1K

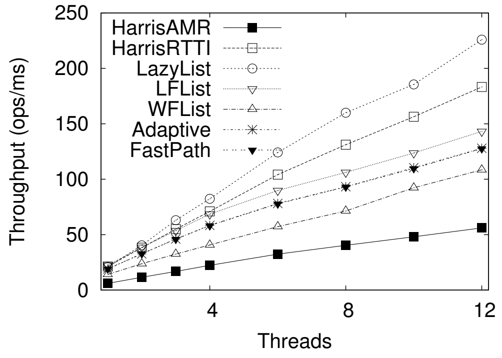


(d) L=100% R=2K S=1K

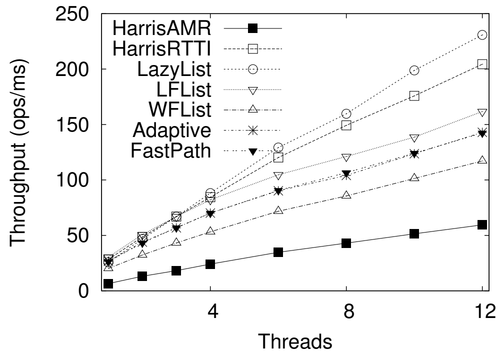
Figure 6: Microbenchmark - Medium Lists (L: Lookup Ratio, R: Key Range, S: List Size)



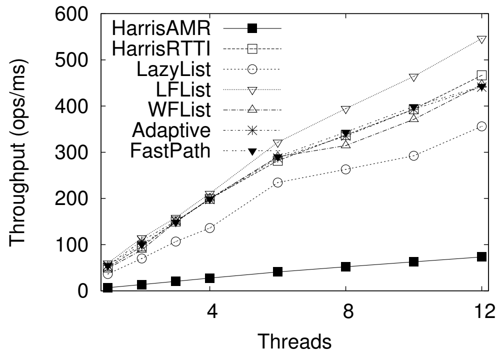
(a) L=0% R=16K S=8K



(b) L=34% R=16K S=8K



(c) L=80% R=16K S=8K



(d) L=100% R=16K S=8K

Figure 7: Microbenchmark - Long Lists (L: Lookup Ratio, R: Key Range, S: List Size)

(instructions that occur before the linearization point), a CAS operation (the linearization point), and a suffix (clean-up operations that follow the linearization point). Considering the three existing fast-path-slow-path algorithms (this work, ordered lists [23], and queues [15]), we see a pattern emerge.

First, a correct wait-free algorithm *W* must be constructed. This entails adding an announcement operation and operation descriptors to *L*. However, this step introduces the possibility of helping in the prefix, and thus makes it possible for helping operations to race (particularly if there are stores to memory that would not be shared in *L*). To correct these races, extra fields must be added to nodes of the data structure, stores must be upgraded to CAS instructions, and these CAS instructions must be sequenced by performing intermediate updates (via CAS) to a descriptor after each prefix step. It appears that changes to the suffix of the operation are not required, since the suffix is either clean-up operations that already support helping (e.g., the second CAS in the M&S queue [18]), or else operations that do not affect data structure invariants (e.g., the list traversal in HELPINSERT).

The second step is to perform a reduction that yields a lock-free algorithm *L'* that remains compatible with *W*. The first step of the reduction is to elide the announce operation and descriptor updates in *L'*. Then *W* must be analyzed, step-by-step, and simplified in an ad-hoc manner. In the ideal case, the result is the original lock-free algorithm *L*. Currently, it appears that the ideal case only occurs when the prefix is empty and the linearization point is the first CAS. Otherwise (as is the case in our list and the ordered list [23]), *L'* will need additional CAS instructions (relative to *L*) to keep its prefix compatible with the prefix of *W*.

Nonetheless, the ability to create low-latency wait-free data structures is valuable, particularly data structures as fundamental as linked lists. To emphasize the significance of our wait-free unordered list, note that our list can be extended to support a REMOVEHEAD operation. Such an operation would resemble our REMOVE operation, but using a wildcard as its key value, and would immediately yield a wait-free stack. In contrast to stacks, constructing wait-free resizable hash tables based on our lists will be nontrivial. One challenge is that the shared descriptor array may become a bottleneck; were it not for resizing, each bucket could have its own descriptor array. However, the unordered nature may simplify other aspects of the design, for example, easing the implementation of list merging/splitting since the resulting lists need not be sorted.

Acknowledgements

We would like to thank Tim Harris, Alex Kogan, Victor Luchangco and our anonymous reviewers for their helpful suggestions during the preparation of our final manuscript.

References

[1] H. Attiya and E. Hillel. Built-In Coloring for Highly-Concurrent Doubly-Linked Lists. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[2] A. Braginsky and E. Petrank. Locality-Conscious Lock-Free Linked Lists. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, Bangalore, India, Jan. 2011.

[3] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification*, Seattle, WA, Aug. 2006.

[4] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, July 2012.

[5] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[6] P. Fatourou and N. D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[7] M. Fomitchev and E. Ruppert. Lock-Free Linked Lists and Skip Lists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[8] T. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

[9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, Pisa, Italy, Dec. 2006.

[10] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[11] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[14] A. Kogan and E. Petrank. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[15] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[16] M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, Aug. 2002.

[17] M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[18] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[19] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying Linearizability with Hindsight. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[20] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[21] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing*, 65:609–627, May 2005.

[22] H. Sundell and P. Tsigas. Lock-Free Deques and Doubly Linked Lists. *Journal of Parallel and Distributed Computing*, 68(7), July 2008.

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-Free Linked-Lists. In *Proceedings of the 16th International Conference on Principles of Distributed Systems*, Rome, Italy, Dec. 2012.

[24] J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, Ottowa, Ontario, Canada, Aug. 1995.

A Proof of Linearizability

In this section, we prove the linearizability of the LFList algorithm. We start by proving basic invariants about the data structure. We then show the algorithm implements an integer set object by mapping the program states to an abstract integer set, and show that every INSERT, REMOVE and CONTAINS operation happens at its linearization point.

In the proof, we add an auxiliary integer field seq to each node object. The auxiliary fields serve to model the order in which nodes enter the list, where the order is used in the acyclicity proof. We also add an auxiliary local variable h to the CONTAINS operation, which records the value of $head$ returned by the first read. The augmented code is listed in Figure 8, and the auxiliary lines are marked by "*".

We say a thread $p@n$ if p is about to executed the line numbered by n.

We use the notation $p@\{l_1, l_2, ..., l_n\}$ as the abbreviation for $p@l_1 \vee p@l_2 ... \vee p@l_n$, and notation $p@\{m-n\}$ $(m < n)$ as the abbreviation for $p@m \vee p@(m+1) ... \vee p@n$.

We say a thread $p@\text{ENLIST}$, if $p@\{25-28\}$.

We say a thread $p@\text{HELPINSERT}$, if $p@\{31-46\}$.

We say a thread $p@\text{HELPREMOVE}$, if $p@\{48-67\}$.

We say a thread $p@\text{CONTAINS}$, if $p@\{16-23\}$.

We say a thread $p@\text{INSERT}$, if $p@\{2-8\}$, or $p@\{\text{HELPINSERT}, \text{HELPREMOVE}, \text{ENLIST}\}$ and p's the current operation is invoked by an INSERT operation.

We say a thread $p@\text{REMOVE}$, if $p@\{10-14\}$, or $p@\{\text{HELPREMOVE}, \text{ENLIST}\}$ and p's current operation is invoked by a REMOVE operation.

A.1 Basic Properties

The following observations assert some basic properties of the algorithm (i.e. invariants about local variables and state changing of nodes), and can be verified easily from the code.

The following observation ensures that every newly allocated node is unique.

```
**datatype** NODE
    $key$    : $\mathbb{N}$       // integer data field
    $state$  : $\mathbb{N}$       // $INS$, $REM$, $DAT$, or $INV$
    $next$   : NODE    // pointer to the successor
    $prev$   : NODE    // pointer to the predecessor
    $tid$    : $\mathbb{N}$       // thread id of the creater
    $seq$    : $\mathbb{N}$       // auxiliary: sequence number

**global variables**
    $head$   : NODE    // initially **nil**

**function** INSERT($k : \mathbb{N}$) : $\mathbb{B}$
    $h \leftarrow$ **new** NODE$\langle k, INS,$ **nil**, **nil**, **threadid**$\rangle$
    ENLIST($h$)
    $b \leftarrow$ HELPINSERT($h, k$)
    **if** $\neg CAS(\&h.state, INS, (b?\ DAT : INV))$ **then**
        HELPREMOVE($h, k$)
        $h.state \leftarrow INV$
    **return** $b$

**function** REMOVE($k : \mathbb{N}$) : $\mathbb{B}$
    $h \leftarrow$ **new** NODE$\langle k, REM,$ **nil**, **nil**, **threadid**$\rangle$
    ENLIST($h$)
    $b \leftarrow$ HELPREMOVE($h, k$)
    $h.state \leftarrow INV$
    **return** $b$

**function** CONTAINS($k : \mathbb{N}$) : $\mathbb{B}$
    $curr \leftarrow head$
*   $h \leftarrow curr$
    **while** $curr \neq$ **nil** **do**
        **if** $curr.key = k$ **then**
            $s \leftarrow curr.state$
            **if** $s \neq INV$ **then**
                **return** $(s = INS) \vee (s = DAT)$
        $curr \leftarrow curr.next$
    **return false**

**procedure** ENLIST($h$ : NODE)
    **while true do**
        $old \leftarrow head$
        $h.next \leftarrow old$
*       $h.seq \leftarrow (old =$ **nil**$)\ ?\ 0\ :\ (old.seq + 1)$
        **if** $CAS(\&head, old, h)$ **then**
            **return**

**function** HELPINSERT($h$ : NODE, $k : \mathbb{N}$) : $\mathbb{B}$
    $pred \leftarrow h$
    $curr \leftarrow pred.next$
    **while** $curr \neq$ **nil** **do**
        $s \leftarrow curr.state$
        **if** $s = INV$ **then**
            $succ \leftarrow curr.next$
            $pred.next \leftarrow succ$
            $curr \leftarrow succ$
        **else if** $curr.key \neq k$ **then**
            $pred \leftarrow curr$
            $curr \leftarrow curr.next$
        **else if** $s = REM$ **then**
            **return true**
        **else if** $(s = INS) \vee (s = DAT)$ **then**
            **return false**
    **return true**

**function** HELPREMOVE($h$ : NODE, $k : \mathbb{N}$) : $\mathbb{B}$
    $pred \leftarrow h$
    $curr \leftarrow pred.next$
    **while** $curr \neq$ **nil** **do**
        $s \leftarrow curr.state$
        **if** $s = INV$ **then**
            $succ \leftarrow curr.next$
            $pred.next \leftarrow succ$
            $curr \leftarrow succ$
        **else if** $curr.key \neq k$ **then**
            $pred \leftarrow curr$
            $curr \leftarrow curr.next$
        **else if** $s = REM$ **then**
            **return false**
        **else if** $s = INS$ **then**
            **if** $CAS(\&curr.state, INS, REM)$ **then**
                **return true**
        **else if** $s = DAT$ **then**
            $curr.state \leftarrow INV$
            **return true**
    **return false**
```

Figure 8: LFList with Auxiliary Variables

Observation 1. For any different threads p and q, if $p, q@\{\text{INSERT}, \text{REMOVE}\}$, then $h_p \neq h_q$.

The following observation captures the property that a newly allocated node is not reachable from $head$, or local variables of any other thread, before the node is inserted at the head position.

Observation 2. If p@ENLIST while q is another thread, then $h_p \neq head$ or any local variable of q, including h_q, old_q, $pred_q$, $curr_q$, or $succ_q$.

Observation 3. The key of a node is never changed.

Observation 4. The $state$ of a node A is never changed if $A.state =$ *INV*.

Observation 5. The $state$ of a node cannot be changed to *INS* unless at initialization.

Observation 6. The $state$ of a node is changed to *REM* only by a CAS at line 62 unless at initialization, and the $state$ of the node must be *INS* in the prestate.

Observation 7. The $state$ of a node is changed to *DAT* only by a CAS at line 5.

Lemma 8. For any node A, if $A.state =$ *REM* in the prestate, then $A.state$ can be changed by some thread p only via lines 7 or 13, and $h_p = A$.

Proof. Notice that $A.state$ cannot be changed by a step at line 65, since it requires that $A.state$ was changed from *DAT* to *REM* (the prestate), which is impossible. □

Lemma 9. For any node A, if $A.state =$ *INS* in the prestate, then $A.state$ can be changed by some thread p only via the CAS at line 5 and $h_p = A$, or via the CAS at line 62 and $curr_p = A$.

Proof. Notice that $A.state$ cannot be changed by a step at lines 7, 13 or 65, since these steps require that $A.state$ was changed from some other state to *INS* (the prestate), which is impossible. □

Invariant 10. If $p@\{5, \text{HELPINSERT}\}$, then $h_p.state =$ *INS* or $h_p.state =$ *REM*.

Proof. Initially, $h_p.state =$ *INS*. By lemma 9, $h_p.state$ can be changed only to *REM*, and cannot be changed further by any other threads. □

Invariant 11. If $p@\{7, 13, \text{HELPREMOVE}\}$, then $h_p.state =$ *REM*.

Proof. If p@13, or p@HELPREMOVE invoked from a REMOVE operation, then h_p is initially a *REM* node. By lemma 8, $h_p.state$ cannot be changed by any thread other than p.

If p@7, or p@HELPREMOVE invoked from a INSERT operation, then h_p is initially a *INS* node and p's CAS at line 5 failed. By lemma 9, $h_p.state =$ *REM*, and by lemma 8, $h_p.state$ cannot be changed by any thread other than p. □

A.2 Acyclicity

For any two nodes A and B, we define predicate Reach(A, B) as follows:

$$\text{Reach}(A, B) \equiv \langle A, B\rangle \in \text{TRANSITIVECLOSURE}(\{\langle X, Y\rangle \mid X \neq \textbf{nil} \land X.next = Y\})$$

We prove the following invariants together by induction over reachable states:

Invariant 12. If $p@\{36, 53\}$, then $pred_p.seq > curr_p.seq$.

Invariant 13. If $p@\{37, 54\}$, then $curr_p.seq > succ_p.seq$.

Invariant 14. For any non-nil nodes A and B, $A.next = B \Rightarrow A.seq > B.seq$.

Proof. Assume the invariants hold in the prestate of any reachable step α, we show the invariants hold in the poststate. Initially, invariants 12 and 13 hold vacuously, and invariant 14 holds since the $next$ field of every node is set to **nil**. It is easy to verify that invariants 12 and 13 hold in the poststate, since observation 2 ensures that the seq fields of $pred_p$, $curr_p$ and $succ_p$ cannot be changed by a step at line 27 of any other thread q.

To show that invariant 14 holds in the poststate, we consider the steps that change the $next$ fields: If α is a step at line 27 and $old_p \neq$ **nil**, then α sets $h_p.next = old_p \wedge h_p.seq = old_p.seq + 1$, and the invariant holds in the poststate. If α is a step at line 37 or line 54, then α sets $pred_p.next$ to $succ_p$. Since $curr_p.seq > succ_p.seq \wedge pred_p.seq > curr_p.seq$ holds in the prestate, we have $pred_p.next = succ_p \wedge pred_p.seq > succ_p.seq$ in the poststate. □

For any node A, we define set List(A) as follows:

$$\text{List}(A) \equiv \{A\} \cup \{X \mid \text{Reach}(A, X)\}$$

Corollary 15. For every node A, there is a bijection from $\{1..n\}$ to List(A), where n is the number of nodes in the list such that for $i \in [1, n-1]$, the next field of the i-th node points to the $(i+1)$-st node, and the next field of the n-th node is **nil**. (Note that the bijection is unique. When there are no nodes in the list, and the domain of the bijection is the empty set.)

Proof. The above invariants show that the $next$ field of a node always points to another node with smaller seq field, or points to **nil**. □

A.3 Reachability

The reachability property ensures that any non-*INV* node is always reachable from the $head$ pointer. To prove this property, we show that the physical deletion steps (lines 35 - 38 and lines 52 - 55) cannot make any non-*INV* unreachable.

We start by proving the following two invariants together by induction:

Invariant 16. If $p@\{36, 53\}$ and $X.state \neq \mathit{INV}$, Reach$(curr_p, X) \Rightarrow$ Reach$(pred_p, X)$.

Invariant 17. If $p@\{37, 54\}$ and $X.state \neq \mathit{INV}$, Reach$(succ_p, X) \Rightarrow$ Reach$(pred_p, X)$.

Proof. Assume the invariants hold in the prestate of any reachable step α, we show the invariants hold in the poststate. Initially, both invariants hold vacuously. It is also easy to verify that the invariants hold in the poststate if α is a step by thread p itself, or a step at line 27.

We now consider the cases where α is a step made by another thread q that changes the $next$ fields. That is, α is a step at line 37 or 54. We show that any non-*INV* node X that is reachable from $pred_p$ remains reachable from $pred_p$ in the poststate. Notice that if $pred_q \notin$ List$(pred_p)$, α cannot change the reachability between nodes in List$(pred_p)$. Now we assume $pred_q \in$ List$(pred_p)$. Since invariant 17 holds in the prestate, any non-*INV* node X that is reachable from $pred_q$ (also from $pred_p$) remains reachable from $pred_q$ (also from $pred_p$) in the poststate. On the other hand, for any non-*INV* node A that is reachable from $pred_p$ but not from $pred_q$, Reach$(pred_p, X)$ remains to hold in the poststate. □

Corollary 18 (Safety of Deletion)**.** For any node A and B, if $B.state \neq \mathit{INV} \wedge$ Reach(A, B) in the prestate of any reachable step, then Reach(A, B) holds in the poststate.

Corollary 19 (Reachability)**.** For any node A that $A.state \neq \mathit{INV}$, $A \in \text{List}(head)$ unless $A = h_p$ and $p@\text{ENLIST}$.

The safety of deletion property ensures that a node cannot be physically deleted (made unreachable from any other node) before its $state$ field is changed to *INV*, and the reachability property ensures that every non-*INV* node is reachable from $head$ unless the node is not yet enlisted by the CAS at line 28.

A.4 INSERT and REMOVE Invariants

For any node A and integer k, we define set RelatedSuccSet(A, k) as follows:

$$\text{RelatedSuccSet}(A, k) \equiv \{X \mid \text{Reach}(A, X) \wedge X.key = k \wedge X.state \neq \mathit{INV}\} \cup \{\textbf{nil}\}$$

Lemma 20. For any node A and integer k, exists unique node X (which we denoted as RelatedSucc(A, k)) such that

$$X \in \text{RelatedSuccSet}(A, k) \wedge (\forall Y \in \text{RelatedSuccSet}(A, k).\ \text{Reach}(X, Y) \vee X = Y)$$

Proof. We first notice that RelatedSuccSet(A, k) is not empty as **nil** is in the set. Since for every node Y in RelatedSuccSet(A, k), $Y.state \neq \mathit{INV}$, and by corollary 19, Y is in List(A). Therefore, all nodes in RelatedSuccSet(A, k) are totally ordered under reachability relation, and there exists a minimal element X such that Reach(X, Y) for every other Y in the set. $\square$

The following lemmas characterize the relationship between the RelatedSucc(A, k) node(s) in prestate to poststate for any node A and integer k.

Lemma 21. For any node A and integer k, let $X = \text{RelatedSucc}(A, k)$ in the prestate of any reachable step α that sets $X.state = \mathit{INV}$, then in the poststate RelatedSucc$(A, k) = \text{RelatedSucc}(X, k)$.

Proof. Immediate from the definition in lemma 20. $\square$

Lemma 22. For any node A and integer k, let $X = \text{RelatedSucc}(A, k)$ in the prestate of any reachable step α which is *not* a step that sets $X.state = \mathit{INV}$, then in the poststate $X = \text{RelatedSucc}(A, k)$.

Proof. If α is a step that changes the $next$ field, then by corollary 18, only *INV* nodes can be made unreachable. Thus, $X = \text{RelatedSucc}(A, k)$ in the poststate.

If α is a step that changes the $state$ field of some node M:

- If $X = M$ and α sets $X.state$ to some non-*INV* state, then $X = \text{RelatedSucc}(A, k)$ in the poststate.
- If Reach$(A, M) \wedge$ Reach(M, X) in the prestate: If $M.key \neq k$, then $X = \text{RelatedSucc}(A, k)$ holds in the prestate and poststate since the key fields are immutable. Otherwise, $M.key = k$, and we have $M.state = \mathit{INV}$ in the prestate (otherwise X does not satisfy the property defined in lemma 20). By observation 4, $M.state = \mathit{INV}$ holds in the poststate, and thus, $X = \text{RelatedSucc}(A, k)$.
- Otherwise, M is not between A and X ($A \in \text{List}(M)$ or Reach(X, M)). Thus, changing $M.state$ cannot change RelatedSucc(A, k). $\square$

We use the abbreviation RS$_p$ to denote RelatedSucc(h_p, k_p), if p is at a HELPINSERT, HELPREMOVE, or CONTAINS operation.

$$\text{RS}_p \equiv \text{RelatedSucc}(h_p, k_p)$$

Corollary 23. For any thread p, RS_p refers to the same node in the prestate and poststate of any reachable step α, unless α is a step that sets the $state$ field of RS_p (in the prestate) to *INV*.

The following lemma shows that the RS nodes are different for different threads p and q, if p and q are at HELPINSERT or HELPREMOVE operations.

Lemma 24. For any different threads p and q, if $p, q@\{\text{HELPINSERT}, \text{HELPREMOVE}\}$, then $\text{RS}_p \neq \text{RS}_q$ unless $\text{RS}_p = \text{RS}_q = \textbf{nil}$.

Proof. By contradiction assume $\text{RS}_p = \text{RS}_q \neq \textbf{nil}$. By definition, $k_p = k_q$. By invariants 10 and 11, h_p and h_q are non-*INV* nodes, and by corollary 19, both are in $\text{List}(head)$. Without loss of generality, assume $\text{Reach}(h_p, h_q)$. On the other hand, from the definition of RS_q we have $\text{Reach}(h_q, \text{RS}_p)$, which implies that RS_p does not satisfy the required property defined in lemma 20. □

The following invariant ensures that a while loop in the HELPINSERT or HELPREMOVE operation by thread p either encounters RS_p and exits, or continues to the next iteration.

Invariant 25. If $p@\{\text{HELPINSERT}, \text{HELPREMOVE}, \text{CONTAINS}\}$, then either $\text{RS}_p = curr_p$ or $\text{RS}_p = \text{RelatedSucc}(curr_p, h.key)$.

Proof. The invariant vacuously holds initially. Assume the invariant holds in the prestate before any reachable step α, we show it holds in the poststate. It is easy to verify that the invariant holds in the poststate if α is a step of p. We now consider the cases where α is a step of some other thread q.

If $\text{RS}_p = curr_p$ holds in the prestate, then by lemmas 21 and 22, we have $\text{RS}_p = curr_p \vee \text{RS}_p = \text{RelatedSucc}(curr_p, h.key)$ holds in the poststate.

If $\text{RS}_p = \text{RelatedSucc}(curr_p, h.key)$ holds and let $X = \text{RS}_p$ in the prestate, then in the poststate, by lemmas 21 and 22, either $\text{RS}_p = \text{RelatedSucc}(curr_p, h.key) = X$, or $\text{RS}_p = \text{RelatedSucc}(curr_p, h.key) = \text{RelatedSucc}(X, h.key)$ holds. □

The following invariant ensures that in a HELPINSERT or HELPREMOVE operation, $curr_p$ is always reachable from h_p if $curr_p$ is not an *INV* node.

Invariant 26. If $p@\{\text{HELPINSERT}, \text{HELPREMOVE}, \text{CONTAINS}\}$, then $curr_p.state \neq \textit{INV} \Rightarrow \text{Reach}(h_p, curr_p)$.

Proof. The invariant vacuously holds initially. Assume the invariant holds in the prestate before any reachable step α, we show it holds in the poststate. It is easy to verify that the invariant holds in the poststate if α is a step of p. If α is a step of some other thread q, since $curr_p.state = \textit{INV}$, by corollary 18, $\text{Reach}(h_p, curr_p)$ holds in the poststate. □

The following lemma ensures that if a HELPINSERT or HELPREMOVE encounters a non-*INV* node with the same key with h_p, the node is RS_p.

Invariant 27. If $p@\{\text{HELPINSERT}, \text{HELPREMOVE}, \text{CONTAINS}\}$, then $(curr_p.state \neq \textit{INV} \wedge curr_p.key = k_p) \Rightarrow \text{RS}_p = curr_p$.

Proof. By contradiction assume $curr_p.state \neq \textit{INV} \wedge curr_p.key = k_p$ and $curr_p \neq \text{RS}_p$, then by invariant 25, $\text{RS}_p = \text{RelatedSucc}(curr_p, h.key)$. Since $curr_p.state \neq \textit{INV}$, by invariant 26, $\text{Reach}(h_p, curr_p)$. This creates a contradiction with the property of RS_p defined in lemma 20. □

A.5 Linearizability

We define two auxiliary predicates $\mathrm{K}(X)$ and $\overline{\mathrm{K}}(X)$ to facilitate subsequent proofs:

$$\mathrm{K}(X) \equiv\ (X \neq \textbf{nil}) \wedge ((X.state = \mathit{DAT}) \vee (X.state = \mathit{INS}))$$

$$\overline{\mathrm{K}}(X) \equiv\ (X = \textbf{nil}) \vee (X.state = \mathit{REM}) \vee (X.state = \mathit{INV})$$

Intuitively, for any node A and X such that $\mathrm{RelatedSucc}(A, X.key) = X$, $\mathrm{K}(X)$ means that $X.key$ is in the set represented by $\mathrm{List}(A.next)$, and $\overline{\mathrm{K}}(X)$ means $X.key$ is not in the set.

Invariant 28. If $p@\{7, 13\}$, or $p@5$ and $b_p = \textbf{true}$, then $\overline{\mathrm{K}}(\mathrm{RS}_p)$.

Proof. The invariant vacuously holds initially. Assume the invariant holds in the prestate before any reachable step α, we show it holds in the poststate.

If $p@\{7, 13\}$, or $p@5$ and $b_p = \textbf{true}$ only in the poststate, then by invariant 27, $\overline{\mathrm{K}}(\mathrm{RS}_p)$ holds in the poststate.

If $p@\{7, 13\}$, or $p@5$ and $b_p = \textbf{true}$, in both prestate and poststate, α is a step of another thread q. Since $\overline{\mathrm{K}}(\mathrm{RS}_p)$ holds in the prestate, $\mathrm{RS}_p = \textbf{nil}$ or $\mathrm{RS}_p.state = \mathit{REM}$. By lemma 8, q cannot change $\mathrm{RS}_p.state$, and by corollary 23, RS_p refers to the same node in the prestate and poststate. Thus, $\overline{\mathrm{K}}(\mathrm{RS}_p)$ holds in the poststate. □

Invariant 29. If $p@5 \wedge b_p = \textbf{false}$, then $\mathrm{K}(\mathrm{RS}_p)$.

Proof. The invariant vacuously holds initially. Assume the invariant holds in the prestate before any reachable step α, we show it holds in the poststate.

If $p@5$ only in the poststate, then by invariant 27, $\neg b_p \Rightarrow \mathrm{K}(\mathrm{RS}_p)$ holds in the poststate.

If $p@5$ in both prestate and poststate, α is a step of another thread q.

- If α is a step at line 5 that changes $\mathrm{RS}_p.state$ to *INV*, then $b_q = \textbf{false}$. Since $\mathrm{RS}_p = h_q$ in the prestate, $\mathrm{RS}_p = \mathrm{RS}_q$ in the poststate. Since invariant 29 holds in the prestate (for thread q), $\mathrm{K}(\mathrm{RS}_p)$ holds in the poststate.
- If α is a step at lines 7 or 13 that changes $h_q.state$, then $\mathrm{RS}_p \neq h_q$ in the prestate, since invariant 28 requires $\mathrm{RS}_p.state = \mathit{REM}$ and $\overline{\mathrm{K}}(\mathrm{RS}_p)$ in the prestate. Thus, α does not change $\mathrm{RS}_p.state$.
- If α is a step at lines 62 or 65 that changes $curr_q.state$, then by invariant 27, $\mathrm{RS}_q = curr_q$. Since p and q are different threads, $\mathrm{RS}_p \neq \mathrm{RS}_q$. Thus, α does not change $\mathrm{RS}_p.state$. □

The following lemmas ensure that the return value of an INSERT or a REMOVE operation will be consistent as the moment when it linearizes.

Lemma 30. If $p@\{\text{HELPINSERT}, \text{HELPREMOVE}\}$ and $\mathrm{K}(\mathrm{RS}_p)$ holds in the prestate of any reachable step, then $\mathrm{K}(\mathrm{RS}_p)$ holds in the poststate.

Proof. It is easy to verify that $\mathrm{K}(\mathrm{RS}_p)$ holds in the poststate if α is a step of thread p. We now consider the cases where α be a step by a different thread q.

- If α is a step at line 5 that changes $\mathrm{RS}_p.state$ to *INV*, then $b_q = \textbf{false}$. Since $\mathrm{RS}_p = h_q$ in the prestate, $\mathrm{RS}_p = \mathrm{RS}_q$ in the poststate. By invariant 29, $\mathrm{K}(\mathrm{RS}_p)$ holds in the poststate.
- If α is a step at lines 7 or 13 that changes $h_q.state$, then $\mathrm{RS}_p \neq h_q$ in the prestate, since invariant 28 requires $\mathrm{RS}_p.state = \mathit{REM}$ and $\overline{\mathrm{K}}(\mathrm{RS}_p)$ in the prestate. Thus, α does not change $\mathrm{RS}_p.state$.

- If α is a step at lines 62 or 65 that changes $curr_q.state$, then by invariant 27, $\text{RS}_q = curr_q$. Since p and q are different threads, $\text{RS}_p \neq \text{RS}_q$. Thus, α does not change $\text{RS}_p.state$. $\square$

Lemma 31. If $p@\{\text{HELPINSERT}, \text{HELPREMOVE}, \text{CONTAINS}\}$ and $\overline{\text{K}}(\text{RS}_p)$ holds in the prestate of any reachable step, then $\overline{\text{K}}(\text{RS}_p)$ holds in the poststate.

Proof. In the prestate, either $\text{RS}_p = \textbf{nil}$ or $\text{RS}_p.state = \mathit{REM}$. It is easy to verify that $\text{K}(\text{RS}_p)$ holds in the poststate if α is a step of thread p. We now consider the cases where α be a step by a different thread q. By lemma 8, q cannot change $\text{RS}_p.state$, and by corollary 23, RS_p refers to the same node in the prestate and poststate. Thus, $\overline{\text{K}}(\text{RS}_p)$ holds in the poststate. $\square$

The subsequent proofs show that the LFList algorithm implements an abstract set object. At any step during the execution, the state of the abstract set AbsSet is defined as follows:

$$\text{AbsSet} \equiv \begin{cases} \emptyset & \text{if } head = \textbf{nil} \\ \{k \mid \text{K}(\text{RelatedSucc}(head, k))\} & \text{if } head.state = \mathit{INV} \\ \{k \mid \text{K}(\text{RelatedSucc}(head, k))\} \cup \{head.key\} & \text{if } head.state = \mathit{INS} \vee head.state = \mathit{DAT} \\ \{k \mid \text{K}(\text{RelatedSucc}(head, k))\} \setminus \{head.key\} & \text{if } head.state = \mathit{REM} \end{cases}$$

We define the linearization point of each operation by thread p as follows:

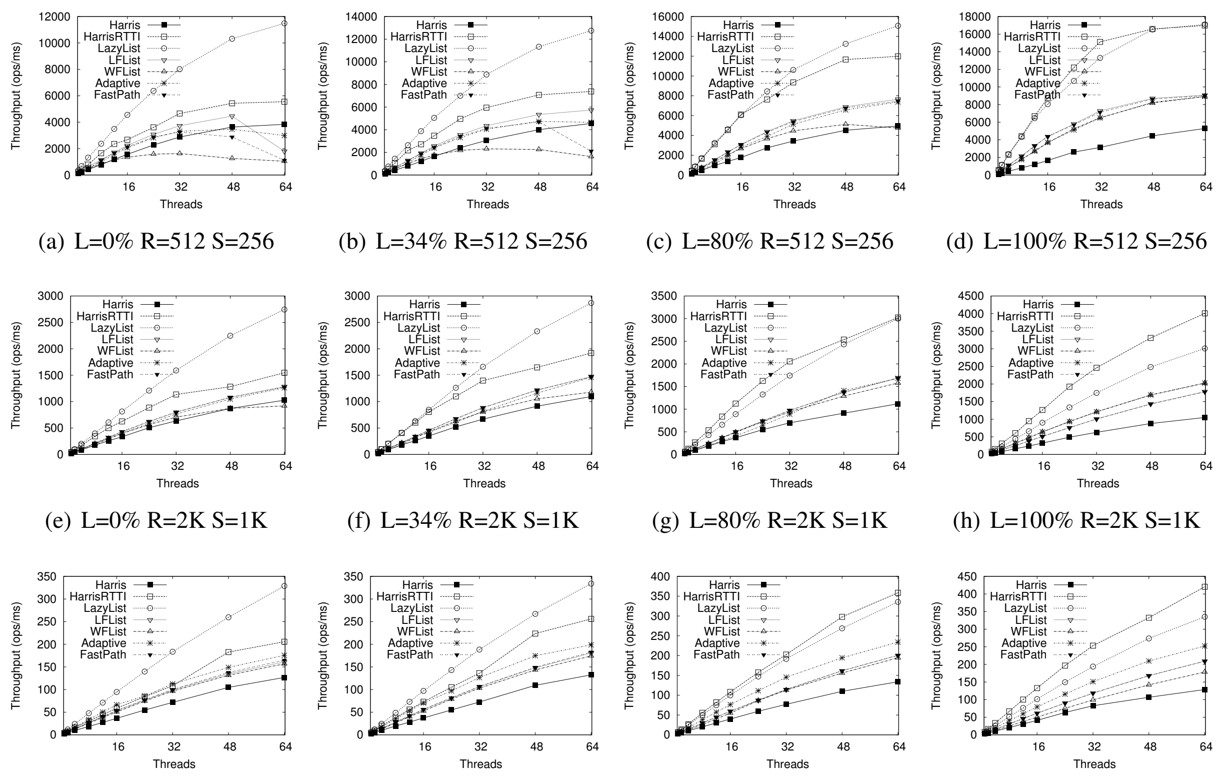
- An $\text{INSERT}(k)$ linearizes at the successful CAS at line 28 in ENLIST.
- A $\text{REMOVE}(k)$ linearizes at the successful CAS at line 28 in ENLIST.
- A $\text{CONTAINS}(k)$ linearizes at line 16, if $k \notin$ AbsSet when p executes this line.
- A $\text{CONTAINS}(k)$ linearizes as follows, if $k \in$ AbsSet when p executes this line:
  - If the operation returns true, then it linearizes at line 16.
  - If the operation returns false, then there exists a concurrent $\text{REMOVE}(k)$ that linearizes after p executes line 16, and before p's $\text{CONTAINS}(k)$ returns, and we let p's $\text{CONTAINS}(k)$ linearize *immediately after* the linearization point of this $\text{REMOVE}(k)$. (Note that multiple $\text{CONTAINS}(k)$ operations may be required to linearize after the same $\text{REMOVE}(k)$ operation, and any two of these $\text{CONTAINS}(k)$ operations can be ordered arbitrarily.)

Theorem 32. The LFList algorithm is linearizable.

Proof. From the definition of AbsSet, it is easy to see that an INSERT operation unions its key with AbsSet at its linearization point, and a REMOVE operation removes its key from AbsSet at its linearization point. The expected return value of each operation is determined by the prestate of the CAS at line 28, and lemmas 30, 31 and invariant 27 ensure the operation will return this value.

For a $\text{CONTAINS}(k)$ operation by thread p and $k \notin$ AbsSet when p executes line 16, lemma 31 and invariant 27 ensure the operation will return false, and the operation can linearize at line 16.

For a $\text{CONTAINS}(k)$ operation by thread p and $k \in$ AbsSet when p executes line 16, $\text{K}(\text{RS}_p)$ holds when $p@16$. Thus, the operation can linearize at line 16 if it returns true. If the operation returns false, then by invariant 27, between its invocation and response, exists a step that makes $\overline{\text{K}}(\text{RS}_p)$ holds from a prestate in which $\text{K}(\text{RS}_p)$ holds. This can only be a step at line 62 from a concurrent $\text{REMOVE}(k)$ by thread q. By invariant 26, h_q is a predecessor of RS_p. And since $\text{K}(\text{RS}_p)$ holds in the prestate, h_q cannot be a successor of h_p, which would otherwise contradicts with the property of RS_p defined in lemma 20. Thus, q's $\text{REMOVE}(k)$ must linearize after p executes line 16, and we can linearize p's $\text{CONTAINS}(k)$ immediately after the linearization point of this $\text{REMOVE}(k)$ operation. $\square$



(a) L=0% R=512 S=256 (b) L=34% R=512 S=256 (c) L=80% R=512 S=256 (d) L=100% R=512 S=256

(e) L=0% R=2K S=1K (f) L=34% R=2K S=1K (g) L=80% R=2K S=1K (h) L=100% R=2K S=1K

(i) L=0% R=16K S=8K (j) L=34% R=16K S=8K (k) L=80% R=16K S=8K (l) L=100% R=16K S=8K

Figure 9: Mixed Workload Performance on SPARC (L: Lookup Ratio, R: Key Range, S: List Size)

B Additional Evaluation: SPARC Results

In general, the SPARC results in Figure 9 confirm our findings on the x86. The main difference is that the cache hierarchy is much more shallow on the SPARC, with the large shared L2 cache accessible to all cores with low latency. While CAS operations are slow, contended CAS operations do not cause a significant slowdown, due to the implementation of CAS at the L2 cache. Furthermore, while there are at most 64 threads, there are only 8 cores, and thus bottlenecks are not expected to become pathological at high thread counts.

Given these architectural differences, the most important observation that follows from evaluation on the SPARC is in Figures 9(a) and (b). Here, we see that the overhead of traversing the enlistment table causes significant slowdown for high thread counts, to the point that the WFList ceases to scale, and even falls below the performance of the unoptimized Harris list. This is not only the case at 64 threads (where any interrupt or garbage collection can significantly impact results), but begins as early as 24 threads.

Another surprise is that the Adaptive algorithm on SPARC typically matches the WFList, rather than the FastPath algorithm. This seems to contradict our findings on x86, and also those of Kogan and Petrank. However, the difference is more a reflection of latency than scalability, suggesting that the just-in-time compiler is optimizing the FastPath and Adaptive algorithms differently.

Lastly, we observe that on the SPARC, LFList always outperforms FastPath. This, again, is an unfortunate finding, since there is no straightforward transformation of LFList that can deliver wait-freedom.

C Wait-free Hash Tables

Either of our wait-free list algorithms is immediately usable as the basic building block of a wait-free fixed-size hash table. Such non-resizable data structures are of use to embedded and real-time operating systems and applications. In a closed-addressing hash table implementation, each bucket implements a set containing elements with the same hash code. We can construct a wait-free closed-addressing hash table by using a wait-free list for each bucket, so that elements can be inserted and removed from a bucket in a bounded number of local steps. Since there is no resizing, a lookup operation is similarly straightforward: after determining which bucket might contain the key, a wait-free lookup can be performed in the appropriate list.

The main challenge for such an implementation is that concurrent operations involving objects whose keys map to different buckets should not experience memory contention. However, in a naive implementation the act of announcing operations in a single array can create a bottleneck, as unrelated operations are forced to both (a) contend over shared variables related to announcing their operations, and (b) help unrelated operations to complete. There are three straightforward techniques for reducing this cost. The first strategy is to use the adaptive wait-free algorithm, with a relatively high bound on the number of attempts before falling back to wait freedom. For workloads with few natural hash collisions, such a strategy may be sufficient. Second, when a thread performing an insertion or removal finds in the announcement array an operation that requires helping, it could ignore the operation if it can prove that its operation and the announced operation hash to different buckets. A third approach is to employ a distinct announcement array for each bucket in the hash table. This has a higher space overhead, but provides true disjoint access parallelism.

D A Wait-free Stack Implementation

We present a wait-free list-based stack implementation in Figure 10. The stack object implements a list with two operations, PUSH and POP, which insert or remove an element from the $head$ of the list.

A PUSH operation simply invokes the wait-free ENLIST operation to append a *PUSH* node at the head of the list. A POP operation first appends a *POP* node by invoking ENLIST, and then traverses the list to find the "matching" *PUSH* node as its return value. The traversal keeps track of a local depth counter which is incremented at a *POP* node and decremented at a *PUSH* node, and returns the value of the current node if the counter is changed from 1 to 0.

The linearizability of the implementation is straightforward to verify: each PUSH and POP operation linearizes at the point when the ENLIST takes effect. A POP operation always returns the value of its uniquely matched *PUSH* node (or **nil**), which cannot change after the ENLIST operation takes effect. The (possibly empty) subsequence of nodes between a *POP* node and its matched *PUSH* node represents a (possibly nested) sequence of matching PUSH and POP operations.

```
datatype NODE
  $key$    : $\mathbb{N}$       // integer data field
  $state$  : $\mathbb{N}$       // *PUSH* or *POP*
  $next$   : NODE    // pointer to the successor
  $prev$   : NODE    // pointer to the predecessor
  $tid$    : $\mathbb{N}$       // thread id of the creater

global variables
  $head$   : NODE    // initially **nil**

function PUSH($k$ : $\mathbb{N}$)
    $h \leftarrow$ new NODE$\langle k, \mathit{PUSH}, \textbf{nil}, \textbf{nil}, \textbf{threadid} \rangle$
    ENLIST($h$)

function POP($k$ : $\mathbb{N}$) : $\mathbb{N}$
    $h \leftarrow$ new NODE$\langle k, \mathit{POP}, \textbf{nil}, \textbf{nil}, \textbf{threadid} \rangle$
    ENLIST($h$)
    $d \leftarrow 1$
    $curr \leftarrow h$
    while $curr \neq$ nil do
        if $curr.state = \mathit{POP}$ then
            $d \leftarrow d + 1$
        else
            $d \leftarrow d - 1$
        if $d = 0$ then
            $h.next \leftarrow curr$
            return $curr.key$
        $curr \leftarrow curr.next$
    return $\perp$
```

Figure 10: A Wait-free Stack Implementation