A C++ Implementation of a Lock-Free Priority Queue Based on Multi-Dimensional Linked List

Steven Carroll
University of Central Florida

A C++ Implementation of a Lock-Free Priority Queue Based on Multi-Dimensional Linked List

Alexander Goponenko
Department of Computer Science
University of Central Florida
Orlando, FL 32826
agoponenko@Knights.ucf.edu

Steven Carroll
Department of Computer Science
University of Central Florida
Orlando, FL 32826
sdc@Knights.ucf.edu

***Abstract*—This paper is the third and final publication in a series dedicated to a reimplementation of the priority queue based on multi-dimensional linked lists (Zhang and Dechev [1]). This variant of priority queue guarantees $O(\log N)$ worst-case time complexity where N is the size of the key universe. It also provides improved performance over the state of the art approaches under high concurrency because each insertion modifies at most two consecutive nodes, allowing concurrent insertions to be executed with minimal interference. The current publication continues discussion of the lock-free implementation of the priority queue. The performance of the lock-free implementation is compared to the performance of the MRLock-based implementation. The correctness of the lock-free implementation is also discussed and corrections required to maintain quiescent consistency in case of ongoing purge and insert operation are introduced. Alternative implementations are also discussed.**

***Index Terms*—priority queue, multi-dimensional list**

I. Introduction

A priority queue is a fundamental data structure that comprises a set of key-value pairs where keys indicate priorities (by convention, a smaller key indicates higher priority). A typical priority queue implements only two operations: insert, which adds an item with its associated priority to the queue, and DeleteMin, which removes the highest priority item from the queue. This data structure is employed abundantly in algorithms everywhere from high-level applications to low-level system kernels. Its efficient implementation in multithreaded environments is critical for modern and future multi-core systems. [1] introduced a lock-free priority queue based on multi-dimensional linked lists, with worst-case time $O(\log N)$ for a key universe of size N. This paper is the third and final in a series of publications about a reimplementation of the priority queue from [1] which is based on the multi-dimensional list. The current publication continues discussion of the lock-free priority queue, initially described in [2]. The performance of the lock-free priority queue is compared to the performance of the lock-based priority queue [3] that uses MRLock [4]. A literature survey and comparison of the MDList-based priority queue with alternative implementations is also presented (in Section VI).

II. Multi-dimensional lists

As described in [1], a multi-dimensional (linked) list organizes data in multiple dimensions in a way that facilitates search and insertion. Upon insertion, a scalar key is recalculated into an array of indexes that may be considered multi-dimensional coordinates. This array is used, starting at the lowest dimension, to find pivot points into higher dimensions during the search for the correct insertion point. In contrast to a one-dimensional linked list, many intermediate values can be, and often are, effectively skipped during an insertion. The same is true of the search operation.

Algorithm 1: MDList structures

```
class MDList {
    const int D;
    const int N;
    Node* head;
}

struct Node {
    int key, k[D];
    void* val;
    Node* child[D];
}
```

More formally, A D-dimensional list is a rooted tree in which each node is implicitly assigned a dimension from 0 to $D-1$. The root node's dimension is 0. A node of dimension d has no more than $D-d$ children, and each of its children is assigned a unique dimension in the range from d to $D-1$. The order among nodes is lexicographically based on keys. A dimension d node should share a coordinate prefix of length exactly d with its parent [1].

Each insertion or deletion operation on an MDList requires updating at most two consecutive nodes in the data structure, which makes it suitable for concurrent accesses. Furthermore, the worst-case time of the operations is $O(\log N)$, where N is the size of key universe.

Algorithms 1, 2, and 3 are reproduced from [1]. The priority queue is represented by a class called MDList which has a head field, a key universe size field, N, and a dimension field, D.

As mentioned in [1], the insert algorithm requires only a simple modification to the search algorithm, where a *prev* pointer is maintained for linking when the proper position is found.

In this work we will consider only integer values and keys.

Algorithm 2: Mapping from integer to vector

```
vector<int> keyToCoord(int key) {
    int basis = ceil(pow(N, 1.0 / D));
    int quotient = key;
    vector<int> k;
    k.resize(D);
    for (int i = D − 1; i ≥ 0; i−−) {
        k[i] = quotient % basis;
        quotient = quotient / basis;
    }
    return k;
}
```

Algorithm 3: Search for a Node with Coordinates

```
Node∗ searchNode(vector<int> k) {
    Node ∗cur = head;
    int d = 0;
    while (d < D) {
        while (cur ≠ NULL && k[d] > cur−>k[d])
          cur = cur−>child[d];
        if (cur == NULL || k[d] < cur−>k[d])
          return NULL;
        d++;
    }
    return cur;
}
```

As in [1], only unique keys will be considered. Converting the key into an appropriate D-dimensional vector is accomplished by Algorithm 2.

III. Lock-free implementation

Algorithm 4 shows the structure of the priority queue's node and other data structures. In addition to regular fields key, k, $child$, and val, the node contains field $adesc$, which holds a reference to a descriptor [5] if the node is just inserted and may not have adopted children yet (details are described later). The descriptor (AdoptDesc) contains a reference to the node from which the children must be adopted and a range of dimensions for which the adoption is pending. HeadNode has an additional field – ver, which holds the current version of the head. The version of the head is incremented during the purge operation that is described later.

Algorithm 4: Data Structures for Lock-Free Priority Queue

```
struct AdoptDesc{
  Node∗ curr;
  int dp, dc;
};

struct Node{
  atomic<uintptr_t> val;
  int key;
  int k[D];
  atomic<uintptr_t> child[D];
  atomic<AdoptDesc∗> adesc;
};

struct HeadNode: Node{
  int ver;
};

struct Stack{
  Node∗ node[D];
  HeadNode∗ head;
};
```

Algorithm 5: Marking Scripts

```
#define SetMark(p,m) ((p)|(m))
#define ClearMark(p,m) ((p)&~(uintptr_t)(m))
#define IsMarked(p,m) ((p)&(uintptr_t)(m))
#define F_ADP 0x1U
#define F_PRG 0x2U
#define F_DEL 0x1U
#define F_ALL 0x3U
#define Clear(p) ClearMark(p, F_ALL)
```

Algorithm 6: Priority Queue

```
class PriorityQueue {
  int N;
  int R
  atomic_bool notPurging{true};
  atomic<int> nMarkedNodes{0};
  atomic<uintptr_t> head;
  atomic<Stack∗> stack
  HeadNode firstHeadNode;
  Stack firstStack;

  PriorityQueue(int N, int R): N(N), R(R){
    firstHeadNode.val = F_DEL;
    firstHeadNode.adesc = NULL;
    firstHeadNode.key = 0;
    setCoords(&firstHeadNode, 0);
    firstHeadNode.ver = 1;
    for (int i=0; i<D; i++)
        firstHeadNode.child[i].store(NIL);
    head.store((uintptr_t) (&firstHeadNode));
    firstStack.head = &firstHeadNode;
    for (int i=0; i<D; i++)
        firstStack.node[i] = &firstHeadNode;
    stack.store(&firstStack);
  }

  void setCoords(Node∗ n, int key) {
    int basis = ceil(pow(N, 1.0/D));
    int quotient = key;
    int∗ k = n−>k;
    for (int i = D − 1; i ≥ 0; i−−) {
      k[i] = quotient % basis;
      quotient = quotient / basis;
    }
  }
};
```

The field val of the node has two purposes. Normally, it contains a reference to the node's value. However, if the node has been deleted, val is reused to hold references needed to maintain the queue after purge. The deleted node is marked using the so-called "bit stealing" technique – the last bit of val is set to 1 (in this case we will say that the flag F_DEL is set). The same technique is used to mark invalid references in the $child$ array. If the reference is invalid because it was adopted, it is marked with F_ADP; if it is invalid because of a purge, it is marked with F_PRG. Macros used to mark and unmark references are shown in Algorithm 5.

The structure of the priority queue itself is demonstrated in Algorithm 6. N contains the limit on the value of keys (the size of the key universe); R holds the number of DeleteMin operations between purges; $notPurging$ is a flag needed to make sure that only one purge operation is ongoing; $nMarkedNodes$ is the count of deleted nodes after the last purge; $head$ is the reference to the current head node;

stack contains the pointer to the current deletion stack. The constructor initializes a first head and a first stack of the queue. The head of the queue is a sentinel node, which is marked as deleted and has $key = 0$. The values of the keys of nodes in the queue must be between 0 and N. setCoords perform mapping from key to array k.

In order to attain lock-free and efficient functioning of the priority queue, the operations DeleteMin and insert must be well coordinated. The nodes are deleted from the queue logically by setting the flag F_DEL. The deletion stack is used to store the location of the last logically deleted node to make physical deletion of nodes more efficient. A new node could be inserted in-between logically deleted nodes in a location that is not accessible from the current stack. Therefore, the insert operation may need to rewind the stack to make the newly inserted node accessible.

Insertion of a node is performed in two steps. In the first step the node is spliced into the list using a compare-and-swap (CAS) atomic synchronization primitive in a way similar to that used in a lock-free linked list [6]. In the second step, the node adopts some of the children of the node that occupied its place if the dimension of the replaced node has been changed as a result of the insertion. The need for the second step is announced by descriptor object. Other threads help the adoption if they traverse a node with a descriptor that has been set.

When the number of logically deleted nodes agglomerates above a threshold determined by the variable R, a purge operation is performed to ensure efficient execution and to enable memory reclamation. The purge operation may need to update the deletion stack and also must ensure that all non-deleted nodes remain accessible from the stack. The details of the operations are discussed next.

A. Details of insert operation

Algorithm 7 presents the operation of inserting an item into the priority queue. At the beginning of the operation a new node and a new stack are created. Memory allocation for new elements is dedicated to an object that implements a Handler interface, which is shown in Algorithm 8. The new stack gets recalculated throughout the operation in case the deletion stack needs to be rewound.

LocatePlace traverses the MDList starting from the head and determines the target position for the insertion, i.e. the immediate parent $pred$ for the new node, dimension dp, at which the new node will become the child of $pred$, the node $curr$ that is currently occupying the new node's slot, and dimension dc, at which $curr$ will become the child of the new node. Nodes $pred$ and $curr$ are the only two nodes that are updated by an insertion. During traversal, finishInserting (Algorithm 9) is called for each inspected node, to complete possible ongoing adoption of children.

The CAS operation on line 21 splices the new node into the list. It can fail if the desired location has been updated by a concurrent insertion or because the location was marked invalid by a purge or a child adoption process. In such cases the

Algorithm 7: Inserting a Node into MDList

```
1  bool insert(int key, uintptr_t val, Handler* h){
2    Stack *s = h->newStack();
3    Node *n = h->newNode();
4    n->key = key;
5    n->val = val;
6    setCoords(n, key);
7    for (int i=0; i<D; i++) n->child[i].store(NIL);
8    while (true) {
9      Node* pred = NULL;
10     int dp = 0, dc = 0;
11     s->head = (HeadNode*) (head.load());
12     Node* curr = s->head;
13     LocatePlace(h, dp, dc, pred, curr, n, s);
14     if (dc == D) {
15       // this key is already in the queue
16       return false;
17     }
18     finishInserting(curr, dp, dc);
19     FillNewNode(h, n, dp, dc, curr);
20     uintptr_t temp = (uintptr_t) curr;
21     if (pred->child[dp].compare_exchange_strong(temp, (
           uintptr_t) n)) {
22       finishInserting(n, dp, dc);
23       RewindStack(s, n, pred, dp);
24       return true;
25     }
26   }
27 }

29 inline void LocatePlace(Handler* h, int &dp, int &dc,
30                         Node *&pred, Node *&curr,
31                         Node *n, Stack *s) {
32   while (dc < D) {
33     while (curr != NULL && n->k[dc] > curr->k[dc]) {
34       pred = curr;
35       dp = dc;
36       finishInserting(curr, dc, dc);
37       curr = (Node*) (Clear(curr->child[dc].load()));
38     }
39     if (curr == NULL || n->k[dc] < curr->k[dc]) {
40       break;
41     }
42     s->node[dc] = curr;
43     dc++;
44   }
45 }

47 inline void FillNewNode(Handler* h, int dp, int dc,
48                         Node* n, Node* curr) {
49   if(dp < dc) {
50     AdoptDesc* desc = h->newDesc();
51     desc->curr = curr;
52     desc->dc = dc;
53     desc->dp = dp;
54     n->adesc.store(desc);
55   } else {
56     n->adesc.store(NULL);
57   }
58   for (int i = 0; i < dp; i++) n->child[i] = F_ADP;
59   for (int i = dp; i < D; i++) n->child[i] = NIL;
60   n->child[dc] = (uintptr_t) curr;
61 }
```

Algorithm 8: Priority Queue Handler

```
class Handler{
  Node* newNode();
  AdoptDesc* newDesc();
  Stack* newStack();
  HeadNode* newHeadNode();
};
```

Algorithm 9: Finish Inserting

```
void finishInserting(Node *n, int dp, int dc){
  if (n == NULL) return;
  AdoptDesc* ad = n->adesc;
  if (ad == NULL || dc < ad->dp || dp > ad->dc) return;
  uintptr_t child;
  Node* curr = ad->curr;
  for (int i = ad->dp; i < ad->dc; i++) {
    child = Clear(curr->child[i].fetch_or(F_ADP));
    uintptr_t temp = NIL;
    n->child[i].compare_exchange_strong(temp, child);
  }
  n->adesc = NULL;
}
```

Algorithm 10: Deleting Minimal Node

```
1  uintptr_t DeleteMin(Handler* h){
2    Stack* sOld = stack.load();
3    Stack* s = h->newStack();
4    *s = *sOld;
5    int d = D-1;
6    while (true) {
7      Node* last = s->node[d];
8      finishInserting(last, d, d);
9      Node* child = (Node*) (Clear(last->child[d].load()))
         ;
10     if (child == NULL) {
11       if (d == 0) return NIL;
12       d--;
13       continue;
14     }
15     uintptr_t val = child->val;
16     if (IsMarked(val, F_DEL)) {
17       if (Clear(val) == NIL) {
18         for (int i = d; i < D; i++)
19             s->node[i] = child;
20       } else {
21         s->head = (HeadNode*) (Clear(val));
22         for (int i = 0; i < D; i++)
23             s->node[i] = s->head;
24       }
25       d = D-1;
26     } else {
27       if (child->val.compare_exchange_strong(val, F_DEL)
           ) {
28         for (int i = d; i < D; i++)
29             s->node[i] = child;
30         stack.compare_exchange_strong(sOld, s);
31         int marked = nMarkedNodes.fetch_add(1);
32         if (marked > R)
33             purge(s->head, s->node[D-1], h);
34         return val;
35       }
36     }
37   }
38 }
```

loop beginning at line 8 restarts. Otherwise, the child adoption is completed (finishInserting at line 22) and the deletion stack is rewound if needed (RewindStack at line 23).

B. Details of *DeleteMin* operation

Algorithm 10 demonstrates extraction of the item with highest priority, i.e. deletion of the node with the smallest key. The operation searches for a node that has not been logically deleted, starting from the last entry of the deletion stack. A copy of the stack is maintained following the search in order to update the queue's stack at the end of the operation. When a non-logically-deleted node is found, an attempt to logically delete it is performed by changing val to F_DEL with CAS atomic operation (line 27). In the case of success, the queue's stack is updated unless it has already been updated by another thread (line 30).

If the count of deleted nodes $nMarkedNodes$ surpasses threshold R, a purge operation is attempted (line 33). Importantly, during the traversal of deleted nodes, val is inspected for the presence of a reference to a newer version of the queue's head (line 17). If a reference is found, the search continues from the newer head (lines 21-23).

C. Details of *purge* operation

Algorithm 11 outlines the purge operation. Given the head node hn and the last node to purge prg, the purge operation proceeds only if no other purge operation is ongoing (lines 4-5) and if hn corresponds to the queue's current head node (lines 6-9). The purge operation introduces new sentinel head node $hnNew$ and a copy of prg, $prgNew$. For each dimension d, the $LocatePivot$ function determines the last node (pvt) to be purged at this dimension. If $pvt.child[d]$ is marked with F_ADP, the purge is restarted (lines 23-27). Otherwise, $pvt.child[d]$ is marked with the F_PRG flag (line 57), to prevent it from being changed, and the reference is adapted by either $hnNew$ or $prgNew$ (lines 28-38). When all dimensions have been processed, $hn.val$ and $prg.val$ are updated with references to help maintaining the deletion stack (lines 41-42) and the deletion stack is updated if needed by function UpdateStackAfterPurge.

D. Updating deletion stack

After an insert or a purge operation the deletion stack may need to be updated. Algorithm 12 performs such update after inserting a node. Several cases are possible after insert.

Case 1. The versions of the queue's current stack and the stack after the insertion are the same.

Case 1a. The insert point has lower priority than the last node to be logically deleted. In this case the stack should not be rewound. However, if the queue's stack is older than this insert operation, another concurrent operation, which hasn't seen the effect of this insert, can update the stack and make the inserted node inaccessible. Thus, the stack should be renewed (lines 7-8).

Case 1b. The insert point has higher priority than the last node to be logically deleted. In this case the stack must be rewound (line 10).

Case 2. The version of the queue's current stack is older than the version of the stack after the insertion.

Case 2a. The last node to be logically deleted has lower priority than the prg node that corresponds to the stack's current version. In this case the stack must be rewound to the next version of the head, which is stored in $prg.val$. It may not be the latest head, but the stack will eventually reach the latest head and every non-deleted node is guaranteed to be accessible (line 16).

Case 2b. The last node to be logically deleted has higher priority than than the prg node that corresponds to the stack's

Algorithm 11: Purge

```
void purge(HeadNode *hn, Node *prg, Handler* h) {
  if (!notPurging.load()) return;
  bool temp = true;
  if (!notPurging.compare_exchange_strong(temp, false))
      return;
  if ((uintptr_t) (hn) ≠ head.load()) {
    notPurging.store(true);
    return;
  }
  nMarkedNodes.store(0);
  HeadNode* hnNew = h->newHeadNode();
  Node* prgNew = h->newNode();
  prgNew->setFromNode(prg);
  hnNew->val = F_DEL;
  hnNew->ver = hn->ver + 1;
  hnNew->key = hn->key;
  setCoords(hnNew, 0);
  for (int i=0; i<D; i++) hnNew->child[i].store(NIL);
  int d = 0;
  Node* pvt = hn;
  uintptr_t child;
  while(d < D) {
    if (!LocatePivot(prg, pvt, d, child)) {
      pvt = hn;
      d = 0;
      continue;
    }
    if (hn == pvt) {
      hnNew->child[d].store(child);
      prgNew->child[d].store(F_ADP);
    } else {
      prgNew->child[d].store(child);
      if (d == 0 || prgNew->child[d-1].load() == F_ADP)
          {
        hnNew->child[d].store((uintptr_t) prgNew);
      } else {
        hnNew->child[d].store(NIL);
      }
    }
    d++;
  }
  hn->val.store(SetMark((uintptr_t) prg,F_DEL));
  prg->val.store(SetMark((uintptr_t) hnNew, F_DEL));
  head.store((uintptr_t) hnNew);
  Stack* s = h->newStack();
  UpdateStackAfterPurge(s, hnNew);
  notPurging.store(true);
  return;
}

inline bool LocatePivot(Node* prg, Node* &pvt, int d,
    uintptr_t &child) {
  while (pvt->k[d] < prg->k[d]) {
    finishInserting(pvt, d, d);
    pvt = (Node*) (Clear(pvt->child[d]));
  }
  do {
    child = pvt->child[d];
  } while (!IsMarked(child,F_ALL) && !pvt->child[d].
      compare_exchange_weak(child, SetMark(child, F_PRG
      )));
  if (IsMarked(child,F_ADP)) {
    return false;
  } else {
    child = ClearMark(child, F_PRG);
    return true;
  }
}
```

Algorithm 12: Rewinding deletion stack after insert

```
1  inline void RewindStack(Stack* s, Node* n, Node* pred,
       int dp) {
2    //NOTE: no need to rewind stack if node is already
         deleted...
3    for (bool first_iteration = true; !IsMarked(n->val,
         F_DEL); first_iteration = false) {
4      Stack* sNow = stack.load();
5      if (s->head->ver == sNow->head->ver) {
6        if (n->key > sNow->node[D - 1]->key) {
7          if (!first_iteration) break;
8          *s = *sNow;
9        } else {
10         for (int i=dp; i<D; i++) s->node[i] = pred;
11       }
12     } else if (s->head->ver > sNow->head->ver) {
13       Node* prg = (Node*) (ClearMark(sNow->head->val,
             F_DEL));
14       if (prg->key ≤ sNow->node[D-1]->key) {
15         s->head = (HeadNode*) (ClearMark(prg->val, F_DEL
               ));
16         for (size_t i=0; i<D; i++) s->node[i] = s->head;
17       } else {
18         if (!first_iteration) break;
19         *s = *sNow;
20       }
21     } else { // s->head->ver < sNow->head->ver
22       Node* prg = (Node*) (ClearMark(s->head->val, F_DEL
             ));
23       if (prg->key > n->key) {
24         for (int i=dp; i<D; i++) s->node[i] = pred;
25       } else {
26         s->head = (HeadNode*) (ClearMark(prg->val, F_DEL
               ));
27         for (int i=0; i<D; i++) s->node[i] = s->head;
28       }
29     }
30     if (stack.compare_exchange_strong(sNow, s)) {
31       break;
32     }
33   }
34 }
```

current version. In this case the stack should not be rewound but must be renewed as in Case 1a (lines 18-19).

Case 3. The version of the queue's current stack is newer than the version of the stack after the insertion.

Case 3a. The item was inserted into purged region (that is, $prg.key > n.key$, where prg corresponds to the head of the stack after the insertion). In this case the stack must be rewound. It will eventually reach the latest head and every non-deleted node is guaranteed to be accessible (line 24).

Case 3b. The item was inserted after prg. In this case the stack must be updated to the next version of the head, which is stored in $prg.val$. It may not be the latest head, but the stack will eventually reach the latest node and every non-deleted node is guaranteed to be accessible (lines 26-27).

Algorithm 13 demonstrates the procedure of updating the stack after the purge. Because only a purge operation can increase the version of the head, the number of possible cases in this situation is smaller than after insertion. The current queue's stack should only be updated if it is passed the prg node that corresponds to the stack's head (lines 11-12), as in Case 2a for the insertion. If the queue's current stack is before the prg node that corresponds to its version, the stack must be renewed (lines 14-15), as in Case 2b for the insertion.

Algorithm 13: Updating deletion stack after purge

```
inline void UpdateStackAfterPurge(Stack* s, HeadNode*
    hnNew) {
  for (bool first_iteration = true; true;
      first_iteration = false) {
    Stack* sNow = stack.load();
    if (hnNew->ver ≤ sNow->head->ver) {
      // The stack has been updated already
      return;
    }

    Node* prg = (Node*) (ClearMark(sNow->head->val,
        F_DEL));
    if (prg->key ≤ sNow->node[D-1]->key) {
      s->head = (HeadNode*) (ClearMark(prg->val, F_DEL))
          ;
      for (size_t i=0; i<D; i++) s->node[i] = s->head;
    } else {
      if (!first_iteration) break;
      *s = *sNow;
    }
    if (stack.compare_exchange_strong(sNow, s)) {
      break;
    }
  }
}
```

IV. Evaluation

A. Tests

The correctness of the implementation was evaluated by a series of tests, briefly described below, which are essentially the same as those used for the lock-based implementation [3].

Test 1. Sequential execution: The first set of tests evaluates correctness of the implementation during sequential execution. In the first stage, elements with all possible priorities in the range from 0 to 262144 except 100 were inserted into the queue, in ascending order. Then an element with priority 100 was inserted. Then DeleteMin was called repeatedly and the correct order of elements extracted was checked.

The second stage checks whether elements are correctly extracted in ascending order after they are inserted in descending order.

The third stage ensures that the elements are correctly extracted if they have been inserted in a pseudo-random order.

Tests 2-4. Concurrent execution: The second test checks that if elements are inserted concurrently by 4 threads, they will be later extracted in the correct order.

The third test ensures that if elements are inserted concurrently and also extracted concurrently afterwards then each extracting thread will observe consistent order of extraction.

The last test checks whether the priority queue operates normally in the case of inserts and extractions by different threads. Each thread during the test performs a randomized mixture of inserts and extractions.

B. Benchmarks

Performance of the lock-free and MRLock-based versions of the priority queue was evaluated in three mini-benchmarks representing different patterns and ratios of invocations of insert and DeleteMin. To highlight the functioning of the implementations, memory for all dynamic objects was preallocated before execution of the benchmarks. The MDList-based version of the queue from [3] was modified accordingly.

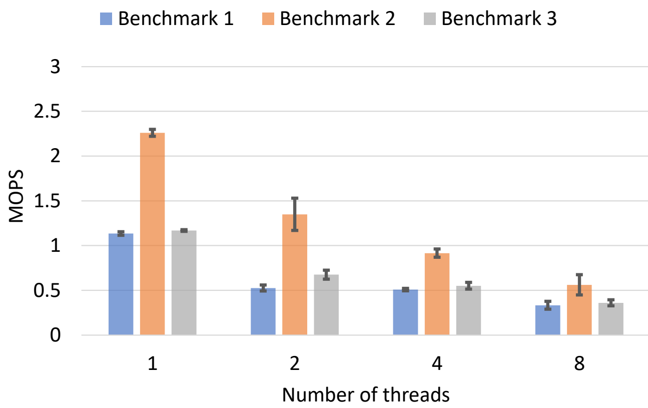
All threads in a benchmark are executing similar jobs, which are described in detail in [3]. In the first benchmark, all threads first only insert elements into the queue, then DeleteMin operators are called after all threads have finished all insertions.

In the second and third benchmarks, each thread randomly calls insert and DeleteMin operations with a given pair of probabilities summing to one. The probability of calling DeleteMin is 0.5 for benchmark 2 and 0.2 for benchmark 3. However, to prevent depletion of the queue, if a number of DeleteMin operations performed by a thread reaches the number of insert operations performed by the same thread, the thread will perform an insert operation, regardless of a probabilistic draw.

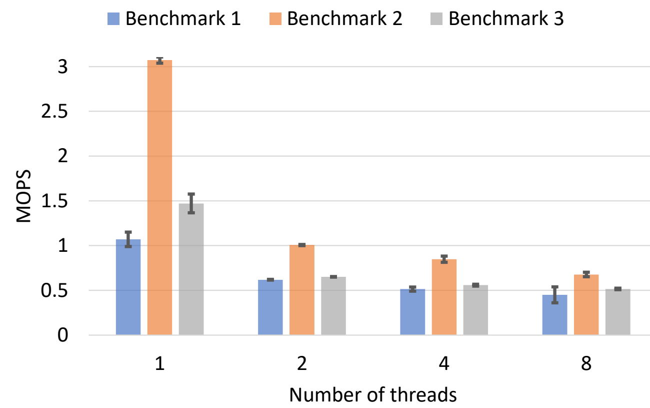
Figure 1 shows the contrast between the performance of the lock-free and MRLock-based implementations. The performance is measured as the combined throughput of all threads (in million operations per second – MOPS) of the benchmarks on two different operating systems (Windows 10 and Ubuntu 18.04.3 LTS) on a machine equipped with AMD FX-8300 (8 cores, 4 units) and 8GB DRAM when the number of threads is varied from 1 to 8 threads for the MRLock-based implementation and from 1 to 32 for the lock-free implementation. The performance characteristics were computed based on 10 observations (error bars on the figure indicate standard deviation).

The throughput of the MRLock-based implementation quickly decreases with the increase of the number of cores for all benchmarks on both operating systems. When the number of threads exceeds the number of cores, the throughput of MRLock-based implementation drops significantly and therefore is not shown on the Figure 1a,b.

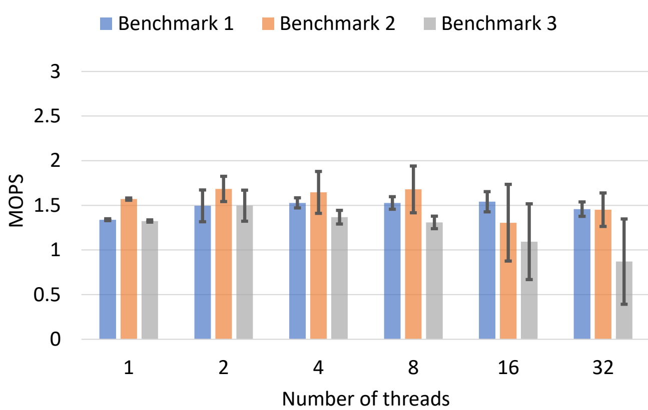
The performance of the lock-free implementation scales better. On the Windows system, its throughput is slightly increasing when the number of threads increases from 1 to 8 and then slightly decreasing with further increase in the number of threads (but the change in the performance does not appear to be substantial). On the Linux system, the throughput is increasing for benchmark 1 when the number of threads increases. For benchmarks 2 and 3, the performance the Linux system stays virtually the same for the number of threads in range from 1 to 4 and in range from 8 to 32. However, it drops significantly when the number of threads increases from 4 to 8. The poor performance of the lock-free priority queue implementation at high concurrency is associated with increase of delete nodes that DeleteMin operations must traverse before they find a node that is not marked as deleted. Perhaps, the ongoing insert operations cause some kind of disruption on the purge operations that leads to accumulation of logically deleted nodes in the queue.



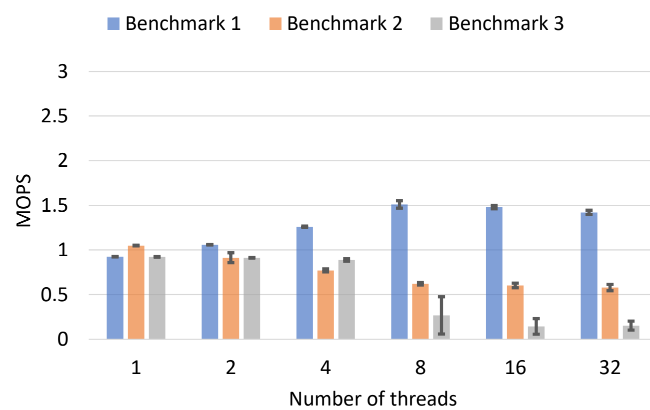
(a) Windows, MRLock



(b) Linux, MRLock



(c) Windows, Lock-free



(d) Linux, Lock-free

Fig. 1: Average performance (in million operations per second – MOPS) vs number of threads for benchmarks 1-3 executed on AMD FX-8300 (8 core 4 units): (a) MRLock-based version under Windows 10, (b) MRLock-based version under Ubuntu 18.04.3 LTS, (c) lock-free version under Windows 10, and (d) lock-free version under Ubuntu 18.04.3 LTS. The results were computed based on 10 observations. Error bars indicate standard deviation.

V. Correctness

In the implementation presented here, overlapping insert operations are linearizable provided that they do not overlap with DeleteMin operations. A successful CAS operation on line 21 of Algorithm 7 is the linearization point of an insert operation with respect to another insert operation if the element was inserted. If the element was not inserted because the key was already in the queue, the linearization point is the last load operation on line 37 before the method returns.

Overlapping DeleteMin operations are also linearizable if they do not overlap with insert operations. The linearization point of a DeleteMin operation with respect to another DeleteMin operation is either a successful CAS on line 27 of Algorithm 10 if the queue was not empty. If, however, the queue was empty, the linearization point is the last load on line 9 before returning on line 11.

Overlapping DeleteMin and insert methods are quiescently consistent provided that insert operations do not overlap with purge invocations.

Quiescent consistency is a weaker consistency property than linearizability. According to [5], quiescent consistency requires that method calls separated by a period of quiescence should appear to take effect in their real-time order, and linearizability implies quiescent consistency.

An example that demonstrates non-linearizable execution is demonstrated in Figure 2. Provided that the second DeleteMin operation reads the deletion stack at time $T1$ while insert(5) updates the stack at time $T2$, and insert(7) adds the item 7 at time $T3$, the DeleteMin method may extract item 7 at time $T4$ and return this item even though linearizability (as well as sequential consistency [5]) requires that insert(5) precedes insert(7).

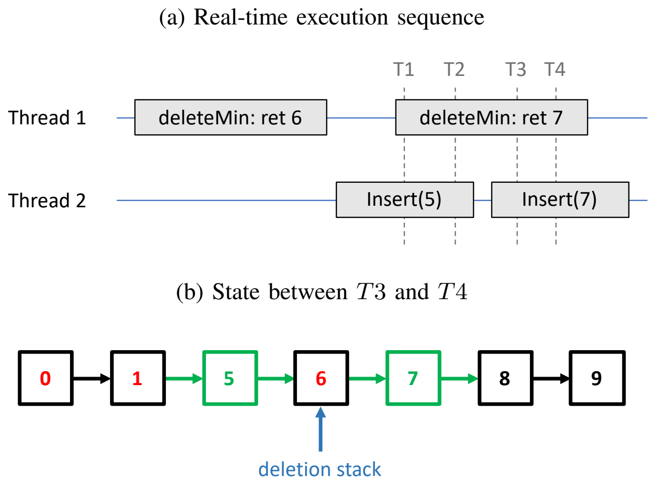


Fig. 2: Example of non-linearizable quiescent consistency.

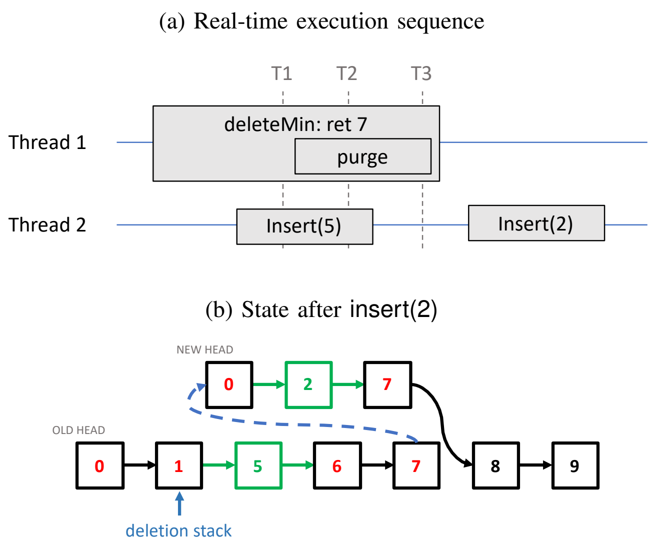


Fig. 3: Example of violation of quiescent consistency.

However, the quiescent consistency can be violated by the described above implementation if an insert operation overlaps with a purge operation. Figure 3 demonstrates an example of violation of quiescent consistency. In this example, insert(5) overlaps with DeleteMin that initiates a purge operation. Assume that the DeleteMin method extracts item 7 at time $T1$, while insert(5) inserts item 5 at time $T2$ before the purge operation updates the queue's head at time $T3$. Another instance of insert, insert(2), is called after a period of quiescence. The method will observe the new head and insert item 2 between 0 and 7. The ensuing state of the queue is shown in Fig. 3b. As a result, two subsequent DeleteMin invocations, even if separated by periods of quiescence, will first return 5 and only then 2.

To make the implementation satisfy the quiescent consistency criteria, a "corrected" version of the priority queue was introduced, in which the purge and insert operations were

Algorithm 14: Modified purge operation

```
1  void purge(HeadNode *hn, Node *prg, Handler* h){
2      ...
3    /*  Lines 2-45 from Algorithm 11    */
4      ...
5    hn = (HeadNode*) (head.load());
6    s = stack.load();
7    if (s->head->ver < hn->ver) {
8      // find elements in purged regions and reinsert
9      while (true) {
10       {val, res} = deleteCleanup(h, hn->ver);
11       if (val == NIL) {
12         // no nodes found in the purged region
13         break;
14       } else {
15         insert(key, val, h);
16       }
17     }
18   }
19   notPurging.store(true);
20   return;
21 }
```

Algorithm 15: Modified insert operation

```
bool insert(uint64_t key, uintptr_t val, Handler* h){
  // repeat until don't have to reinsert
  while (true){
      ...
    /*  Lines 2-7 from Algorithm 7    */
      ...
    while (true) {
        ...
      /*  Lines 9-20 from Algorithm 7    */
        ...
      if (pred->child[dp].compare_exchange_strong(temp,
          (uintptr_t) n)) {
        finishInserting(n, dp, dc);
        HeadNode* hn_used = s->head;
        RewindStack(s, n, pred, dp);
        HeadNode* hn_now = (HeadNode*) (head.load());
        while (true) {
          if (hn_now->ver ≤ hn_used->ver) {
            return true;
          }
          Node* prg = (Node*)(Clear(hn_used->val));
          if (prg->key ≥ key) {
            break;
          }
          hn_used = (HeadNode*)(Clear(prg->val));
        }
        if (n->val.compare_exchange_strong(val, F_DEL))
            {
          break;
        }
        return true;
      }
    }
  }
}
```

modified as follows. Before returning, purge operation checks whether the deletion stack corresponds to the outdated stack (lines 5-7 of Algorithm 14). If this is indeed the case, the purged region is scanned for nodes that were not yet deleted, by calling repeatedly the new deleteCleanup operation. The deleteCleanup operation is similar to the DeleteMin, but with two differences: (1) in addition to returning the value of the deleted element, it also returns its key and and (2) if no node has been found in the purged region, instead of traversing into the nodes that has not been purged, it exits without deleting a node and returns NIL instead of a node value. If the deleteCleanup operation on line 10 of Algorithm 14 deleted an element, the element is reinserted back into the queue (line 15) and thus it is moved outside of the purged region into the region that has not been purged. As soon as no element is found in the purged region, the purge operation exits (lines 19-20).

Because the purge operation may return before a concurrent insert operation updated the deletion stack, the insert operation also must perform a check after updating the deletion stack to make sure that the inserted element doesn't stay in the purged region. The modification to the operation is shown on Algorithm 15. Basically, a check is performed to determine whether the just inserted node is in the purged region (lines 16-24). If the node is determined to be in the purged region, and attempt to delete the node is performed with CAS operation (line 26). If the node was deleted successfully, it is reinserted back into the queue. Otherwise, the insert operation returns.

A. *Evaluation of the corrected implementation*

Performance of the corrected priority queue was evaluated by executing the same three mini-benchmarks. Figure 4 shows the the throughput of the modified lock-free implementations on the same system and under the same conditions as the original implementation presented on Figure 1c,d. If compared with the throughput of the original implementation, the throughput of the corrected implementation is more varied, which manifests in higher standard deviation. Most likely, the performance of the corrected implementation depends stronger on the order of operations, which in the benchmarks 2 and 3 was produced by a random generator.

According to results of the benchmarks execution, the corrections necessary for quiescent consistency do not compromise the performance of the lock-free priority queue very significantly. Moreover, the throughput measured on the benchmark 2 on Windows 10 for 4 and 8 threads, surprisingly, improved to some extend. Possibly, cleaning of purge regions, which was introduce in the corrected version, leads to more efficient subsequent purge and DeleteMin operations.

It is perhaps of interest to estimate how often the purge region cleaning procedures, introduced in the corrected implementation, are getting invoked. Sometime, such as in case of benchmark 2 and 16-32 threads, the fraction of purge operations that execute lines 8-17 of Algorithm 14 may exceed 90%. On average, approximately 1 element of the queue is reallocated by the purge operation per 100 invocation of these lines of code. The insert operations perform reinsert approximately once per 10 purges. Thus, the introduced measures to ensure the quiescent consistency may have significant effect in application.

VI. Literature Survey and Comparison With MDList

Both sequential and concurrent priority queues are generally built from array-based heaps. A heap can be viewed as a binary tree. A min-heap maintains the min-heap property, which states that every node is smaller than both of its children. Array-based heaps display heavy memory contention when a newly inserted key ascends to its target location and after the top item is removed [1]. Furthermore, the heap invariant is not maintained most of the time during the execution of an insert or DeleteMin operation, impeding concurrent access to the structure. Therefore, efficient lock-free heap-based implementations of a priority queue are lacking. The heap based implementation from Intel Threading Building Blocks, an established industry standard concurrent library, employs a dedicated aggregator thread to perform all operations and therefore is not lock-free [1]. In general, heap-based multithread priority queues do not scale well.

In this section, we compare a few of the most advanced modern approaches to concurrent priority queues.

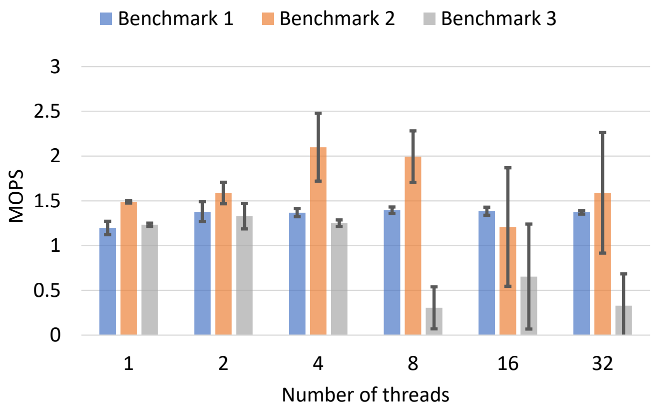
A. *Skiplists*

The structure of the skiplist [7] enables lock-free priority queue implementations that outperform heap-based implementations on modern multi-core systems. The first lock-free priority queues based on skiplists were presented by Sundell and Tisgas [8] and Herlihy and Shavit [5]. In 2013, Linden and Jonnson [9] published a paper about an approach which is sometimes referred to as LJPQ.

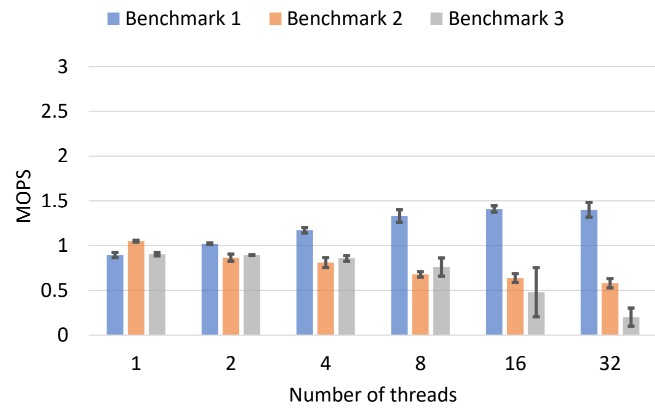
It seems to us that LJPQ is possibly the best skiplist-based concurrent priority queue. It is a linearizable implementation of the priority queue, in which (as also is the case with the MDList-based queue) the DeleteMin operation proceeds without physical deletion of nodes: the nodes are logically deleted (e.g. simply marked) but not unlinked from the list. The logically deleted nodes are physically deleted in batch "by simply moving a few pointers in the sentinel head node of the list, so that they point past logically deleted nodes, thus making them unreachable" [9]. However, physical deletion in LJPQ may have a slightly different meaning than the one to which we are most accustomed. When the nodes become unreachable they are also marked for recycling. An allocator should then take care of the nodes: make sure that no thread holds a reference to a node and then reuse or release it, thus performing a regular garbage collector's job. It is trivial to implement LJPQ with epoch-based memory reclamation in the same way as the MDList-based priority queue is designed.

Linden and Jonnson also opted for less frequent update of an atomic variable that holds the pointer to the last deleted node. Thus, DeleteMin performs a larger number of read operations when traversing the list to find the node to be deleted. This could be justified because, "due to the microarchitecture of today's processors, the cost of these reads, relative to the latencies incurred by an increased number of global writes (e.g., CAS), will be very cheap" [9]. MDList-based priority queue, on the opposite, maintains the deletion stack in order to reduce the number of traversed nodes in DeleteMin operations.

The performance of LJPQ has been measured in different benchmarks in several publications [1], [9]–[11]. On a mixture of insert and DeleteMin operations, the throughput of the queue is peaked for 4-16 threads and then decreases with the increase of the numbers of threads. The bottleneck of



(a) Windows



(b) Linux

Fig. 4: Average performance (in million operations per second – MOPS) of corrected quiescently consistent lock-free version vs number of threads for benchmarks 1-3 executed on AMD FX-8300 (8 core 4 units) (a) under Windows 10, and (b) under Ubuntu 18.04.3 LTS. The results were computed based on 10 observations. Error bars indicate standard deviation.

the queue is the DeleteMin operation [9]. The bottleneck is somewhat exacerbated by the design, because inserting of items is performed after all already deleted nodes (so that the logically deleted nodes always form a prefix of the list), which is exactly the place that the DeleteMin operations are trying to access. The throughput of MDList-based performance queue is less than of LJPQ for small number of threads because of the higher overhead of the former. However, when the number of the threads exceeds 8, the MDList-based performance queue starts to outperform LJPQ [1].

Braginsky, Cohen, and Petrank [10] reported a more involved design of a concurrent priority queue, which is known as CBPQ. It was also also based on a skiplist [5] but include many significant optimizations. The skiplist in CBPQ is composed of chunks of elements. Each chunk has a range of keys associated with it and contains all entries with keys in this range. The ranges do not intersect and the chunks are ordered in the skiplist by their ranges. The first chunk and all other chunks are build differently. The first chunk is an immutable sorted list. To delete the minimum, a thread simply atomically fetches and increments an index to this array. To insert a key to the first chunk, a thread registers this key in a special buffer and requests the first chunk rebuild. Elimination [5] can be executed on the elements in the buffer. Subsequently, a new first chunk is created from the remaining keys in the old first chunk, all keys registered in the buffer, and if needed, more keys from the second chunk. The creation of a new first chunk is also triggered when the old first chunk becomes empty. All other chunks consist of unsorted arrays. The insert operation simply finds the adequate chunk, and – provided that it is not the first chunk – adds the new element to the first empty slot in the array. When the chunk is filled, it is split into two chunks using a lock-free mechanism. For efficiency, CBPQ uses the fetch-and-increment instruction instead of the compare-and-swap (CAS) instruction.

According to the performance evaluation on a benchmark, consisting of a mixture of insert and DeleteMin operations [10], CBPQ scales better than LJPQ. However, because of higher overhead, the throughput of CBPQ is less than the throughput of LJPQ if the number of threads is less than 24. MDList-based priority queue is expected to outperform CBPQ for up to 32 threads.

Although skiplists proved to be an efficient base for concurrent priority queue implementations, they have several potential drawbacks. For instance, although average time complexity of inserting an item in a skiplist is $O(\log N)$, the worst-case time is $O(N)$, where N is the number of items in the structure. Also, inserting an item into a skiplist involves updating several distant nodes, which may cause interference among concurrent operations and reduce throughput in a lock-free scenario.

B. *Relaxed Priority Queues*

The major bottleneck of concurrent priority queues is due to the inherent sequential semantics of the DeleteMin operation [12]. Two approaches have been employed to alleviate this bottleneck: consistency relaxation [1] and semantics relaxation. The first approach was demonstrated by Herlihy and Shavit [5] and Zhang and Dechev [1] through implementation of quiescently consistent rather than linearizable priority queues. The second approach was explored by Alistarh et al. [13], who introduced the Spraylists structure, Wimmer et al. [14], who developed k-log-structured merge-trees, and Rihani et al. [15], who pioneered MultiQueue.

In 2015, Alistarh et al. published a paper titled "The SprayList: A scalable relaxed priority queue" [13]. A common theme that we found during our literature survey was a tradeoff between scale and correctness of the DeleteMin function (or operation), and we believe this tradeoff is clearly implied in this case by the fact that the SprayList's DeleteMin returns an element among the first $O(P \log^3 P)$ where P is the number of threads. Alistarh et al. state, "Starting from a non-blocking SkipList, the main innovation behind our design is that the DeleteMin operations avoid a sequential bottleneck by 'spraying' themselves onto the head of the SkipList in a coordinated fashion. The spraying is implemented using a carefully designed random walk, so that DeleteMin returns an element among the first $O(p \log^3 p)$ in the list, with high probability, where p is the number of threads" [13]. This is clearly an example of a Relaxed Priority Queue, as the semantics of DeleteMin are relaxed here. In contrast, MDList is quiescently consistent [1], and MDList's DeleteMin is much more likely to return the element associated with the minimal key. As Alistarh et al. state, competing threads at the first element (min) are made "to 'skip ahead' in the list, so that concurrent operations attempt to remove distinct, uncontended elements" [13]. Alistarh et al. also state, "The SprayList provides probabilistic guarantees on the priority of returned elements" [13]. In contrast, probabilistic guarantees are not required with MDList. Alistarh et al. then state, "The obvious issue with this approach is that one cannot allow threads to skip ahead too far, or many high priority (minimal key) elements will not be removed. Our solution is to have the DeleteMin operations traverse the SkipList, not along the list, but via a tightly controlled random walk from its head. We call this operation a spray" [13].

With k-LSM, Wimmer et al. presented a "new, lock-free concurrent priority queue built from a logarithmic number of sorted arrays of keys, similar to the log-structured merge-trees used in databases" [14]. As is commonly done with concurrent priority queues, k-LSM relaxes the semantics of the DeleteMin operation. In contrast, MDList is quiescently consistent [1], and MDList's DeleteMin is much more likely to return the element associated with the minimal key. Notably, k-LSM puts a limit, $\rho + 1$, on the highest rank-order of the key associated with an element deleted by the DeleteMin operation, where ρ is related to the number of threads P and a runtime configurable parameter k by the equation $\rho = P \cdot k$. Clearly, in any practical application of k-LSM, ρ will be significantly greater than both k and P. Furthermore, linearization requirements on insertions are relaxed with k-LSM [14]. The experiments of Wimmer et al. showed "high single thread performance" [14], meaning that the 'overhead' of their implementation is low enough that their queue has good performance for a single thread (not much worse than a heap for example). Wimmer et al. also found that k-LSM has "very good scalability when choosing a reasonably large value for k." [14]. Again, we believe that a common trade-off with concurrent priority queues is between scale and correctness of the DeleteMin operation, and that this is clearly applicable to k-LSM queue and is clearly implied regarding k-LSM by the previous quote.

Rihani, Sanders, and Dementiev introduced a MultiQueue data structure [15]. This conceptually simple priority queue is build from $C \cdot P$ sequential priority queues protected by locks, where C is a constant and P is the number of threads. The insert operation tries to lock at random one of the queues, and, if the lock was successful, inserts in that queue. The DeleteMin operation selects at random two queues and then tries to lock the queue with the smallest minimum element out of the two selected. If the lock was successful, the actual DeleteMin operation is performed on the locked queue. The MultiQueue does not provide any guarantee, but according to experimental evaluation, the rank error of the MultiQueue is smaller on average and more tightly distributed than of the SprayList, while the throughput of the former is much higher than of the later [15]. Note that the MulitQueue is lock-based, although a preempted thread that holds a lock does not block the progress of other threads because the number of the sequential queues exceeds the number of threads.

A highly conceptually involved design of the priority queue, named CA-PQ, was presented by Sagonas and Winblad [11]. The main idea of CA-PQ is introduction of thread-local buffers for insert and DeleteMin operations, which capacities increase or decrease depending on contention detected on the global priority queue. In case of no contention, the capacities of the thread-local buffers are equal to zero, thus all operations go through the global queue. Therefore, CA-PQ is a semantically correct priority queue when no contention. If a thread detects contention, it increases the capacity of the thread-local buffers. If the insert buffer is full or the DeleteMin buffer is empty, the thread performs a bulk operation on the global priority queue. Thus, activating contention avoidance reduces the number of accesses to the global priority queue, while at low contention the semantic correctness is restored. Hence, CA-PQ scales well and has performance advantage over related relaxed data structures.

We believe that a common trade-off with concurrent priority queues is between scale and correctness of the DeleteMin operation. Because rank errors of DeleteMin operations may result in wasted computations, the optimal balance between throughput and correctness may depend on the application. For instance, even though all relaxed semantic priority queues had higher throughput than LJPQ, only CA-PQ and MultiQueue outperformed LJPQ in calculation of single source shortest path in an unweighted LiveJournal graph [11]. In contrast, MDList is quiescently consistent [1], and MDList's DeleteMin is much more likely to return the element associated with the minimal key. It may be more advantageous in applications where rank errors of the DeleteMin operations lead to weighty performance penalty.

VII. Conclusions and Future Work

Overall, in this lock-free version, the performance is about the same when the number of threads increases. We did however see increases in performance in some cases with an increasing number of threads. In contrast, the performance of the lock-based version [3] in this series of papers decreased quickly with increasing numbers of threads.

The presented implementation correctly performs the operations of a priority queue in a multithreaded environment. The performance stays about the same when the number of threads increases.

Our strongest takeaway from our literature survey was that scale and correctness of the DeleteMin operation often trade off with one another in Relaxed Priority Queues and Skiplist-based approaches to Priority Queues. It may be better to use MDList [1] if one is working at scale and if out-of-order operations cannot be easily tolerated, i.e. if correctness of the DeleteMin operation is important, than to use a Relaxed Priority Queue or a Skiplist-based Priority Queue.

References

[1] D. Zhang and D. Dechev, "A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 613–626, 2016.

[2] A. Goponenko and S. Carroll, "A C++ implementation of a lock-free priority queue based on Multi-Dimensional Linked List," *ResearchGate*, 11 2019.

[3] S. Carroll and A. Goponenko, "A C++ implementation of a threadsafe priority queue based on multi-dimensional linked lists and MRLock," *ResearchGate*, 10 2019.

[4] D. Zhang, B. Lynch, and D. Dechev, "Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors," in *OPODIS 2013: Principles of Distributed Systems*, pp. 266–280, Springer, Cham, dec 2013.

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[6] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing* (J. Welch, ed.), (Berlin, Heidelberg), pp. 300–314, Springer Berlin Heidelberg, 2001.

[7] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," in *WADS 1989: Algorithms and Data Structures*, pp. 437–449, Springer, Berlin, Heidelberg, 1989.

[8] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[9] J. Lindén and B. Jonsson, "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention," in *OPODIS 2013: Principles of Distributed Systems*, pp. 206–220, Springer, Cham, dec 2013.

[10] A. Braginsky, N. Cohen, and E. Petrank, "CBPQ: High Performance Lock-Free Priority Queue," pp. 460–474, Springer, Cham, 2016.

[11] K. Sagonas and K. Winblad, "The contention avoiding concurrent priority queue," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10136 LNCS, pp. 314–330, 2017.

[12] F. Ellen, D. Hendler, and N. Shavit, "On the Inherent Sequentiality of Concurrent Objects," *SIAM Journal on Computing*, vol. 41, pp. 519–536, jan 2012.

[13] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A scalable relaxed priority queue," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, vol. 2015-Janua, pp. 11–20, 2015.

[14] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-LSM relaxed priority queue," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, vol. 2015-Janua, pp. 277–278, 2015.

[15] H. Rihani, P. Sanders, and R. Dementiev, "MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues," *arXiv*, nov 2014.