Technical Report no. 2016-02

Help-optimal and Language-portable Lock-free Concurrent Data Structures

Bapi Chatterjee *Ivan Walulya* *Philippas Tsigas*

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2016

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report no. 2016-02
ISSN: 1652-926X

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2016

Abstract

Helping is the most common mechanism to guarantee lock-freedom in many concurrent data structures. An optimized helping strategy improves the overall performance of a lock-free algorithm. In this paper, we propose *help-optimality*, which essentially implies that no operation step is accounted for exclusive helping in the lock-free synchronization of concurrent operations. To describe the concept, we revisit the designs of a lock-free linked-list and a lock-free binary search tree and present improved algorithms. Our algorithms employ atomic single-word compare-and-swap (`CAS`) primitives and are linearizable.

Additionally, we do not use a language/platform specific mechanism to modulate helping, specifically, we use neither bit-stealing from a pointer nor runtime type introspection of objects, making the algorithms *language-portable*. Further, to optimize the amortized number of steps per operation, if a `CAS` execution to modify a shared pointer fails, we obtain a fresh set of thread-local variables without restarting an operation from scratch.

We use several micro-benchmarks in both C/C++ and Java to validate the efficiency of our algorithms against existing state-of-the-art. The experiments show that the algorithms are scalable. Our implementations perform on a par with highly optimized ones and in many cases yield 10%-50% higher throughput.

Keywords: concurrent data structure, linked-list, binary search tree, lock-free, linearizability, help, language-portable

Help-optimal and Language-portable Lock-free Concurrent Data Structures

Bapi Chatterjee, Ivan Walulya and Philippas Tsigas
Chalmers University of Technology, Gothenburg, Sweden
{bapic, ivanw, tsigas}@chalmers.se

Abstract—**Helping is the most common mechanism to guarantee lock-freedom in many concurrent data structures. An optimized helping strategy improves the overall performance of a lock-free algorithm. In this paper, we propose *help-optimality*, which essentially implies that no operation step is accounted for exclusive helping in the lock-free synchronization of concurrent operations. To describe the concept, we revisit the designs of a lock-free linked-list and a lock-free binary search tree and present improved algorithms. Our algorithms employ atomic single-word compare-and-swap (`CAS`) primitives and are linearizable.**

Additionally, we do not use a language/platform specific mechanism to modulate helping, specifically, we use neither bit-stealing from a pointer nor runtime type introspection of objects, making the algorithms *language-portable*. Further, to optimize the amortized number of steps per operation, if a `CAS` execution to modify a shared pointer fails, we obtain a fresh set of thread-local variables without restarting an operation from scratch.

We use several micro-benchmarks in both C/C++ and Java to validate the efficiency of our algorithms against existing state-of-the-art. The experiments show that the algorithms are scalable. Our implementations perform on a par with highly optimized ones and in many cases yield 10%-50% higher throughput.

Index Terms—**concurrent data structure; linked-list; binary search tree; lock-free; linearizability; help; language-portable;**

I. Introduction

A. Overview

With the ubiquity of multi-core processors, efficient concurrent data structures have become ever more important. Lock-free concurrent data structures, which guarantee that some non-faulty threads finish their operations in a finite number of steps, provide robustness and better performance compared to their blocking counterparts which are vulnerable to pitfalls such as deadlocks, priority inversion and convoying in an asynchronous shared memory system.

The literature on lock-free data structures has grown sufficiently over the last decade [1]–[12]. Typically, *practical* lock-free designs use single-word atomic compare-and-swap (`CAS`) synchronization primitive (henceforth referred to as `CAS`) to modify shared variables. Thus to implement a lock-free version of a dynamic pointer-based data structure, in which (multiple) mutable links (pointers) are shared among threads in a concurrent set-up, either by design or due to necessity, one or more `CAS` executions are performed to complete a modify (add or remove) operation.

For example, in the lock-free linked-list of [1], two successful `CAS` executions are required to complete a remove operation, whereas in [3] three successful `CAS` steps are required for the same operation. In lock-free external binary search trees (BST) - three successful `CAS` executions are necessary to remove a node in [7], whereas in [4] and [8], four such executions are required for the same purpose. Furthermore, in [4] and [8], two successful `CAS` executions are required to add a node. Naturally, concurrent operations which modify overlapping sets of links, face each other at a stage where they would have partially completed and would still need to perform one or more `CAS` to complete. Generally, we call this situation *concurrent obstruction*.

For operations on a concurrent data structure, linearizability [13] is the most commonly used consistency framework. Intuitively, a concurrent data structure is linearizable if every execution provides time-points, called *linearization points*, between the invocation and response of each operation, where it seems to take effect instantaneously. Thus, using a sequence of seemingly instantaneous operations, described by the *real-time order* of the linearization points, we perceive the concurrent operations displaying their sequential behaviour.

In a lock-free algorithm, often a `CAS` step is taken as the linearization point of an operation performing multiple `CAS`. Such a `CAS` step may not necessarily be the last one. Most commonly in a remove operation, on the success of the `CAS` representing the linearization point, the target node is considered *logically* removed, [1]–[5]. This results in each traversal passing through a logically removed node and hence extra read steps get counted in step complexity of operations.

A well-known methodology to deal with such situations is *helping*. Helping essentially implies that if multiple operations face concurrent obstruction, or need to perform extra read while traversing over a transient deformation in form of a logically removed node, then based on a pre-decided protocol, the pending steps of one of the operations are completed by the concurrent operations, before furthering their own course of steps. This strategy ensures lock-freedom because an operation of some non-faulty thread definitely progresses with the steps performed by the thread.

In the prevalent research of lock-free data structure design, the helping strategy now holds a center stage. In the lock-free linked-lists of [1], [3], every concurrent operation offers helping to a remove operation which successfully performs the `CAS` to logically remove the target node and is yet to execute one more `CAS`. Barnes [14] proposed a helping strategy called *cooperative technique*. The cooperative technique applied to a data structure requires a modify operation to atomically write the description of planned steps in the node whose links it targets to modify and thereby a concurrent obstructed operation ensures completion of those steps in case the original operation gets delayed. This method is applied in the BST of [4], [8], where even add operations require helping.

In the lock-free BST of Natarajan et al. [7], the links are used much in the same way as in the linked-lists of [1], [3] to modulate helping, and surely the add operations do not require help. In general, their design provides better progress conditions for concurrent operations, as observed experimentally in [7]. However, we can notice that they put the linearization point of a remove operation at the very last `CAS` step, which necessitates a concurrent remove operation to help a pending remove operation of the same query key, even though it does not change its return which is false. Clearly, the number of helping steps are not minimized in this design.

In general, authors of lock-free data structure papers suggest that one should avoid helping during traversal by an otherwise unobstructed operation, which in case the obstructing operation is not delayed, predominantly goes to wastage. The works analysing experimental performance of concurrent data structures [10], [11] further emphasise on the same. Gibson et al. [15] showed that the amortized number of steps per operation are asymptotically equivalent irrespective of avoiding help by read operations. It becomes more pertinent with multi-core processors offering an increasing number of cores, and thus making it likely that enough threads accessing a concurrent data structure, proceed without being pre-empted by the operating system scheduler. Yet a design optimization to reduce the number of helping steps by modify operations at concurrent obstruction is largely unattempted.

Another noticeable characteristic of existing lock-free algorithms is that the design descriptions are very close to the programming language of the sample implementations used by the authors to validate their claim of efficiency. For example, in the linked-lists of [1], [3] and the BST of [7], the design is described in terms of using unused bits from a pointer which points to a memory-word aligned at a fixed boundary. This technique is popularly known as *bit-stealing* in programming parlance. The correctness proof thereof is inherently connected to bit-stealing. The lock-free external BST designs of [4], [8] use polymorphism, class inheritance and type introspection of objects at runtime (also known as real-time-type-information or RTTI), to describe their algorithm.

In Java toolkit [16], `AtomicMarkableReference` and `AtomicStampedReference` classes are used to simulate bit-stealing. In the lock-free skip-list implementation in Java [17], Doug Lea uses extra *splice nodes* to simulate the pointers masked with stolen bits. Such a node is identified with a specific assignment of one of its fields, for example, the value field of a *marker* node points to itself in [17]. A marker node stores the original pointer in its next field enabling unmasking of the bit-stolen pointer. Lea remarks that in spite of some temporary extra nodes, this technique could still be faster for a traversal with quick garbage collection of removed nodes and is worth avoiding the overhead of extra type testing.

Usually, the lock-free implementations in C/C++, for example in [11] or [6], use their own memory allocation and garbage collection strategies to improve performance. Obviously, these implementation environments of C/C++ very much resemble one in Java and yet they entail each traversal step to unmask a pointer off a possible stolen bit. This underlies a motivation to present lock-free algorithms that utilize temporary splice nodes and thereby achieving *language portability*. However, the efficiency of such an implementation in C/C++ remains still unexplored for the research community.

In literature, the efficiency of a lock-free algorithm is also presented in terms of the amortized step complexity per operation [3], [8], [9]. Often in a lock-free data structure, when a `CAS` execution in a modify operation returns false, the local variables in the thread become unusable for a reattempt. Hence, the thread needs to restart the operation from a *clean location* to get a fresh set of local variables. Ordinarily, the first sentinel node where an operation starts from (head of a linked-list, root of a BST), is always clean. However, there can be as many as c restarts per operation if c concurrent threads access the data structure. Getting a pointer to backtrack to a *local* clean location and restarting the operation from there, improves the amortized number of steps per operation (counting both read and write). It can be interesting to use splice nodes to store a pointer to a node in a local clean location and thus *locally restart* a modify operation.

The contributions of this work are the following:

1) We introduce the concept of *help-optimality* which essentially revisits the lock-free algorithms to minimize the number of `CAS` steps in helping at concurrent obstructions.
2) We describe help-optimal lock-free designs of a linked-list and a BST to implement Set abstract data types (ADT) which export linearizable ADD, REMOVE, and CONTAINS operations. CONTAINS are wait-free for a finite key space.
3) Our algorithms do not use language specific constructs like bit-stealing or type introspection of objects at runtime and hence are *language-portable* for a programmer.
4) On a `CAS` failure at a conflict, the modify operations in our algorithms restart locally to optimize the amortized step complexity per operation
5) We implement the algorithms in both C++ and Java. Our implementations perform on a par with highly optimized implementations and outperform them in many cases.

B. Related Work

The first `CAS`-based lock-free linked-list was presented by Valois [18]. He suggested to augment every node with an auxiliary node to manage synchronization. Heller et al. [19] were perhaps the first to suggest that the CONTAINS operations in a concurrent linked-list must progress in a wait-free manner for a finite key space. They presented lock-based linked-list, called lazy list, to show that it favours performance. They also recommended that the CONTAINS operations in Michael's lock-free linked-list [2] should not be involved in helping. Subsequently, to the best of our knowledge, no concurrent data structure was designed in which CONTAINS operations are obstructed; interestingly, some researchers called it *conservative helping*, for example in [4]. In the lock-free internal BSTs presented by Howley et al. [5], Chatterjee et al. [9] and Ramachandran et al. [12] CONTAINS operations complete without helping any concurrent operation.

```
1   class Node {K k; NdPtr lt, rt;};
    //NdPtr: A pointer to a Node.
2   root := Node($\infty_1$, Node($\infty_0$).ref, Node($\infty_1$).ref);
3   Dir(NdPtr par, K k) {return k < par.k ? L : R};
    Child(NdPtr par, dir cD)
4       return cD == L ? par.lt : par.rt;
    ChCAS(NdPtr par, NdPtr exp, NdPtr new, dir cD)
5       if (cD == L) and par.lt == exp then
6           return CAS(par.lt.ref, exp, new);
7       else if (cD == R) and par.rt == exp then
8           return CAS(par.rt.ref, exp, new);
9       else return false;
10  GetDead(K k) {n := Node(k); n.rt := n; return n;}
11  IsDead(NdPtr leaf) {return leaf.rt == leaf;}

    Search(NdPtr par, NdPtr leaf, K k)
12      while leaf.lt $\neq$ null do
13          par := leaf; leaf := Child(par, Dir(par, k));
    REMOVE(K k)
14      p := root.ref; $\ell$ := root.lt;
15      while true do
16          Search(p.ref, $\ell$.ref, k); cD := Dir(p, k);
17          if $\ell$.k $\neq$ k or IsDead($\ell$) then return false;
18          if ChCAS(p, $\ell$, GetDead(k), cD) then
19              return true;
20          $\ell$ := Child(p, Dir(p, k));
    CONTAINS(K k)
21      p := root.ref; $\ell$ := root.lt;
22      Search(p.ref, $\ell$.ref, k); cD := Dir(p, k);
23      return $\ell$.k == k and !IsDead($\ell$);

    NewNod(NdPtr a, NdPtr b, K pKey)
24      left := (a.k < b.k ? a : b);
25      right := (a.k < b.k ? a : b);
26      return Node(pKey, left, right, null);
    ADD(K k)
27      nd := Node(k).ref;
28      p := root.ref; $\ell$ := root.lt;
29      while true do
30          Search(p.ref, $\ell$.ref, k); cD := Dir(p, k);
31          if !IsDead($\ell$) then
32              if $\ell$.k == k then return false;
33              n := NewNod(nd, $\ell$, max{k, $\ell$.k}).ref;
34              if ChCAS(p, $\ell$, n, cD) then return true;
35          else if ChCAS(p, $\ell$, nd, cD) then return true;
36          $\ell$ := Child(p, Dir(p, k));
```

Algorithm 1. Basic Help-optimal Language-portable Lock-free Binary Search Tree

Roadmap: In section II, we present a simple lock-free BST algorithm as a motivation for a help-optimal design. In sections III and IV, we present efficient lock-free algorithms of a linked-list and a BST, to describe the concept of help-optimally used in practice. Having described it algorithmically, we specify help-optimality more formally in section V. We discuss the experimental performance of the presented algorithms in section VI. Section VII concludes the paper.

II. Help-optimality: Motivation

Let us consider a very simple lock-free design of an external BST to implement a Set ADT exporting ADD, REMOVE, and CONTAINS operations as given in algorithm 1.

In this data structure, a node has two pointer fields lt and rt in addition to a key field k, see line 1. Without ambiguity, we shall use k to denote a node with key k. The pointer fields lt and rt connect a node to its left and right children respectively, which are null in a leaf (also called external) node. In this BST, the external nodes are *data-nodes* and the internal nodes are *routing-nodes*. There is a *symmetric order* of node-arrangement - the nodes in the *left subtree* of a routing-node k have keys less than k, whereas in its *right subtree* the nodes have keys at least k. We denote the parent of a node k by $p(k)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected to its children via links (we indicate the link emanating from k and incoming to l by $k \rightsquigarrow l$; we use the terms pointer and link interchangeably). The other child of $p(k)$, i.e. sibling of k, is denoted by $s(k)$.

Pseudo-code convention: N.*ref* represents the reference to a variable N. Thus, f(N.*ref*) indicates passing N by reference to a method f. If x is a member of a class C then pc.x returns field x of an instance of C pointed by pc. dir L and dir R represent the left and right directions. CAS(A.*ref*, exp, new) compares A with exp and updates to new in one atomic step if A == exp and returns true; else it returns false without any update at A.

We initialize the BST with a subtree consisting of an internal node root with key ∞_1 and two children with key ∞_0 and ∞_1, where $\infty_1 > \infty_0 > k \ \forall$ key k, as its left- and right- child respectively, see fig. 1 and line 2. The method Search, line 12 to 13, is used for traversal by a data structure operation. Search takes variables *par*, *cur* and k as input which are two node-pointers and a query key, respectively. At the invocation of Search, *cur* points to the child of the node pointed by *par* in the direction of the subtree which can contain k. At the termination of Search, *cur* points to a leaf-node which is identified by the lt field being null.

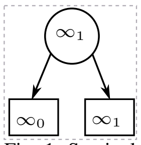


Fig. 1: Sentinel

To perform REMOVE(k), line 14 to 20, we use Search to arrive at a leaf-node pointed by ℓ. If k matches the key at ℓ, we use a CAS to replace ℓ with a special node with the same key, but with its rt field pointing to itself, see line 18. We call such special nodes Dead. See method IsDead at line 11 which is used to identify a Dead node. If the CAS succeeds, REMOVE returns true; if k was not found or ℓ was already Dead, REMOVE returns false. For ADD(k), line 27 to 36, arriving at ℓ using Search, we use a CAS to replace ℓ with (i) a new leaf-node with key k, if ℓ was Dead and (ii) a new internal node created using NewNod, line 24 to 26, if ℓ was not Dead and k does not match at ℓ. If the CAS succeeds at line 34 or at line 35, ADD returns true; if ℓ was not Dead and contained k, it returns false. A CONTAINS(k), line 21 to 23, returns true if k is found at a leaf-node which is not Dead, else it returns false.

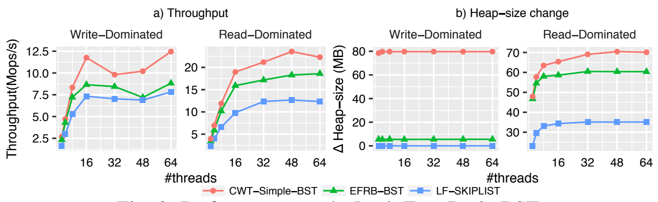


Fig. 2: Performance graph: Lock-Free Basic BST

The main idea of this algorithm is to discard the requirement of helping by *not* cleaning out a node in a REMOVE operation, which otherwise uses multiple CAS executions. Thus, a single successful CAS is required by both ADD and REMOVE operations, much like a lock-free stack. We skip the proofs of correctness and lock-freedom of this algorithm, which are straightforward. An interested reader may take them as a simple exercise. Please note that we have not used any language specific construct to describe this algorithm.

We implemented algorithm 1 in Java and compared it against the (author provided) implementation of lock-free BST of [4] and the lock-free skip-list of Java library [17]. The set-up and methodology of the experiments are described in section VI. The throughput and memory usage by the algorithms to implement a Set formed by at most 2^{20} distinct keys are plotted in fig. 2a and 2b, respectively.

We see that this simple lock-free BST handsomely outperforms state-of-the-art implementations of a skip-list and a BST.

```
1  class Node {K k; NdPtr nxt, bck;};
2  tailNxt := Node($\infty_1$, null, null);
3  tail := Node($\infty_0$, tailNxt.ref, null);
4  headNxt := Node($-\infty_0$, tail.ref, null);
5  head := Node($-\infty_1$, headNext.ref, null);
   Search(NdPtr pre, NdPtr nex, NdPtr cur, NdPtr suc, K k)
6      while cur.k < k do
7          if IsSp(suc) then cur := suc.nxt;
8          else pre := cur; nex := suc; cur := suc;
9          suc := cur.nxt;
   BckTrck(NdPtr pre, NdPtr nex)
10     nex := pre.nxt;
11     while IsSp(nex) do
12         pre := nex.bck; nex := pre.nxt;
13 IsSp(NdPtr c)  {return c.k == $-\infty_2$;}

14 ADD(K k)
15     p := head.ref; n := headNxt.ref;
16     c := headNxt.ref; s := headNxt.nxt;
17     while true do
18         Search(p.ref, n.ref, c.ref, s.ref, k);
19         if IsSp(s) then
20             while IsSp(s) do
21                 c := s.nxt; s := c.nxt;
22         else if c.k == k then return false;
23         if CAS(p.nxt.ref, n, Node(k, c, null)) then
24             return true;
25         BckTrck(p.ref, n.ref); c := p; s := n;
   CONTAINS(K k)
26     c := headNext.nxt;
27     while c.k < k do c := cur.nxt;
28     return c.k == k and !IsSp(c.nxt);

   REMOVE(K k)
       //Initialize the variables p,n,c,s.
29     r := null; spNd := null; mode := INIT;
30     while true do
31         Search(p.ref, n.ref, c.ref, s.ref, k);
32         if mode == INIT then
33             if c.k $\neq$ k or IsSp(s) then return false;
34             spNd := Node($-\infty_2$, s, p).ref;
35             while true do
36                 if CAS(c.nxt.ref, s, spNd) then
37                     if CAS(p.nxt.ref, n, s) then return true;
38                     r := s; mode := CLEAN; break;
39                 s := c.nxt; if IsSp(s) then return false;
40                 spNd.nxt := s;
41         else if s $\neq$ spNd or CAS(p.nxt.ref, n, r) then
42             return true;
43         BckTrck(p.ref, n.ref); c := p; s := n;
```

Algorithm 2. Help-optimal lock-free linked-list

However, on account of memory footprint, it performs poorly. We get enough motivation to design lock-free data structures which maximally reduces the number of CAS executions by each ADT operation aided with optimal memory footprints.

III. HELP-OPTIMAL LOCK-FREE LINKED-LIST

A. Design

We implement a linked-list based Set ADT which exports ADD, REMOVE and CONTAINS operations and in which nodes are arranged in increasing order of unique keys. The pseudo-code is given in Algorithm 2. A node has two pointer fields nxt and bck in addition to the key field k. Without ambiguity, we shall use k to denote a node with key k. The field nxt points to the successor of k, denoted by $s(k)$. We describe the use of bck later; it is null in a regular node. The predecessor of k is denoted by $p(k)$. On initialization, the linked-list consists of four sentinel nodes tailNxt, tail, headNxt and head with keys ∞_1, ∞_0, $-\infty_0$ and $-\infty_1$, respectively, where $\infty_1 > \infty_0 > k \; \forall$ key k. See line 2 to 5 and fig. 3.

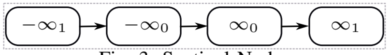


Fig. 3: Sentinel Nodes

We aim to reduce the number of CAS steps for helping not only during traversal, which is simple but also in the concurrent obstruction. In algorithm 1 we observed that obstruction can be fully avoided in an external BST if REMOVE operations do not try to clean out the removed nodes. However, that strategy led to large memory footprint. So, the question we ask - can we overcome the drawbacks? Observing carefully, in a linked-list we can leverage the linear structure to connect the predecessor of the leftmost node to the successor of the rightmost node of a contiguous chunk of removed nodes by a single CAS and thus solve the issue. We describe it below.

At the basic level, ADD(k) in a lock-free linked list comprises - finding $p(k)$ and $s(k)$ s.t. $p(k).\mathsf{k} < k < s(k).\mathsf{k}$, allocating the node k s.t. $k.\mathsf{nxt} = s(k)$ and using a single CAS execution to swing the $p(k).\mathsf{nxt}$ from $s(k)$ to k. We have seen it in [1], [3]. Similarly, REMOVE(k) comprises - finding nodes $p(k)$ and k, logically removing k using a single CAS and then swing the $p(k).\mathsf{nxt}$ from k to $s(k)$ using a CAS.

However, our aimed implementation as described before will require additional tricks over this basic idea. Firstly, in order to make the algorithm language-portable, we shall go the way of using *splice nodes* as Lea [17], instead of bit-stealing like [1], [3]. Thus to logically remove k, we add a splice node between k and $s(k)$. We fix the key of a splice node as $-\infty_2$, where $\infty_2 > \infty_1$, by which it can be identified, see line 34 and the method IsSp at line 13.

Secondly, to avoid eager helping during traversal by a modify operation and yet be able to clean out the logically removed nodes (along with the splice nodes succeeding them), we use two trailing node-pointers during traversal. This trick is similar to [7] applied in BST. We use them to store the address of the last node, which was *not* logically removed, and its successor. Thus, at the termination of a traversal, we can have reference to the predecessor, say $p(k)$, of the leftmost node of a contiguous chunk of logically removed nodes. Thus, when we use a CAS to swing the nxt pointer of $p(k)$ to connect either to the new node k for ADD or to $s(k)$ for REMOVE, zero or more logically removed nodes are cleaned out.

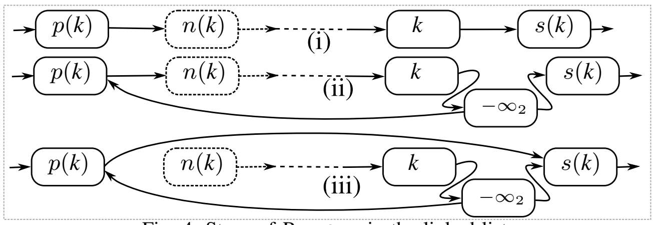


Fig. 4: Steps of REMOVE in the linked-list.

And thirdly, to backtrack to a clean zone on CAS failures, we use the idea of back-pointers as applied in [3]. When allocating a splice node, we save the address of $p(k)$ in its bck field which is always null for a regular node. Thus, our approach differs from [3] in the following ways - (a) We do not use an extra CAS to fix (*flagging*) the pointer $p(k).\mathsf{nxt}$. Given the use of trailing pointers, we do not often travel a long chain of back pointers. And (b) we do not set back-pointer of a regular node and thus, save an extra atomic write of a shared pointer. Indeed, a splice node in our algorithm splices two node paths.

The basic steps of REMOVE(k), are shown in the fig. 4. The node $n(k)$ denotes the first node of a possible contiguous chunk of logically removed nodes before and adjacent to k. In case there is no such chunk before k, $n(k)$ coincides with it.

We perform traversal for a modify operation using the method Search, line 6 to 9. We advance the variables *pre* and *nex* only if *suc* is not a splice node, that is when *cur* is not a logically removed node. Otherwise, we advance *cur* to the node saved at the *nxt* of the splice node *suc*. In REMOVE and ADD, at the first call of Search, the variable *pre* points to head, *nex*, *cur* point to headNxt and *suc* points to the successor of headNext, see lines 15 and 16. Thus, at the termination of a traversal, when *cur* points to a node with key *not greater than k*, *pre* points to the predecessor of the first node of a possible contiguous chunk of logically removed nodes and *nex* points to the first node of such a chunk.

To perform REMOVE(k), line 29 to 43, at the termination of a traversal, we check the key at the node pointed by c, and if k does not match at it or the node pointed by s is found splice (indicating node pointed by c is already logically removed), we return false, line 33. Otherwise, we perform a `CAS` to add a splice node between c and s to logically remove c, line 36. The steps taken up to this point are identified by a variable mode with value `INIT`. After this step, mode changes to `CLEAN` and we attempt to swing the p.nxt from n to s using a `CAS` at line 37. If the `CAS` fails, we save s as r, and perform a `BckTrck` at line 43 to find a fresh pair of p and n.

In the method `BckTrck`, line 10 to 12, we keep on traversing the bck of splice nodes until we find the first node which is not logically removed. If the call of `BckTrck` was due to a `CAS` failure caused by an ADD of a new node, added between *pre* and *nex*, then it is guaranteed that the chunk of contiguous logically removed nodes must have been cleaned out. We explain it in the next paragraph.

The operation ADD(k), line 15 to 25, performs a similar traversal. At the termination of the traversal, we check if the node pointed by c is logically removed by checking whether s points to a splice node, line 19. If the node at c is not logically removed and contains the query key k, we return false; else, we find the first node succeeding it which is still not logically removed, line 21, and attempt a `CAS` to add the new node between p and c to return true. On a `CAS` failure, we perform `BckTrck` as explained before and reattempt the previous steps. Thus, on a successful ADD it cleans out a complete chunk of contiguous logically removed nodes.

Note that, a modify operation in algorithm 2 differs from one in [1], [3], in the sense that on a `CAS` failure at $p(k)$.nxt, we do not perform any help before reattempting rather selfishly attempt the `CAS` from a fresh location. The operations are essentially *selfish* in our algorithm.

A CONTAINS operation, line 26 to 28, traverses in a wait-free manner and returns true only if the node at which it terminates, the one pointed by c, is not logically removed and contains the query key, else it returns false.

B. Correctness and Lock-freedom

It is easy to observe that the k field of a node is never modified after initialization. Scanning through the pseudo-code, we can observe that once a splice node is added at the nxt of a node, no `CAS` is performed at it. Further, unless the nxt of a node k is splice, it is not removed from the list. Thus, we can show that a node $p(k)$ is present in the list, when we connect a new node k or successor $s(k)$ of a removed node k to it. And, we can observe that a traversal terminates with c pointing to a node which has a key greater than or equal to k in all the operations, which in turn shows that we maintain the order of node arrangement in an ADD or a REMOVE operation. At the initialization, the sentinel nodes form a valid ordered list. Hence, using induction we can prove that the ADT operations maintain a valid ordered list.

The linearization point for an unsuccessful ADD operation and a successful CONTAINS operation is at line 9 during a call of `Search` and at line 28, respectively, when we read c.nxt. For a successful ADD or a REMOVE operation, the linearization point lies at the first successful `CAS` execution to add a new or a splice node. For an unsuccessful CONTAINS, the linearization point is (a) just after that of the concurrent REMOVE operation which (logically) removed k, if k existed in the list at the invocation point and (b) at the invocation point itself, if k was not present in the list at that point. The linearization point of an unsuccessful REMOVE is determined similarly to an unsuccessful CONTAINS operation.

We can observe that the `CAS` to add a splice node is reattempted only if a new node is added at the nxt of k. Before every reattempt of a `CAS` to swing the nxt pointer of $p(k)$, in both ADD and REMOVE, we perform a `BckTrck` and a `Search` which guarantees that we have a fresh set of variables for references of $p(k)$ and $n(k)$. Hence, it is guaranteed that a modify operation can not take an infinite number of steps without a modification in the data structure. It shows the lock-freedom of the ADD and REMOVE operation. It is easy to observe that a non-faulty CONTAINS always finishes in a finite number of steps if the key space is finite and thus is wait-free.

C. Amortized Step Complexity

We can observe that the splice nodes are never adjacent. Similar to [10], we do not perform help in a CONTAINS operation. Additionally, in ADD and REMOVE as well, no step is taken for helping during traversal. On a `CAS` failure to add a splice node, we do not perform any traversal. On a `CAS` failure to add a new node or to clean out a chunk of logically removed nodes, we perform backtrack and do not start from the head. Following the same method as [3], we can show that the amortized number of steps per operation is $O(n+c_I)$, where c_I is the total number of concurrent operations between invocation and response of o, called *interval contention* [20] and n is the size of list at the invocation of o. In the light of theorem 1 of [15], it is equivalent to $O(n+c_P)$, where c_P is the maximum number of concurrent operations at any point in the lifetime o, called *point contention* [21].

IV. HELP-OPTIMAL LOCK-FREE BST

Having described a simple lock-free BST and an improved lock-free linked-list, where we do not spend any `CAS` execution for helping, we are ready to describe an efficient lock-free BST, in which we introduce the notion of *help-awareness*.

A. Design

The pseudo-code of the design is given in algorithm 3. The symmetric order of the BST is same as that in section II. We borrow the notations from algorithm 1 along with the methods `Dir`, `Child`, `ChCAS`, `GetDead`, `IsDead` and `NewNod` as they are described there. We denote the parent of $p(k)$ by $g(k)$.

The main drawback of the lock-free BST of algorithm 1 was removing a node k by replacing it with a `Dead` node and not cleaning the `Dead` node out. It caused memory-wastage. Therefore, in algorithm 3 we make a REMOVE operation clean out the added `Dead` node. Consequently, the ADD operations

```
1  class Node {K k; NdPtr lt, rt, bck;};
2  root := Node($\infty_1$); grRoot := Node($\infty_0$);
3  root.lt := Node($\infty_2$).ref; root.rt := Node($\infty_1$).ref;
4  grRoot.lt := root.ref; grRoot.rt := Node($\infty_0$).ref;
   Search(NdPtr gPar, NdPtr nex, NdPtr par, NdPtr leaf, K k)
5    while leaf.lt $\neq$ null do
6      if IsSp(leaf) then par := leaf.rt;
7      else gPar := par; nex := leaf; par := leaf;
8      leaf := Child(par, Dir(par, k));
   GetNxt(NdPtr leaf)
9    return IsSp(leaf) ? leaf.rt : leaf;
10 GetKey(NdPtr leaf) {return GetNxt(leaf).k;}
   GetDeadBl(NdPtr gPar, K k)
11   n := GetDead(k); n.bck := gPar; return n;
   GetSp(NdPtr gPar, NdPtr leaf)
12   if IsDead(leaf) then
13     return GetDeadBl(gPar, leaf);
14   else return Node($-\infty_3$, leaf.lt, leaf, gPar);
   AddSp(NdPtr par, NdPtr gPar, dir sD)
15   while true do
16     sib := Child(par, sD);
17     if IsBl(sib) then return sib;
18     else if ChCAS(par, sib, GetSp(gPar, sib), sD) then
19       return sib;
20 IsSp(NdPtr leaf) {return leaf.k == $-\infty_3$;}
   REMOVE(K k)
21   g := grRoot.ref; n := root.ref;
22   p := root.ref; $\ell$ := root.lt;
23   dNdBl := null; sib := null; mode := INIT;
24   while true do
25     Search(g.ref, n.ref, p.ref, $\ell$.ref, k);
26     cD := Dir(p, k); pD := Dir(g, k);
27     if mode == INIT then
28       if GetKey($\ell$) $\neq$ k or IsDead($\ell$) then
29         return false;
30       dNd := GetDead(k);
31       if !IsSp($\ell$) and p $\neq$ g then
32         dNdBl := GetDeadBl(g, k);
33         if ChCAS(p, $\ell$, dNdBl, cD) then
34           sib := AddSp(p, g, !cD); mode := CLEAN;
35           if IsSp(sib) then return true;
36           else if IsDead(sib) then
37             ChCAS(g, n, dNd, pD); return true;
38           else if ChCAS(g, n, sib, pD) then return true;
39       else if ChCAS(g, n, dNd, pD) then return true;
40     else
41       if $\ell$ == dNdBl and p $\neq$ g then
42         if ChCAS(g, n, sib, pD) then return true;
43       else return true;
44     BckTrck(g.ref, n.ref, k); p := g; $\ell$ := n;
   BckTrck(NdPtr gPar, NdPtr nex, K k)
45   nex := Child(gPar, k);
46   while IsSp(nex) do
47     gPar := nex.bck; nex := Child(gPar, k);
48 IsBl(NdPtr leaf) {return leaf.bck $\neq$ null;}
   ADD(K k)
49   g := grRoot.ref; n := root.ref;
50   p := root.ref; $\ell$ := root.lt; nd := Node(k).ref;
51   while true do
52     Search(g.ref, n.ref, p.ref, $\ell$.ref, k);
53     cD := Dir(p, k); pD := Dir(g, k);
54     if !IsDead($\ell$) then
55       if GetKey($\ell$) == k then return false;
56       nl := NewNod(nd, GetNxt($\ell$), $\frac{k+\text{GetKey}(\ell)}{2}$).ref;
57       if IsSp($\ell$) then
58         if ChCAS(g, n, nl, pD) then return true;
59       else if ChCAS(p, $\ell$, nl, cD) then return true;
60     else
61       if IsBl($\ell$) then
62         sib := AddSp(p, g, !cD);
63         if !IsDead(sib) then
64           nl := NewNod(nd, GetNxt(sib), $\frac{k+\text{p.k}}{2}$).ref;
65           if ChCAS(g, n, nl, pD) then return true;
66         else if ChCAS(g, n, nd, pD) then return true;
67       else if ChCAS(p, $\ell$, nd, cD) then return true;
68     BckTrck(g.ref, n.ref, k); p := g; $\ell$ := n;
```

Algorithm 3. Help-optimal lock-free binary search tree

will have to be aligned with the REMOVE operations which now make structural changes in the BST.

In a sequential set-up, removing a node k from an external BST is a one step process of modifying the link $g(k)\rightsquigarrow p(k)$ to connect $s(k)$ to $g(k)$. This process also cleans out the removed node. It essentially removes the node $p(k)$ from the (unordered) linked-list described by the nodes on the path from the root to $s(k)$. Thus, to perform REMOVE(k) with cleaning out k in a lock-free BST can be visualized as a two stage process - (a) single CAS to logically remove k by replacing it with a Dead node as in algorithm 1 and (b) two CAS steps to remove $p(k)$ - adding a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ and then swinging the pointer $g(k)\rightsquigarrow p(k)$ to connect $s(k)$ to $g(k)$, as in algorithm 2. Let us call these stages LREMOVE and PREMOVE, respectively. This understanding gives us the fundamental idea of algorithm 3.

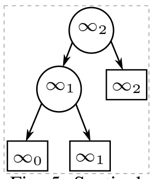


Fig. 5: Sentinel

LREMOVE is quite straightforward. Now, to perform PREMOVE efficiently, along the lines of algorithm 2, we carry two trailing node-pointers during the traversal for a modify operation. Thus, the method Search in algorithm 3, line 5 to 8, becomes a blend of the same method in the previous two algorithms. At the termination of Search, the variable *gPar* points to the parent of the root of the sub-tree in which all the nodes are logically removed. To avoid special cases arising in placing the trailing node-pointers in an empty BST, we use a set of sentinel nodes as shown in line 2 to 4 and fig. 5.

We assign key $-\infty_3$ for a splice node, such that $\infty_3 > \infty_2 > \infty_1 > \infty_0 > k\ \forall$ key k. It ensures that at a splice node a traversal always *goes right*. Hence, we connect $s(k)$ to rt of a splice node. We copy the lt field of $s(k)$ to the splice node that it connects to, which if null, indicates that $s(k)$ is a leaf node. Thus, a traversal may terminate at a splice node. Considering that, we always use the method GetNxt to access the actual leaf node, see line 9; and following that the method GetKey gives the key at that leaf node, see line 10. Also, to achieve local restart as in algorithm 2, we include a bck pointer in the node structure to implement splice nodes which can provide reference to a node in a local clean zone.

Now consider the following two cases:

(A) An ADD operation o just before performing its CAS step gets pre-empted by the operating system scheduler. Let g be the trailing node-pointer pointing to the last internal node of the traversal path which is not logically removed. Suppose that, by the time o wakes up, the BST changes in a way that both the children of the node pointed by g are replaced by Dead nodes and the node itself cleaned out of the BST. Consequently, o will have *no* link to reach a clean zone except restarting from the root of the BST, which we want to avoid. To tackle this issue, we use the bck pointer of a Dead node, which replaces a node k in a REMOVE operation, to store g. We call a Dead node with a non-null bck field a DeadBl node.

(B) Two concurrent REMOVE operations o_1 and o_2, at the end of their traversal, target to remove two leaf nodes k_1 and k_2, which are children of the same internal node, say p. Also suppose that o_1 and o_2 have same pair of trailing node-pointers - g and n - in their thread-local memory and thus for both o_1 and o_2 there is access to no link to backtrack above g in the BST. Suppose that LREMOVE stage of both o_1 and o_2 finished without contention, and thus after that both the children of p are DeadBl. Therefore, after its PREMOVE, if o_1 successfully connects the DeadBl node k_2 to g, o_2 will not get a node-pointer to reach a local clean zone to get a fresh g. It becomes untenable to restart o_2 in such a situation without accessing root, which we want to avoid (it may well be with o_1 symmetrically). To tackle this issue, we let o_2 fall back to the approach of algorithm 1 and instead of cleaning the DeadBl node out it adds a Dead node containing key k_2 at g and gets out of the system to ensure progress. Therefore, similar to algorithm 1, we make an ADD operation replace a Dead node with a new leaf node, knowing that no REMOVE operation takes step to clean out such a node.

With basics in place, we are ready to describe the pseudo-

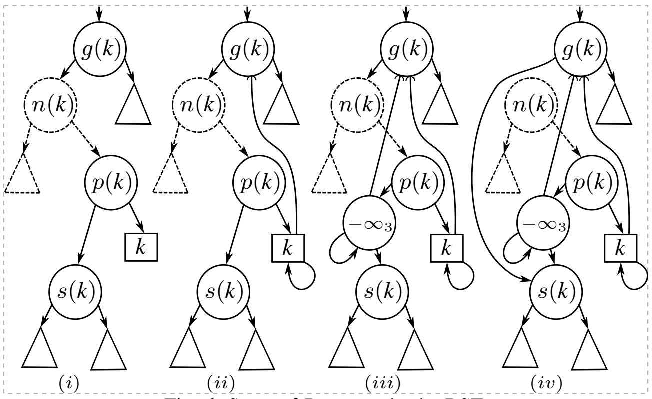


Fig. 6: Steps of REMOVE in the BST.

code of REMOVE and ADD operations of algorithm 3; a CONTAINS operation works absolutely same as that in algorithm 1.

The steps of a REMOVE(k) operation, line 21 to 44, are shown in fig. 6. Let $n(k)$ be the last logically removed internal node in the traversal path obtained by a call of `Search` at line 25, and $g(k)$ be its parent as shown in fig. 6 (i). $n(k)$ coincides with $p(k)$ in case there does not exist a chunk of logically removed nodes above $p(k)$ in the traversal path. Replacing the target node k with a `DeadBl` node containing same key, fig. 6 (ii), logically removes k, line 33. After that, we add a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ as shown in fig. 6 (iii). Finally, update the link $g(k)\rightsquigarrow n(k)$ to connect $s(k)$ to $g(k)$ as shown in fig. 6 (iv). If all these `CAS` executions are successful, we complete the REMOVE operation with cleaning out the `DeadBl` node.

In the stage LREMOVE itself, if the leaf-node at which traversal terminates, say ℓ, does not contain k or is found `Dead` (note that a `DeadBl` node is also `Dead`), REMOVE(k) returns false, line 29. If ℓ is a splice node, it shows that the actual node to remove is pointed by `GetNxt`(ℓ) which is ℓ.rt, see line 9. To make a REMOVE operation *selfish*, we do not perform any `CAS` to help the pending REMOVE. However, we do not have a possibility for a local restart to get a fresh $g(k)$ after the completion of LREMOVE, similar to case (B) above. Therefore, we replace the link $g(k)\rightsquigarrow n(k)$ by a `Dead` node containing k, and return true if `CAS` succeeds, line 39.

In the stage PREMOVE, the first step is to add a splice node between $p(k)$ and $s(k)$, line 34. We use the method `AddSp`, line 15 to 19, to do that. In `AddSp`, if $s(k)$, denoted by sib, is found splice or `DeadBl`, indicated by non-null bck link, we return it as it is, line 17, because both indicate that the child-link of *par* indicated by sib is never updated ever after. If sib is `Dead`, we perform a `CAS`, line 19, to replace it with a `DeadBl` node connecting its bck field to *gPar*, done at line 13, so that a concurrent ADD does not replace it directly.

If the method `AddSp` returns a splice node at line 34, it indicates that a concurrent ADD operation is working selfishly to progress (we describe it later) and we can safely allow the remaining steps of REMOVE(k) to be assimilated in the steps of ADD. Considering that, REMOVE(k) return true, line 35. We call this behaviour of REMOVE(k) its *help-awareness* which is a main component of a *help-optimal implementation*.

On the other hand, if `AddSp` returns a `Dead` or `DeadBl` node, it indicates a scenario of case (B) and we handle it accordingly, see line 37. Finally, if a regular leaf node is returned as sib, we attempt a `CAS` to connect it to $g(k)$ to return true at line 38. If this `CAS` fails, it indicates that $g(k)\rightsquigarrow n(k)$ has changed and therefore we perform a `BckTrck` at line 44 similar to algorithm 2 and reattempt the `CAS` if required, see line 42. Along the lines of algorithm 2, the steps taken to add splice node between $p(k)$ and $s(k)$ are identified by the value of a variable mode set as `INIT` and after that mode is changed to `CLEAN`, line 34.

To add a new node in an external BST, we add a new subtree. We use the following *midpoint rule* to determine the key at the root of the new sub-tree.

Rule 1 (**Midpoint rule**). *Let k be a query key and A be the (partially ordered) set of keys stored in a sub-tree. Let $a_l \leq a\ \forall\ a \in A$ and $a_u \geq a\ \forall\ a \in A$. To add a new node at the root of the sub-tree, assign a key k_p at the root of the new sub-tree such that $k_p=\frac{k+a_u}{2}$ if $k>a_u$ and $k_p=\frac{k+a_l}{2}$ if $k<a_l$.*

The mid-point rule maintains the symmetric order of the BST. Intuitively, rule 1 optimizes the average hight of the BST. We do not delve into an analytical discussion of this rule in the present work. In experiments, we observed that this technique improves the average throughput.

An ADD(k), line 49 to 68, performs a traversal using `Search` to reach a leaf node ℓ. If ℓ is neither `Dead` nor `DeadBl`, we find the regular leaf node using `GetNxt`(ℓ). It calls the `NewNod` method to create a new internal node pointed by nl. We apply rule 1 at line 56. If ℓ is a regular node, it perform as in algorithm 1. However, if ℓ is a splice node, it does not take steps to help the pending REMOVE operation and behaves in a selfish manner to directly update $g(k)\rightsquigarrow n(k)$ to nl using a `CAS`. If `CAS` succeeds, it not only ensures success of ADD(k), but also guarantees the completion of some pending REMOVE operations. If `CAS` to connect nl at line 58 or 59 succeeds, we return true.

On the other hand, if ℓ is found `Dead`, ADD(k) behaves along the lines of algorithm 1, see line 67. And finally, if ℓ is `DeadBl`, to ensure progress, we first fix the sibling of ℓ using the method `AddSp`, line 62, and then add either a new node, line 66, or a new internal node, line 65, at $g(k)$ in a selfish fashion. The call of `AddSp` at line 62 may assimilate the steps of a concurrent pending REMOVE operation, which being help-aware, terminates immediately, as discussed before. Note that, to apply rule 1 here, we use p.k instead of `GetKey`(sib) because the latter may not provide the required bound of the set of keys stored in the sub-tree rooted at sib.

On a `CAS` failure, we perform a `BckTrck` at line 68 to get a fresh set of thread-local variables and reattempt.

B. Correctness and Lock-freedom

Proving that the modify operations maintain a valid external BST requires similar approach as that in algorithm 2. Therefore, without repeating them, we mention that we derive an induction based proof building on the arguments that the sentinel nodes form a valid BST at the initialization and no modify operation invalidates the symmetric order of the BST.

In this algorithm, the linearization points of a successful ADD, REMOVE and CONTAINS operations and an unsuccessful ADD operation are similar to their counterparts in algorithm 2. A CONTAINS or a REMOVE operation returns false also in case the node containing query key is found `Dead`, in addition to the cases already discussed in algorithm 2. The linearization point of such a CONTAINS or REMOVE operation is taken at own invocation point.

Finally, we can prove the lock-freedom of algorithm 3 using arguments which are parallel to those used in algorithm 2. Very evidently, a CONTAINS is wait-free for a finite key universe.

V. HELP-OPTIMALITY: SPECIFICATION

We consider a *shared memory system* U comprising of a finite set of *threads* Λ and a finite set of *shared variables* V. At time t, the *states* of Λ and V are denoted by Λ_t and V_t, respectively. Let Υ be a lock-free data structure formed by variables $v \in V$. Let $\mathcal{O}_\lambda$ be the set of *operations* performed by a $\lambda \in \Lambda$ on Υ. A *step* s of an operation $o \in \mathcal{O}_\lambda$ comprises local computations in λ and at most a single execution of an atomic primitive $a \in \{$`read, write, CAS`$\}$ on a shared variable $v \in V$. A state Υ_t of Υ is formed by variables $v \in V_t$. On execution of a step s, Υ can change from a state Υ_t to another state $\Upsilon_{t'}$. We denote such a state change by $\Delta\Upsilon_{t,t'}$. Let S_o denote the set of steps to complete an operation $o \in \mathcal{O}_\lambda$.

We call $s \in S_o$ an *altruistic step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$ (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of a concurrent operation $o' \in \mathcal{O}_{\lambda'}$ and (c) $\Delta\Upsilon_{t,t'}$ is *not* necessary for completion of o. We call an operation o *selfish* if *no* step $s \in S_o$ is altruistic.

We call $s \in S_o$ a *wasted step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$ (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of o (c) $\Delta\Upsilon_{t,t'} \subseteq \Delta\Upsilon_{t,t''}$ (c) $\Delta\Upsilon_{t,t''}$ has already been applied by a set of steps $\{s'_1, \ldots, s'_n\}$, where $s'_i \in S_{o'_i}$ is a step of a concurrent operation $o'_i \in \mathcal{O}_{\lambda'_i}$. We call $o \in \mathcal{O}_\lambda$ *help-aware* if it performs *no more than one* wasted step.

A lock-free data structure Υ is called *help-optimal* if every operation $o \in \mathcal{O}_\lambda$ for each $\lambda \in \Lambda$ is both selfish and help-aware. In algorithms 1 to 3, we can observe that every operation satisfies the requirements of both selfishness and help-awareness. We skip a rigorous definitional discussion on selfishness, help-awareness, and help-optimality to a future work.

Censor-Hillel et al. [22] defined *help-freedom*, which intuitively implies that an operation does not *altruistically* help a concurrent (slow) operation to guarantee wait-freedom. In contrast to that, in lock-free algorithms, help-optimality not only implies an absence of altruistic helping but also indicates that an operation is aware of intended modification getting applied as a part of a modification by a concurrent operation. Thereby, in a finite execution, the aggregate number of steps is optimized. In this work, we do not delve into a formal comparison between help-optimality and help-freedom.

VI. EXPERIMENTAL EVALUATION

A. Overview

In this section, we present a detailed performance analysis of our implementations against a number of existing concurrent Set implementations both in C/C++ and Java.

We compare the following concurrent linked-lists:

1) **HO-LL:** An implementation of lock-free linked-list of [1], where a CONTAINS does not clean out logically removed nodes, which is an optimization over the original algorithm.
2) **Lazy-LL:** Lock-based linked-list of [19], in which logically removed nodes are ignored during traversal. Operations acquire locks attached to target nodes, then perform updates.
3) **CWT-LL:** Lock-free linked-list described in section III.

We compare the following lock-free binary search trees:

1) **EFRB-BST:** Lock-free external BST of [4], in which both ADD and REMOVE require help to complete at a conflict.
2) **LF-SKIPLIST:** Concurrent skip-list implementation that is part of the java.util.concurrent package [17].
3) **NM-BST:** Lock-free external BST of [7], in which multiple nodes under REMOVE by different threads are cleaned out together similar to algorithm 3. The modify operations restart from root of the BST at each `CAS` failure.
4) **CWT-BST:** Lock-free BST described in section IV.
5) **CWT-Simple-BST:** Simple Lock-free BST of section II.

B. Experimental Set-up

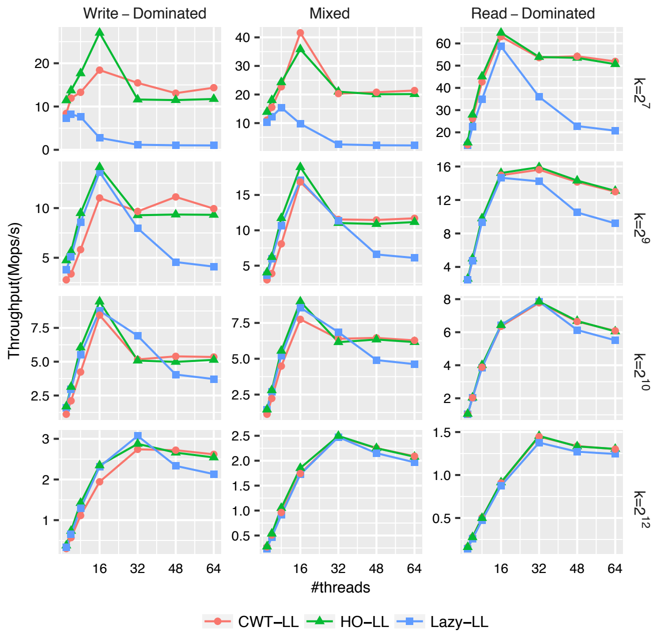
We performed our evaluations on a dual-socket server with a 3.4 GHz Intel (R) Xeon (R) E5-2687W-v2 having 16 physical cores (32 hardware threads), 16 GB of RAM, and runs Ubuntu 13.04 Linux (Kernel version: 3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23, mixed mode). We compiled all Java implementations with `javac` version `1.8.0_60` and used the runtime flags `-d64 -server`. C/C++ implementations were compiled with `g++` version `4.9.2`, `O3` optimization. We used Google's TCMalloc library [23] to reduce dynamic memory allocation overheads.

We compared the performance of various implementations measuring throughput as Million operations per second (Mops/s). We measured the change in heap-size by comparing the memory consumption of the JVM after loading the set with initial elements and after execution of the workload. We considered memory consumption value ascertained after 6 calls of `System.gc()` method, after which it usually converges. Each experiment was run for 5 seconds, then the average over 6 trials was taken under the following parameters:

1) **Workload Distribution:** Similar to [7], we considered three workload distributions: (a) *write-dominated*: 0% search, 50% add and 50% remove. *mixed*: 70% search, 20% add and 10% remove, and (c) *read-dominated*: 90% search, 9% add and 1% remove.
2) **Set Size:**. The maximum set size depends on the range of the keys. We consider the following key ranges for the linked-lists: 2^7, 2^9, 2^{10} and 2^{12}. For the BSTs we consider the ranges: 2^{10}, 2^{14}, 2^{17} and 2^{20}. In each experiment, the set is pre-loaded with approximately half the keys in the key range to ensure consistent results.

C/C++ and Java Implementations: As we pointed out in section I, many concurrent data structures are designed with language specific dependencies and as such offer varying performance in different languages. Additionally, original implementations of the algorithms are available either in Java or C/C++. With this in mind, we implemented our new language-portable lock-free algorithms in both C/C++ and Java.

To ensure a fair comparison, we implemented our C/C++ versions of the algorithms as part of the ASCYLIB library


(a) Java Implementation

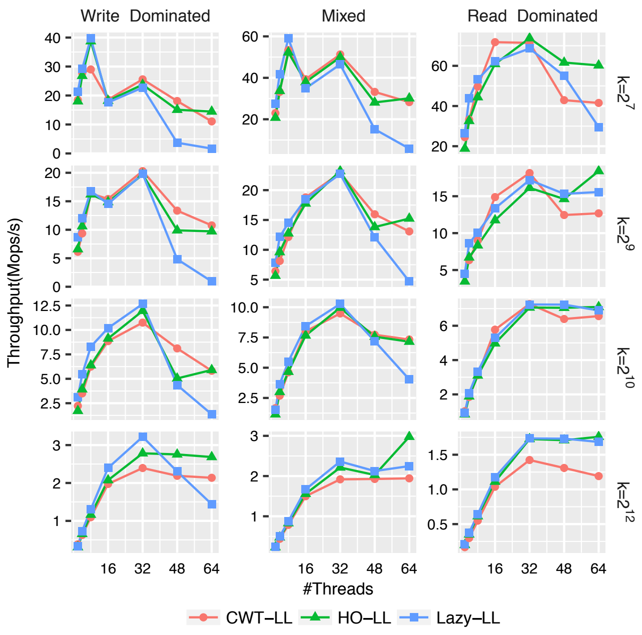

(b) C/C++ Implementation

Fig. 7: Performance comparison of concurrent linked-list algorithms

[11], with SSMEM - a memory allocator with epoch-based garbage collection. We used the same benchmarks which are part thereof. HO-LL in Java employs RTTI. For locking, Lazy-LL uses ReentrantLock in Java and a ticket lock in C/C++.

C. Performance Results and Discussion

Figure 7 depicts the comparative performance of linked-list-based Set algorithms in Java (fig. 7a) and C/C++ (fig. 7b). At low contention i.e. with read-dominated workloads and large key space sizes, the lists scale with increasing thread count. CWT-LL performs on a par with HO-LL in both Java and C/C++. In the high contention cases, mainly write-dominated and small key space sizes, Lazy-LL degrades significantly with increasing thread count. This is mainly due to the increased contention on the locks and cache misses resulting from the lock migrations. Contention increases as the list gets shorter in size with a smaller key space size. At high contention CWT-LL outperforms HO-LL by 5% for Write-Dominated and 3%-6% for Mixed workloads. This can be attributed to the local restart and the ability to clean out multiple nodes in a single step.

In C/C++ we observe similar relative performance, however, the list performance degrades significantly when the cores are saturated with threads (most especially in the write-dominated workload). The effect of oversubscribing the cores with more threads is bigger in Lazy-LL than that in other algorithms as a result of increased lock-contention. Performance degradation due to NUMA effects is also more pronounced in the C/C++ implementation, however, the absolute throughput values are still higher than those for the Java implementation.

Figure 8 shows the comparative performance of considered lock-free BST and skip-list algorithms in Java (8a) and C/C++ (8b). We have not included CWT-Simple-BST here considering its incomparable memory footprint. It is clear that among the Java implementations, CWT-BST offers the best throughput for all key space sizes and workloads. CWT-Simple-BST outperforms EFRB-BST by 10%- 50% and LF-SKIPLIST 20%-100% over Write-Dominated and Mixed workloads. We further observe that all algorithms scale with increasing thread count until it saturates the available cores.

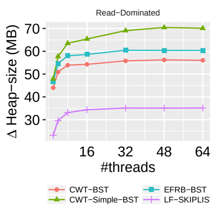
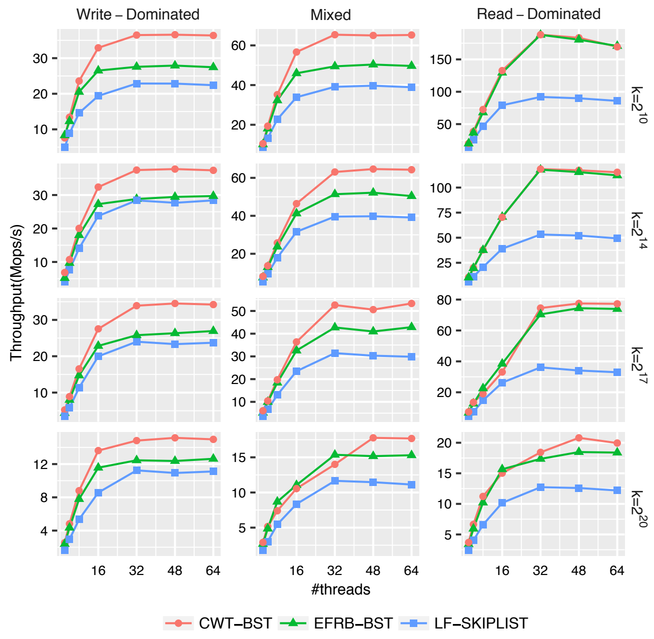


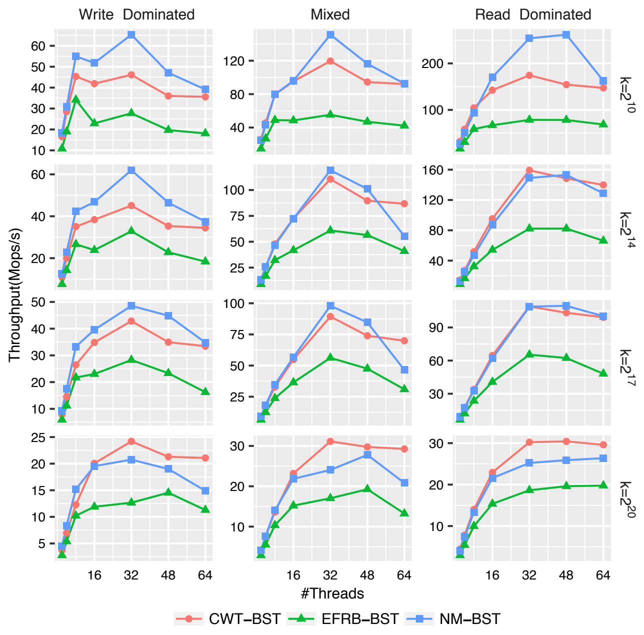
Fig. 9: Heap size change

With regards to C/C++, NM-BST implementation outperforms other implementations in the high contention cases. This can be explained in terms of the advantage of bit-stealing technique over explicit object allocations. Bit masking, unmasking and other bitwise operations in C/C++ are incredibly simple and faster than object creation, however not portable to other high-level programming languages. On the other hand, as we increase the key space size, CWT-BST offers performance similar to NM-BST, especially in Mixed and Read-Dominated workloads, even dominating in the low contention case with key space (2^{20}) by 3%-15%. This can be attributed to a comparative cost of object allocation but lowered cost of reading a pointer without bit unmasking. It can be noted that although EFRB-BST implementation is based on bit-stealing, CWT-BST outperforms it in every considered scenario by 10%-50%.

Memory Management and Garbage Collection: With each REMOVE operation requiring an extra splice node the load on garbage collector definitely increases. However, as illustrated by fig. 9, in a garbage collected environment, CWT-BST experiences no unexpected growth in heap-memory usage. In fact, on this account, it fares better than EFRB-BST. Though the figure presents a case for one workload setting, we observed similar relative memory usage with every workload settings. Nevertheless, We do have to caution that these techniques should not be used without memory reclamation.



(a) Java Implementation



(b) C/C++ Implementation

Fig. 8: Performance comparison of lock-Free BST algorithms

VII. Conclusion and Future Work

In this paper, we introduced the notion of help-optimality in a lock-free algorithm. Intuitively, in a lock-free data structure satisfying help-optimality, at a conflict over modification of a shared variable, we avoid both offer and acceptance of help in form of a step comprising an atomic primitive execution. Help-optimality consists of the notions of selfishness and help-awareness. Selfishness implies optimization of the count of steps of CAS executions by an obstructed operation, whereas help-awareness implies the same for an obstructing operation.

The present work is mostly experimental in nature to demonstrate the utility of the concept of help-optimality in a lock-free linked-list and a BST. In future, we plan to develop formal specifications of the introduced notions.

Following a state-of-the-art implementation of lock-free skip-list in Java library, in this paper, we designed the lock-free data structures to provide a language-portable implementation. We experimentally showed that such an implementation performs on a par with highly optimized implementations in C++ which use the technique of bit-stealing. The Go programming language, which reasonably focuses on concurrency, provides pointers without pointer-arithmetic and does not provide type-inheritance. We believe that with growing popularity of such languages, designing language-portable lock-free data structures will be increasingly significant.

References

[1] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *15th DISC*, 2001, pp. 300–314.

[2] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *14th SPAA*, 2002, pp. 73–82.

[3] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *23rd PODC*, 2004, pp. 50–59.

[4] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *29th PODC*, 2010, pp. 131–140.

[5] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *24th SPAA*, 2012, pp. 161–171.

[6] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *27th IPDPS*, 2013, pp. 1309–1320.

[7] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *19th PPoPP*, 2014, pp. 317–328.

[8] F. Ellen, P. Fatourou, J. Helga, and E. Rupert, "The amortized complexity of non-blocking binary search trees," in *33rd PODC*, 2014, pp. 332–341.

[9] B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient lock-free binary search trees," in *33rd PODC*. ACM, 2014, pp. 322–331.

[10] V. Gramoli, "More than you ever wanted to know about synchronization," *20th PPoPP*, 2015.

[11] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *20th ASPLOS*, 2015, pp. 631–644.

[12] A. Ramachandran and N. Mittal, "A fast lock-free internal binary search tree," in *17th ICDCN*, 2015, pp. 37:1–37:10.

[13] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[14] G. Barnes, "A method for implementing lock-free shared-data structures," in *5th SPAA*, 1993, pp. 261–270.

[15] J. Gibson and V. Gramoli, "Why non-blocking operations should be selfish," in *Distributed Computing*. Springer, 2015, pp. 200–214.

[16] "java.util.concurrent," *https://docs.oracle.com/javase/8/docs/api/*.

[17] D. Lea, "Concurrentskiplistset," *https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html*.

[18] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *14th PODC*, 1995, pp. 214–222.

[19] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. Iii, and N. Shavit, "A Lazy Concurrent List-Based Set Algorithm," in *9th OPODIS*, 2005, pp. 3–16.

[20] Y. Afek, G. Stupp, and D. Touitou, "Long lived adaptive splitter and applications," *Distributed Computing*, vol. 15, no. 2, pp. 67–86, 2002.

[21] H. Attiya and A. Fouren, "Algorithms adapting to point contention," *Journal of the ACM*, vol. 50, no. 4, pp. 444–468, 2003.

[22] K. Censor-Hillel, E. Petrank, and S. Timnat, "Help!" in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. ACM, 2015, pp. 241–250.

[23] S. Ghemawat and P. Menage, "Tcmalloc : Thread-caching malloc," *http://goog-perftools.sourceforge.net/doc/tcmalloc.html*, 2009.