TSLQueue: An Efficient Lock-Free Design for Priority Queues

Adones Rukundo(✉) and Philippas Tsigas

Chalmers University of Technology,
Gothenburg, Sweden
{adones,tsigas}@chalmers.se

Abstract. Priority queues are fundamental abstract data types, often used to manage limited resources in parallel systems. Typical proposed parallel priority queue implementations are based on heaps or skip lists. In recent literature, skip lists have been shown to be the most efficient design choice for implementing priority queues. Though numerous intricate implementations of skip list based queues have been proposed in the literature, their performance is constrained by the high number of global atomic updates per operation and the high memory consumption, which are proportional to the number of sub-lists in the queue.

In this paper, we propose an alternative approach for designing lock-free linearizable priority queues, that significantly improves memory efficiency and throughput performance, by reducing the number of global atomic updates and memory consumption as compared to skip-list based queues. To achieve this, our new design combines two structures; a search tree and a linked list, forming what we call a Tree Search List Queue (*TSLQueue*). The leaves of the tree are linked together to form a linked list of leaves with a head as an access point. Analytically, a skip-list based queue insert or delete operation has at worst case $O(\log n)$ global atomic updates, where n is the size of the queue. While the *TSLQueue* insert or delete operations require only 2 or 3 global atomic updates respectively. When it comes to memory consumption, *TSLQueue* exhibits $O(n)$ memory consumption, compared to $O(n \log n)$ worst case for a skip-list based queue, making the *TSLQueue* more memory efficient than a skip-list based queue of the same size. We experimentally show, that *TSLQueue* significantly outperforms the best previous proposed skip-list based queues, with respect to throughput performance.

Keywords: Shared data-structures · Concurrency · Lock-freedom · Performance scalability · Priority queues · External trees · Skip lists · Linked lists

1 Introduction

A priority queue is an abstract data type that stores a set of items and serves them according to their given priorities (*keys*). There are two typical operations

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 385–401, 2021.
https://doi.org/10.1007/978-3-030-85665-6_24

supported by a priority queue: *Insert()* to insert a new item with a given priority in the priority queue, and *DeleteMin()* to remove a minimum item from the priority queue. Priority queues are of fundamental importance and are essential to designing components of many applications ranging from numerical algorithms, discrete event simulations, and operating systems. Though there is a wide body of literature addressing the design of concurrent priority queue algorithms, the problem of designing linearizable lock-free, scalable, priority queues is still of high interest and importance in the research and application communities. While early efforts have focused mostly on parallelising Heap structures [12], recent priority queues, based on Pugh's skip lists [16], arguably have higher throughput performance [13,18,19].

Skip lists are increasingly popular for designing priority queues due to their performance behaviour, mentioned above. A major reason being that skip lists allow concurrent accesses to different parts of the data structure and are probabilistically balanced. Skip lists achieve probabilistic balance through having a logarithmic number of sub-list layers that route search operations [16]. Although this is good for the search cost, it penalises the *Insert()* and *DeleteMin()* performance due to the high number of atomic instructions required to update multiple nodes belonging to different sub-lists per operation (*global atomic updates*). This also leads to high memory utilisation/consumption to accommodate the sub-lists. Trying to address this problem, a *quiescently consistent*[1] [10] multi-dimensional linked list priority queue [22], was proposed to localise the multiple global atomic updates to a few consecutive nodes in the queue. However, similar to the skip list, the multi-dimensional queue also suffers from a high number of global atomic updates and high memory consumption, which are proportional to the number of queue dimensions. A high number of global atomic updates typically increases the latency of an operation, and can also lead to high contention which limits scalability. Optimisation techniques such as lock-free chunks [2], flat combining [9], elimination [3], batch deleting [13] and back-off [19], and semantic relaxation [1] have been proposed to improve the performance of priority queues. However, these techniques mostly target to reduce the scalability challenges associated with the *DeleteMin()* sequential bottleneck. Although numerous skip list queue designs and optimisation techniques have been proposed in the literature, the underlying skip list design behaviour that generates a high number of atomic updates and memory consumption persists.

In this paper, we propose an alternative approach for designing efficient, lock-free, lineraizable priority queues with a minimal number of global atomic updates per operation. Our design is based on a combination of a binary external search tree [7,15] and an ordered linked list [8,21]. Typically, an external tree is composed of a sentinel node (*root*), internal-nodes and leaf-nodes. We modify the tree to add a link between the leaf-nodes, forming a linked list of leaf-nodes with a sentinel node (*head*). We also combine the internal-node and leaf-node into one physical *node*. Within the tree, the *node* is accessed as an internal-node,

[1] Quiescent consistency semantics allow weaker object behaviour than strong consistency models like linearizability to allow for better performance.

whereas at the list level, the *node* is accessed as a leaf-node. We maintain only two levels in which a *node* can be accessed, that is, tree level and list level. Our combination of a tree and a linked list forms what we refer to as a tree-search-list priority queue (*TSLQueue*). *TSLQueue* is not guaranteed to be balanced due to the underlying binary external search tree structure.

Similar to a balanced skip list, a balanced *TSLQueue* has a search cost of $O(\log n)$. However, *TSLQueue* has a minimal number of global atomic updates. *TSLQueue Insert()* performs one or two global atomic updates on one or two consecutive *nodes*. *TSLQueue DeleteMin()* performs two or three global atomic updates on two or three *nodes*. The tree design requires only one internal-node update for either *Insert()* or *DeleteMin()*, a property that we leverage to achieve low global atomic updates and consequently better performance. Having a single *node* to represent both the tree and the list level, gives *TSLQueue* minimal memory consumption of $O(n)$. Reducing the memory consumption significantly improves cache behaviour and lowers memory latency, especially for larger priority queues as we discuss later in Sect. 5. Optimisation techniques such as batch deleting, frequently used in concurrent data structure designs are limited by memory availability. However, *TSLQueue* can efficiently execute batch deletes due to its low memory consumption.

We experimentally compare our implementation of *TSLQueue* to two state-of-the-art skip list based priority queues, one that is linearizable [13] and one that is quiescently consistent [18]. Overall *TSLQueue* outperforms the two algorithms in all the tested benchmarks, with a throughput performance improvement of up to more than 400% in the case of *DeleteMin()* and up to more than 65% in the case of *Insert()*.

The rest of the paper is organised as follows. In Sect. 2 we discuss the literature related to this work. We present our proposed *TSLQueue* design together with its implementation details in Sect. 3, and prove its correctness in Sect. 4. We experimentally evaluate our implementation in comparison to skip list based priority queues in Sect. 5 and conclude in Sect. 6.

2 Related Work

Concurrent priority queues have been extensively studied in the literature, with most proposed designs based on three paradigms: heaps [5,6,12,14,20], skip lists [13,18,19] and multi-dimensional linked lists [22]. However, empirical results in the literature show that heap based priority queues do not scale past a few numbers of threads. Therefore, we leave out the details of heap priority queues due to space constraints.

Skip lists are search structures based on hierarchically ordered linked lists, with a probabilistic guarantee of being balanced [16]. The basic idea behind skip lists is to keep items in an ordered list, but have each record in the list be part of up to a logarithmic number of sub-lists. These sub-lists play the same role as the routing nodes (internal-nodes) of a binary search tree structure. To search a list of n items, $O(\log n)$ sub-list levels are traversed, and a constant number of items is traversed per sub-list level, making the expected overall complexity of $O(\log n)$.

Maintaining sub-lists penalizes the *Insert()* and *DeleteMin()* performance, due to the high number of global atomic memory updates required to update multiple nodes belonging to different sub-lists. This also leads to high memory consumption to accommodate the sub-lists. In the worst case, the number of global atomic updates of a balanced skip list based queue can go up to $O(\log n)$ for each operation, whereas memory consumption can go up to $O(n \log n)$, where n is the number of items in the queue. Several skip list based priority queues have been proposed in the literature, including; quiescently consistent ones [18,22] and linerizable ones [13,19]. Linerizabilty [11] is the typical expected behaviour of a concurrent data structure. Quiescent consistency [10] is a form of relaxed linearizability that allows weaker data structure behaviour to achieve better performance.

A quiescently consistent multi-dimensional linked list priority queue [22] has been proposed to avoid distant updates by localising the queue operations to a few consecutive nodes. The multi-dimensional queue is composed of nodes that contain multiple links to child nodes arranged by their dimensionality. Nodes are ordered by their coordinate prefixes through which they can be searched. The insertion operation maps a scalar key to a high-dimensional vector, then uniquely locates the insertion point by using the vector as coordinates. Generally, the *Insert()* and *DeleteMin()* operations update pointers in consecutive nodes, but the number of global atomic updates is proportional to the number of dimensions, similar to the skip list. Just like the skip list, multi-dimensional queues also exhibit high memory consumption, proportional to the number of dimensions.

Like other concurrent priority queues, skip list based priority queues have an inherent sequential bottleneck when accessing the minimum item, especially when performing a *DeleteMin()*. Building on [19], a skip list based priority queue with batch deleting has been proposed with the aim of addressing the sequential bottleneck challenge [13]. The algorithm achieves batch deleting, by not physically deleting nodes after performing a logical delete. Instead, the algorithm performs physical deletion in batches for a given number of logically deleted nodes. Each batch deletion is performed by simply moving the queue *head* pointers, so that they point past the logically deleted nodes, thus making them unreachable. Batch deleting achieves high performance by reducing the number of global atomic updates associated with physical deletes. However, this does not reduce the number of global atomic updates associated with the *Insert()*. Also, the batch deleting technique used is limited by the memory latency generated from traversing logically deleted nodes.

3 Algorithm

3.1 Structure Overview

In this section, we give an overview of our *TSLQueue* design structure depicted in Fig. 1. Our *TSLQueue* is a combination of two structures; a binary external

(a) *TSLQueue node* structure

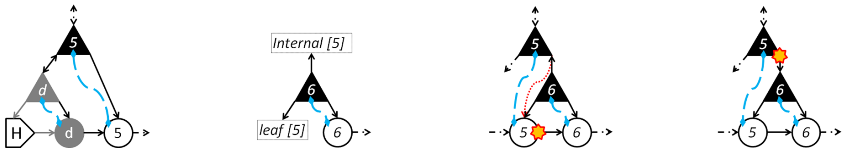
(b) *TSLQueue* with three *nodes* (5, 6 and a *dummy*)

Fig. 1. The triangle represents the *internal*, the circle represents the *leaf*. *internal* and *leaf* physical connection is represented by the blue dashed diamond pointed line. (H) represents the *head* while (R) represents the *root* and (d) for *dummy*. (Color figure online)

search tree and an ordered linked list. The *TSLQueue* is comprised of *nodes* in which queue items are stored and can be accessed through the *root* or the *head*. Each physical *node* has two logical forms; internal-node (*internal*) and leaf-node (*leaf*), and can either be active or deleted. At the tree level, the *node* is accessed as an *internal* to facilitate binary search operations. Whereas at the list level, the *node* is accessed as a *leaf* to facilitate linear search operations. A *node* has four pointers as shown in Fig. 1a: left child, right child, parent and next. A list of *leaves* is created using the next-pointer while the other pointers are used to create the tree. To identify a *leaf*, we reserve one least significant bit (leaf-flag) on each child-pointer, a common method used in the literature [8]. If the leaf-flag is set to true, then the given child is a *leaf*. To simplify linearizability and also to avoid the ABA[2] problem at the *head*, we keep an empty node (*dummy*) between the *head* and the rest of the active *nodes* as shown in Fig. 1b. *DeleteMin()* always deletes the *dummy* and returns the value of the next *leaf*. The *node* whose value has been returned becomes a *dummy*. In other words, a *leaf* preceded by a deleted *leaf* is always a *dummy*. *TSLQueue* supports the two typical priority queue operations; insert an item and delete a minimum item, plus search for an item. Just like for the leaf-flag, we reserve one least significant bit (delete-flag) on the next-pointer. A *node* is logically deleted if the delete-flag is set to true, and physically deleted if it cannot be reached through the *head* or the *root*.

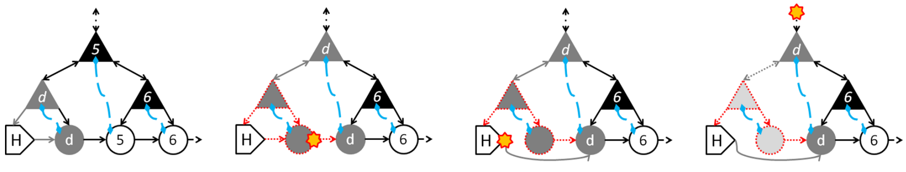
Search: For simplicity, we design two types of tree search operations; *InsertSearch()* and *CleanTree()*. *InsertSearch()* searches for an active *leaf* preceding a given search key, whereas *CleanTree()* searches for an active *dummy* while physically deleting logically deleted nodes from the tree. Both search operations start from the *root*, traversing *internals* until the desired *leaf* is reached. Partial search operations are also supported, as we describe later.

[2] The ABA problem occurs when multiple processors access a shared location without noticing each others' changes.



(a) Initial state (b) Insert node (c) Update I (d) Update II

Fig. 2. An illustration of inserting node (key = 6). The starred areas indicate the atomic operation execution point.



(a) Initial state (b) Update I (c) Update II (d) Update III

Fig. 3. An illustration of *DeleteMin()* that returns minimum node (key = 5). The starred areas indicate the atomic operation execution points.

Insert: Inserting a queue item starts with an *InsertSearch()* operation. *InsertSearch()* locates an active preceding *leaf* (*precLeaf*) to the item key, together with the *precLeaf parent* and succeeding *leaf* (*succLeaf*). Using the search information, a new-node is allocated with its next-pointer pointing to the *succLeaf*, left child-pointer pointing to the *precLeaf* and parent-pointer pointing to the *parent* as shown in Fig. 2b. Both left and right child-pointers are marked as *leaves* since insertion happens at the list level. *Insert()* performs two global atomic updates as illustrated in Fig. 2 in the following order:

I Atomically adds the new-node to the list, by updating the *precLeaf* next-pointer from *succLeaf* to new-node, see Fig. 2c. On the success of this atomic operation, *Insert()* linearizes and the new-node becomes active.

II Atomically adds the new-node to the tree, by updating the given *parent* child-pointer from *precLeaf* to the new-node with the leaf-flag set to false, see Fig. 2d. *Insert()* completes and returns success.

Delete Minimum: For retrieving a minimum queue item, the operation starts from the *head*. A linear search on the list is performed until an active *dummy* is located. *DeleteMin()* performs three atomic global updates as shown in Fig. 3 and in the following order:

I Atomically, logically deletes the *dummy* by setting the next-pointer delete-flag to true. On the success of this atomic operation, *DeleteMin()* linearizes and reads the succeeding *leaf* as the minimum item to be returned. The succeeding *leaf* becomes a *dummy* as shown Fig. 3b.

II Atomically, physically deletes the logically deleted *dummy* from the list, by updating the *head* next-pointer from the deleted *dummy* to the new active *dummy* as shown in Fig. 3c.

III Atomically, physically deletes the logically deleted *dummy* from the tree, by updating the closest active ancestor's left child-pointer to point to the active *dummy*. It is likely that the closest active ancestor is already pointing to the active *dummy* as illustrated in Fig. 3d, in that case, *DeleteMin()* ignores the update. *DeleteMin()* completes and returns the value read at the linearization point (update I). We note that there can be different methods of locating the closest active ancestor. However for simplicity reasons, in this paper, we use the earlier discussed *CleanTree()* to locate the closest active ancestor to the *dummy*, as detailed in Sect. 3.2.

3.2 Implementation

Algorithm 1: TSLQueue

```
Struct node
    key; val;
    *parent; *left; *next; *right; ins; ptrp;
Struct head
    *next;
Struct root
    *child;
Struct seek
    *sucNode; *pNode; *preNode; dup; ptrp;
Function insert(key,val)
    while true do
        seek←InsertSearch(key);
        if seek.dup then
            return dup;
        pNode←seek.pNode;
          nextLeaf←seek.sucNode;
          leaf←seek.preNode; ptrp←seek.ptrp;
        newNode←allocNode(key,val);
        newNode.right←MRKLEAF(newNode);
          newNode.left←MRKLEAF(leaf);
          newNode.next←nextLeaf;
          newNode.parent←pNode;
          newNode.ptrp←ptrp; newNode.ins←1;
        if CAS(leaf.next,nextLeaf,newNode) then
            if ptrp=RIGHT then
                CAS(pNode.right,leaf,newNode);
            else if ptrp=LEFT then
                CAS(pNode.left,leaf,newNode);
            newNode.ins←0; return success;
Function delete()
    hNode←head.next;
    if prHead=hNode then
        dummy←prDummy;
    else
        GarbageCollector(timestamp);
          dummy←prHead←hNode;
    while true do
        nextLeaf←dummy.next;
        if nextLeaf=null then
            return null;
        else
            if DEL(nextLeaf) then
                dummy←nextLeaf; continue;
            xorLeaf←FAXOR(dummy.next,1);
            if !DEL(xorLeaf) then
                value←xorLeaf.val; prDummy←xorLeaf;
                if !randPhysicalDel() then
                    return value;
                if CAS(head.next,hNode,xorLeaf) then
                    CleanTree(xorLeaf);
                    nextLeaf←hNode;
                    while nextLeaf≠xorLeaf do
                        cur←nextLeaf;
                          nextLeaf←nextLeaf.next; FREE(cur);
                return value;
            dummy←xorLeaf;
```

Algorithm 2: Search Functions

```
Macro GORIGHT(pNode)
  ptrp←RIGHT; cNode←pNode.right;

Macro GOLEFT(pNode)
  ptrp←LEFT; cNode←pNode.left;

Macro TRAVERSE()
  if sKey ≤ pNode.key ∧ !DEL(pNode) then
    GOLEFT(pNode);
  else
    GORIGHT(pNode);

Function InsertSearch(sKey)
  pNode←root; cNode←root.child;
  while true do
    if DEL(pNode) then
      GORIGHT(pNode); mNode←pNode;
      while True do
        if DEL(pNode) then
          if !LEAF(cNode) then
            pNode←cNode; GORIGHT(pNode);
              continue;
          else
            pNode←cNode.next; GORIGHT(pNode);
              break;
        else
          if randInsClean() then
            CAS(gNode.left,mNode,pNode);
          TREVERSE(); break;
      continue;
    if !LEAF(cNode) then
      gNode←pNode; pNode←cNode;
      TRAVERSE();
    else
      next←cNode.next;
      if DEL(cNode) then
        pNode←next;
        GORIGHT(pNode);
      else if next ∧ next.ins then
        HelpInsert(next);
        pNode←next; TRAVERSE();
      else if next ∧ next=sKey then
        seek.dup←True; return seek;
      else if ptrp=LEFT ∧ pNode.left!=cNode
       then
        TRAVERSE();
      else if ptrp=RIGHT ∧
       pNode.right!=cNode then
        TRAVERSE();
      else
        seek.preNode←cNode;
        seek.pNode←pNode;
        seek.sucNode←next;
        seek.ptrp←ptrp;
        return seek;

Function CleanTree(dummy)
  pNode←root; cNode←root.child;
  while True do
    if DEL(pNode) then
      GORIGHT(pNode); mNode←pNode;
      while True do
        if DEL(pNode) then
          if !LEAF(cNode) then
            pNode←cNode;
            GORIGHT(pNode); continue;
          else
            next←cNode.next;
            if next.ins then
              HelpInsert(next);
            else if pNode.right=cNode then
              gNode.key←0; goto FINISH;
            GORIGHT(pNode); continue;
        else
          if !DEL(gNode) then
            if CAS(gNode.left,mNode,pNode) then
              GOLEFT(pNode); break;
            pNode=gNode; GOLEFT(pNode);
              break;
          goto FINISH;
    else
      if !LEAF(cNode) then
        if !pNode.key ∨ pNode=dummy then
          pNode.key←0; goto FINISH;
        gNode←pNode; pNode←cNode;
          GOLEFT(pNode); continue;
      else
        next←cNode.next;
        if DEL(cNode) then
          if next.ins then
            HelpInsert(next);
          else if pNode.left=cNode then
            next.key←0; goto FINISH;
          GOLEFT(pNode); continue;
      FINISH: break;
```

In this section, we present the implementation of our *TSLQueue* design. *TSLQueue Insert()* and *DeleteMin()* are presented in Algorithm 1, while *InsertSearch()* and *CleanTree()* are presented in Algorithm 2.

Insert() (Line 1.10) takes two parameters, the key and value of the item to be inserted into the queue. Using the item key as the search key, an insertion point is located by performing the *InsertSearch()* operation (Line 1.12). *InsertSearch()* returns a *precLeaf* (`preNode`) together with its *parent* (`pNode`), *succLeaf* (`sucNode`) and its pointer position (`ptrp`) on the *parent* (left or right). However, if the search key is a duplicate *Insert()* terminates (Line 1.13), otherwise a new-node is allocated (Line 1.16). The new-node is then prepared for insertion using the search information (Line 1.17). *Insert()* occurs at the *leaves* level, and therefore, the left child-pointer of the new-node always points to the *precLeaf* as a *leaf* while the right child-pointer points to the new-node self as a *leaf*. The new-node next-pointer points to the *succLeaf* to maintain a link between the *leaves*. Using a CAS[3] instruction, insert first adds the new-node to the list, by atomically updating the next-pointer of the *precLeaf* from *succLeaf* to new-node (Line 1.18). If the CAS fails, insert retries with another *InsertSearch()* operation. If the CAS succeeds, insert proceeds to add the new-node to the tree, by atomically updating the given *parent* child-pointer from *precLeaf* to the new-node using a CAS instruction (Line 1.20 or 1.22). The CAS adding a new-node to the tree can only fail if another concurrent thread performing a search operation has helped to complete the process (Line 2.35 or 2.62). Therefore, the inserting thread does not need to retry the tree update, but rather continues and sets the new-node insert label to complete and returns success (Line 1.23).

DeleteMin() (Line 1.24) does not take any parameter. A thread trying to retrieve a minimum item, accesses the list through the *head dummy* (Line 1.25) or a *dummy* (Line 1.27) whose value was last returned by the thread (*prevDummy*). If the *dummy* is the last node in the list, the queue is empty and the thread returns the empty state (Line 1.32). Otherwise, if the *dummy* is deleted, the thread hops to the next *dummy* (Line 1.36). The thread linearly hops from one deleted *dummy* to another until an active *dummy* is reached, and tries to logically delete the *dummy* using a fetch-and-xor[4] instruction (Line 1.37). If the *dummy* is already deleted (Line 1.48), the thread hops to the next *dummy* (Line 1.48) in the list and retries. Otherwise, if the thread successfully marks an unmarked *dummy* (Line 1.38), it randomly decides whether to physically delete the logically deleted *dummy* (or *dummies*) or return (Line 1.40). We randomise physical deletes to reduce possible contention that might arise from multiple concurrent threads attempting to perform the physical delete procedure at the same time. To physically delete *dummies* from the queue, the thread starts by updating first the *head* to point to an active *dummy* (Line 1.42). Consequently, a *CleanTree()* is performed to update active ancestors pointing to logically deleted children to point to active children (Line 1.43). By updating the *head* and the ancestors, the thread physically deletes *dummies* (batch physical delete) from

[3] CAS atomically compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location.

[4] Fetch and xor atomically replaces the current value of a memory location with the result of bit-wise XOR of the memory location value, and returns the previous memory location value before the XOR.

the list and the tree respectively. Only the thread that physically deleted a given set of logically deleted *dummies* from the list can free their memory for reclamation (Line 1.45). The thread always returns the value of the *dummy* next to the *dummy* it logically deleted (Line 1.39).

As discussed earlier, a thread inserting a queue item has to perform an *InsertSearch()* operation to get the insertion point of the given item. Using the item key as the search key, *InsertSearch()* starts from the *root* (Line 2.11) and traverses the tree *nodes* (*internals*) until an active *internal* with a *leaf* child is reached (Line 2.29). While searching, if the search key is less than or equal to that of an active *internal*, the left of the *internal* is traversed (Line 2.6), whereas if the search key is higher than that of an active *internal*, the right of the *internal* is traversed (Line 2.8). On the other hand, if the *internal* is deleted (Line 2.13), the search traverses the right of the *internal* until an active *internal* (Line 2.21) or *leaf* child is reached (Line 2.19). If the search reaches an active *internal* preceded by a logically deleted *internal*, the thread randomly decides whether to physically delete the preceding logically deleted *internal* (or *internals*) from the tree (Line 2.22) or not. The thread can then proceed with the traversal at the given active *internal* (Line 2.24). The physical delete is accomplished by updating the left of the last traversed active ancestor (Line 2.27) to point to the active *internal* using a CAS instruction (Line 2.23). This operation facilitates batch physical deleting of logically deleted *nodes* from the tree. Randomising physical deletes for the *InsertSearch()* operation, reduces possible contention between concurrent threads trying to physically delete the same *internal* (or *internals*).

If *InsertSearch()* reaches a deleted *leaf* (Line 2.19 or 2.31), the thread proceeds to the next *leaf* as the *dummy* and performs a partial tree search starting with a right traverse on the *dummy* (Line 2.33). If *InsertSearch()* reaches an active *leaf*, the parent to child edge must be checked for incomplete *Insert()* before the search operation returns the *leaf* as the point for insertion. An incomplete *Insert()* operation on the edge must be helped to complete (Line 2.34), and for that, a partial search is performed starting from the helped *internal* (Line 2.35 to 2.36). Our implementation does not consider duplicate keys, if the search key is equal to an active *node* that is not a *dummy*, duplicate is returned (Line 2.37) and *Insert()* returns. To know if an active *internal* is not a *dummy*, the thread has to traverse until a *leaf* is reached. A thread with a search key equal to the *internal* key always traverses the left of the *internal* (Line 2.6), therefore, if the *internal* is not a *dummy* it will be a *succLeaf* to the search key (Line 2.37).

The *CleanTree()* operation physically deletes logically deleted *internals* from the tree. Only a thread that has physically deleted a set of logically deleted *leaves* from the list (Line 1.42) can perform a *CleanTree()* operation (Line 1.43). This is to make sure that the logically deleted *nodes* are completely physically deleted from the queue (Line 1.43) before their memory can be reclaimed. *CleanTree()* searches for an active *dummy* following the same basic steps as the *InsertSearch()* operation with a few differences. *CleanTree()* always traverses the left of an active *internal* and only traverses the right if the *internal* is deleted. If *CleanTree()* encounters an active *internal* after traversing a logically deleted *internal* or a series of logically deleted *internals* (Line 2.66), the thread must try to physically delete the *internal* (or *internals*) by updating the left of the last traversed active ancestor to point to the active *internal* using a CAS instruction (Line 2.68). The update can fail if another concurrent thread performing a *CleanTree()* or *InsertSearch()* has updated the ancestor. In this case, unlike *InsertSearch()* that proceeds to traverse the *internal*, the thread performing the *CleanTree()* has to retry with a partial search starting from the ancestor (Line 2.70). This facilitates batch physical deleting of logically deleted *nodes* and allows memory for the given *nodes* to be freed for reclamation.

Unlike *InsertSearch()*, if *CleanTree()* reaches an active *leaf*, it terminates. However, if the *leaf* is logically deleted, the parent to child edge must be checked for incomplete *Insert()* before *CleanTree()* terminates (Line 2.64 or 2.83). An incomplete insert on the given edge must be helped to complete, and for that, a partial search retry is performed starting from the previous *internal*. Since logical deleting (Line 1.37) does not check for incomplete inserts, there can be deleted *leaves* pending to be added to the tree. Completing insert updates on edges leading to a deleted *leaf* helps search traverse all possible deleted *internals* leading to a *dummy*. Helping inserts during the *CleanTree()* operation is more efficient than blocking *DeleteMin()* operations from marking *nodes* with pending insert updates on their edges.

The information used to help an incomplete insert is stored in the *node* by the inserting thread that allocated the *node* (Line 1.17).

3.3 Memory Management

We manage memory allocation and reclamation using, but not limited to, a generic epoch-based garbage collection library (ssmem) from the ASCYLIB framework [4]. The freed memory is not reclaimed until when it is certain that no other threads can access that memory. Using timestamps, each thread holds a garbage collector version number (*gc-version*) that it timestamps every time it performs a fresh access to the *TSLQueue* list through the *head* (Line 1.29). Accessing the list through the *head* means that the thread cannot access previously freed *nodes*. The garbage collector will only reclaim the memory of *nodes* that were freed before all the threads performed a fresh access to the list through the *head*. Note that a thread that accesses the list using a previous-*dummy* (Line 1.27) does not update its *gc-version*, implying that, *nodes* accessed through the previous-*dummy* will still be un-reclaimed even if they are freed, and thus can route the thread to an active *dummy*.

4 Correctness

In this section, we prove correctness and progress guarantees for our proposed *TSLQueue*. Line numbers in this section refer to the algorithmic functions presented in Sect. 3. The *TSLQueue* is composed of *nodes* that can be accessed through the *head* or *root*. Queue items are stored within the *nodes*, each *node* having an item priority (*key*) k, where $0 < k$.

Linearizability [11] is widely accepted as the strongest correctness condition of concurrent data structures. Informally, linearizability states that in every execution of the data structure implementation, each supporting operation appears to take effect instantaneously at some point (*linearization point*) between the operation's invocation and response. *TSLQueue Insert()* linearizes on the success of the CAS operation that adds the new-node to the list, by updating the next-pointer of the preceding *leaf* to point to the new-node (Line 1.18). The new-node becomes active (visible to other threads) after the *Insert()* has been linearized. If the queue is not empty, *TSLQueue DeleteMin()* linearizes on the success of the fetch-and-xor operation that logically deletes an active *dummy* by setting the delete-flag of the active *dummy* from false to true (Line 1.37). If the queue is empty, *TSLQueue DeleteMin()* linearizes on the reading of the NULL next-pointer of the *dummy* (Line 1.32).

The proofs of the following Lemmas and Theorems have been omitted because of space constraints. They can be found in the extended version of the paper.

Lemma 1. *An active* node *key is the maximum key of its left-descendant(s) and the minimum key of its right-descendant(s). Also implying, that an active* leaf *key is the maximum key of all its preceding active* leaves *and the minimum of all its succeeding active* leaves *in the list.*

Lemma 2. *An active* node *can only be logically deleted once and after all its left-descendants have been deleted. Also implying, that an active* leaf *can only be logically deleted once and after all its preceding* leaves *have been logically deleted. Further implying that* DeleteMin() *cannot linearize on a logically deleted* node.

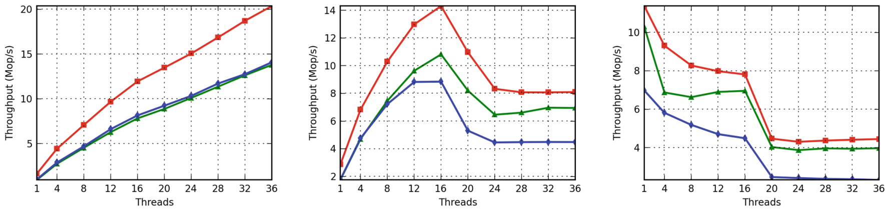
Lemma 3. *An inserted* node *is always pointed to by an active* node *next-pointer. Also implying, that* Insert() *cannot linearize on a logically deleted* node.

Theorem 1. TSLQueue *is a linearizable priority queue.*

Theorem 2. TSLQueue *is lock-free.*

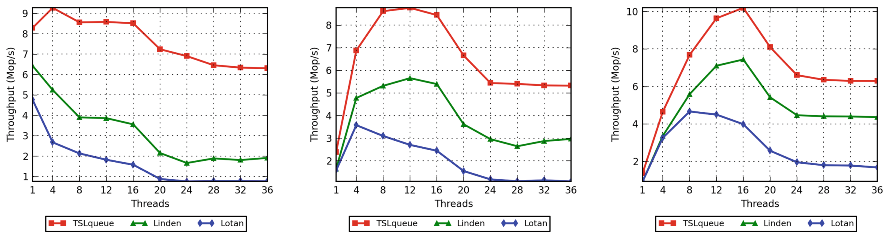
5 Evaluation

To evaluate our queue design, we compare it with two state-of-the-art queue designs based on the skip list; *Lotan* [18] and *Linden* [13]. As we mentioned earlier, skip list based concurrent priority queues arguably have better throughput performance [13,18,19] compared to other previously proposed designs. These two algorithms are performance wise the best representatives of the skip list based queues designs. *Lotan* is a quiescently consistent lock-free adaptation of [18] and maintains most of the skip list operation routines. *Linden* on the other hand customises the skip list to optimise the queue *DeleteMin()* operation through batch deleting. *Linden* is an adaptation of [19], and is accepted by the community, to be one of the fastest skip list based priority queue in the literature.



(a) 100% *Insert()* 0% *DeleteMin()* (b) 75% *Insert()* 25% *DeleteMin()* (c) 50% *Insert()* 50% *DeleteMin()*

Fig. 4. Throughput results for intra-socket (1 to 18 threads) and inter-socket (20 to 36 threads) with an initial queue size of $12 * 10^3$ items.



(a) 100% *DeleteMin()* 0% *Insert()* (b) 75% *DeleteMin()* 25% *Insert()* (c) 50% *DeleteMin()* 50% *Insert()*

Fig. 5. Throughput results for intra-socket (1 to 18 threads) and inter-socket (20 to 36 threads) with an initial queue size of $12 * 10^6$ items.

Our benchmark methodology is a variation of a commonly used synthetic benchmark [4], in which Threads randomly choose whether to perform an *Insert()* or a *DeleteMin()*. The priorities of inserted items are chosen uniformly at random, attempting to capture a common priority queue access pattern. To maintain fairness, the algorithms are run using the same framework and the same memory management scheme [4]. Both *Lotan* and *Linden* use the same method to determine node height based on the distribution of the skip list. To demonstrate the effect of the queue size on the throughput performance, we consider first a queue of small size with $12 * 10^3$ items Fig. 4, and one of a larger size with $12 * 10^6$ items Fig. 5. The queue size is an important evaluation parameter that helps us to also evaluate the memory latency effects for the three algorithms. All three algorithms do not consider duplicates, in their original designs, an insert with a duplicate completes without changing the state of the queue. To avoid duplicates that could skew our results, we use a key range of 2^{30} which is big enough to accommodate the queue sizes experimented with.

We conduct our experiments on a dual-processor machine with two Intel Xeon E5-2695 v4 @ 2.10 GHz. Each processor has 18 physical cores with three cache levels; 32 KB L1 and 256 KB L2 private to each core and 46 MB L3 shared among the 18 cores. Threads were pinned one per physical core filling one socket at a time. Throughput is measured as an average of the number of million operations performed per second out of five runs per benchmark. We observed similar trends on two other hardware platforms; a single processor Intel Xeon Phi CPU 7290 @ 1.50 GHz with 72 cores and dual-processor Intel Xeon CPU E5-2687W v2 @ 3.40 GHz with 16 cores; results are shown in the extended version of this paper due to space constraints.

5.1 Results

First, we evaluate the performance of the *Insert()* operation running without any concurrent *DeleteMin()* operation, the results are presented in Fig. 4a. We observe that *Linden* and *Lotan* have similar *Insert()* throughput performance due to their similar *Insert()* design that is based on the skip list structure. The *Insert()* operation can achieve high parallelism through concurrent distributed accesses of different parts of the queue. However, for *Linden* and *Lotan*, the *Insert()* performance is limited by the high number of global atomic updates, proportional to the number of list levels (*node* height), that the *Insert()* performs to several distant *nodes*. Concurrent threads inserting *nodes* at different points within the queue can still contend while updating the different sub-list level *nodes*, shared by the given inserted *nodes*. Unlike *Linden* and *Lotan*, the *TSLQueue* scales better by leveraging on the lower number of required global atomic updates. *TSLQueue* updates only one or two consecutive *nodes* for each *Insert()*. *TSLQueue* supports single *node* update by storing both the right-child pointer and the next-pointer in the same physical node. When inserting a right child to a *node*, the two atomic updates will operate on the same physical *node*. Having one tree update per insert operation reduces the possible contention between concurrent threads inserting *nodes* at different points within the queue list. For this part of the valuation, *TSLQueue* achieves from 40% to more than 65% better throughput performance compared to both *Lotan* and *Linden*.

Then we evaluate the performance of the *DeleteMin()* operation running without any concurrent *Insert()*, the results are presented in Fig. 5a. The three algorithms use a similar marking method to logically delete a *node*. For *Linden* and *TSLQueue*, a single *node* can be marked at a time, turning the logical delete into a sequential bottleneck. *Lotan* is quiescently consistent and it is possible for more than one *node* to be marked at a time. However, *TSLQueue* and *Linden* batch physical deletes to reduce contention at *nodes* that have to be updated for each physical delete, especially the *head*. *Linden* batch performance is limited by the fact that threads have to linearly transverse all logically deleted *nodes* to reach an active *node*. *TSLQueue* on the other hand uses a randomised approach to avoid contention, and partial linear search to reduce operation latency. *TSLQueue* combines the advantages of partial linear search, batched physical deletes and reduced number of atomic updates per physical delete, to achieve, for this part of the evaluation, from 25% to more than 400% throughput performance compared to both *Linden* and *Lotan* as observed in Fig. 5a. For the three algorithms, *DeleteMin()* scalability is generally limited by the sequential bottleneck at the queue *head* and the minimum queue item.

Lastly, we evaluate the algorithms on workloads that include concurrent *Insert()* and *DeleteMin()* operations. In Fig. 4c and 5c we observe a significant drop in throughput under inter-socket executions for all three algorithms. This drop is attributed to the expensive communication between sockets. *TSLQueue* can efficiently execute batch operations and overall keep a low number of global atomic updates reducing inter-socket communication, thus the observed better performance. Unlike *Lotan* and *Linden*, *TSLQueue* can perform partial searches further reducing the inter-socket communication, especially for the larger queue size as observed in Fig. 5. In Fig. 4c, we observe limited or no scalability as the number of threads increase. This is because for smaller queue sizes, there are few items on which threads can spread their operations leading to contention. However, *TSLQueue* still has better throughput due to the same structural advantages discussed above. We also observe that *TSLQueue* has a significant performance advantage over *Linden* and *Lotan* for the larger queue size compared to the smaller one. Apart from the above structural advantages, this can also be attributed to low memory usage which reduces memory latency.

6 Conclusion

In this paper, we have introduced a new design approach for designing efficient priority queues. We have demonstrated the design with a linearizable lock-free priority queue implementation. Our implementation has outperformed the previously proposed state-of-the-art skip list based priority queues. In the case of *DeleteMin()* we have achieved a performance improvement of up to more than 400% and up to more than 65% in the case of *Insert()*. Though numerous optimisation techniques such as flat combining, elimination and back-off can be applied to further enhance the performance of *TSLQueue*, they are beyond the scope of this paper and are considered for future research.

Acknowledgements and Data Availability Statement. The data sets and code generated and/or analysed during the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.14748420 [17]. This work has been supported by SIDA/Bright Project (317) under the Makerere-Sweden bilateral research programme 2015–2020, Mbarara University of Science and Technology and Swedish Research Council (Vetenskapsrådet) Under Contract No.: 2016-05360.

References

1. Alistarh, D., Kopinsky, J., Li, J., Shavit, N.: The SprayList: a scalable relaxed priority queue. ACM SIGPLAN Not. **50**(8), 11–20 (2015). https://doi.org/10.1145/2858788.2688523
2. Braginsky, A., Cohen, N., Petrank, E.: CBPQ: high performance lock-free priority queue. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 460–474. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_34
3. Calciu, I., Mendes, H., Herlihy, M.: The adaptive priority queue with elimination and combining. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 406–420. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45174-8_28

4. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: the secret to scaling concurrent search data structures. SIGARCH Comput. Archit. News **43**(1), 631–644 (2015). https://doi.org/10.1145/2786763.2694359
5. Deo, N., Prasad, S.: Parallel heap: an optimal parallel priority queue. J. Supercomput. **6**(1), 87–98 (1992). https://doi.org/10.1007/BF00128644
6. Dragicevic, K., Bauer, D.: Optimization techniques for concurrent STM-based implementations: a concurrent binary heap as a case study. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–8 (2009). https://doi.org/10.1109/IPDPS.2009.5161153
7. Ellen, F., Fatourou, P., Helga, J., Ruppert, E.: The amortized complexity of non-blocking binary search trees. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC 2014, New York, NY, USA, pp. 332–340. Association for Computing Machinery (2014). https://doi.org/10.1145/2611462.2611486
8. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45414-4_21
9. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, New York, NY, USA, pp. 355–364. Association for Computing Machinery (2010). https://doi.org/10.1145/1810479.1810540
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
12. Hunt, G.C., Michael, M.M., Parthasarathy, S., Scott, M.L.: An efficient algorithm for concurrent priority queue heaps. Inf. Process. Lett. **60**(3), 151–157 (1996). https://doi.org/10.1016/S0020-0190(96)00148-2
13. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) OPODIS 2013. LNCS, vol. 8304, pp. 206–220. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03850-6_15
14. Liu, Y., Spear, M.: Mounds: array-based concurrent priority queues. In: Proceedings of the 2012 41st International Conference on Parallel Processing, ICPP 2012, USA, pp. 1–10. IEEE Computer Society (2012). https://doi.org/10.1109/ICPP.2012.42
15. Natarajan, A., Ramachandran, A., Mittal, N.: FEAST: a lightweight lock-free concurrent binary search tree. ACM Trans. Parallel Comput. **7**(2), 1–64 (2020). https://doi.org/10.1145/3391438
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990). https://doi.org/10.1145/78973.78977
17. Rukundo, A., Tsigas, P.: Artifact and instructions to generate experimental results for the euro-par 2021 paper: Tslqueue: An efficient lock-free design for priority queues, August 2021. https://doi.org/10.6084/m9.figshare.14748420
18. Shavit, N., Lotan, I.: Skiplist-based concurrent priority queues. In: Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, pp. 263–268 (2000). https://doi.org/10.1109/IPDPS.2000.845994

19. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. **65**(5), 609–627 (2005). https://doi.org/10.1016/j.jpdc.2004.12.005
20. Tamir, O., Morrison, A., Rinetzky, N.: A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In: 19th International Conference on Principles of Distributed Systems (OPODIS 2015), volume 46 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, pp. 1–16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.OPODIS.2015.15
21. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, New York, NY, USA, pp. 214–222. ACM (1995). https://doi.org/10.1145/224964.224988
22. Zhang, D., Dechev, D.: A lock-free priority queue design based on multi-dimensional linked lists. IEEE Trans. Parallel Distrib. Syst. **27**(3), 613–626 (2016). https://doi.org/10.1109/TPDS.2015.2419651