A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists

Deli Zhang and Damian Dechev

Abstract—The throughput of concurrent priority queues is pivotal to multiprocessor applications such as discrete event simulation, best-first search and task scheduling. Existing lock-free priority queues are mostly based on skiplists, which probabilistically create shortcuts in an ordered list for fast insertion of elements. The use of skiplists eliminates the need of global rebalancing in balanced search trees and ensures logarithmic sequential search time on average, but the worst-case performance is linear with respect to the input size. In this paper, we propose a quiescently consistent lock-free priority queue based on a multi-dimensional list that guarantees worst-case search time of $\mathcal{O}(\log N)$ for key universe of size N. The novel multi-dimensional list (MDList) is composed of nodes that contain multiple links to child nodes arranged by their dimensionality. The insertion operation works by first injectively mapping the scalar key to a high-dimensional vector, then uniquely locating the target position by using the vector as coordinates. Nodes in MDList are ordered by their coordinate prefixes and the ordering property of the data structure is readily maintained during insertion without rebalancing nor randomization. In our experimental evaluation using a micro-benchmark, our priority queue achieves an average of 50 percent speedup over the state of the art approaches under high concurrency.

Index Terms—Concurrent data structure, priority queue, lock-freedom, multi-dimensional List, skiplist

1 Introduction

Scalable non-blocking priority queues are pivotal to the performance of parallel applications on current multi-core and future many-core systems. Attempts on parallelizing search algorithms, such as best-first search [3], do not achieve the desired performance gains due to the lack of scalable concurrent priority queues. Priority task scheduling [27] and discrete event simulation applications [16] also demand high-throughput priority queues to efficiently distribute workload. A priority queues is an abstract data structure that stores a set of key-value pairs where the keys are totally ordered and interpreted as priorities. A priority queue is defined only by its semantics that specify two canonical operations: Insert, which adds a key-value pair, and DeleteMin, which returns the value associated with the key of highest priority and removes the pair from the queue. In sequential execution scenarios, priority queues can be implemented on top of balanced search trees or array-based binary heaps [4]. The latter is more efficient in practice because its compact memory footprint exploits spatial locality that optimizes cache utilization. For concurrent accesses by a large number of threads balanced search trees suffer from sequential bottlenecks due to required global rebalancing. Array-based heaps suffer from heavy memory contention in the heapify operation when a newly inserted key ascends to its target location.

In recent research studies [7], [12], [16], [25], concurrent priority queue algorithms based on skiplists are gaining momentum. A skiplist [21] is a linked list that provides a probabilistic alternative to search trees with logarithmic sequential search time on average. It eliminates the need of rebalancing by using several linked lists, organized into multiple levels, where each list skips a few elements. Links in the upper levels are created with exponentially smaller probability. Due to the nature of randomization, skiplists still exhibit less than ideal linear worst-case search time. Skiplist-based concurrent priority queues have a distributed memory structure that allows concurrent accesses to different parts of the data structure efficiently with low contention. However, Insert operations on skiplists involve updating shortcut links in distant nodes, which incurs unnecessary data dependencies among concurrent operations and limits the overall throughput. Another bottleneck faced by concurrent priority queues is the inherent sequential semantics required by the DeleteMin operation [5]. Threads competing to remove the minimal element from the queue squander most of their effort trying to decide which one gets the minimal node. The best existing approach employs logical deletion and batch physical deletion to alleviate the contention on head nodes [16], but the parallelism achieved by this approach is still limited because threads performing deletion have to traverse all logically deleted nodes and are forced to wait for slow insertions.

In this paper, we present a quiescently consistent lock-free priority queue design based on a multi-dimensional list (MDList). The proposed multi-dimensional list stores ordered key-value pairs and guarantees worst-case sequential search time of $\mathcal{O}(\log N)$ where N is the size of the key universe. It is composed of nodes that contain multiple links to the child nodes arranged by their dimensionality, and provides convenient concurrent accesses to different parts

• *The authors are with the Department of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32826. E-mail: de-li.zhang@knights.ucf.edu, dechev@eecs.ucf.edu.*

Manuscript received 13 Nov. 2014; revised 19 Mar. 2015; accepted 30 Mar. 2015. Date of publication 2 Apr. 2015; date of current version 12 Feb. 2016. Recommended for acceptance by G. Tan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2419651

of the data structure. The insertion works by first injectively mapping a scalar key into a high dimensional vector, then uniquely locating the target position using the vector as coordinates. Similar to a trie data structure [26], a node in MDList shares the same coordinate prefix with all of its parent nodes, and the search is done through prefix matching rather than key comparison. As a result, the ordering property of the data structure is readily maintained during insertion without rebalancing nor randomization. The proposed priority queue satisfies lock-freedom, which ensures system wide progress while allowing individual threads to starve [13]. It has the following algorithmic characteristics that aim to further improve the throughput over existing approaches by exploiting a greater level of parallelism and reducing contention among concurrent operations.

- Each insertion modifies at most two consecutive nodes, which allows concurrent insertions to be executed with minimal interference.
- A *deletion stack* is used to provide hints about the next minimum node so that DeleteMin operations do not need to traverse logically deleted nodes.
- Insertions proceed optimistically without blocking overlapping deletions; they synchronize with DeleteMin operations only when necessary by rewinding the deletion stack.

In our experimental evaluation, we compare our algorithm with Intel TBB's priority queue and the best available skiplist-based priority queues using a micro-benchmark on a 64-core NUMA and a 12-core SMP system for three types of workload. The result shows that on the NUMA system our algorithm outperforms the alternative approaches by an average of 50 percent under high concurrency. On the SMP system, our algorithm achieves the same level of performance as the best skiplist-based approach for mixed workload and outperforms the alternatives for insert-only workload. We also show that the dimensionality of an MDList-based priority queue can be tuned to fit different workloads.

The rest of the paper is organized as follows. In Section 2, we review existing concurrent priority queue algorithms. In Section 3, we define the multi-dimensional list data structure and introduce the sequential search algorithm. We present the concurrent Insert and DeleteMin operations for our MDList-based lock-free priority queue in Section 4. We reason about its correctness and progress properties in Section 5. The performance evaluation and result analysis is given in Section 6. We conclude the paper in Section 7.

2 Related Work

As a fundamental data structure, concurrent priority queues have been extensively studied in the literature. Early concurrent priority queue algorithms are mostly adaptations of the sequential heap data structure. Hunt et al. [15] present a fine-grained locking approach that is built on a number of earlier heap-based algorithms. Herlihy [11] uses an array-based binary heap as an example for the universal construction of lock-free data structures. It requires that each thread only modify a local copy of the heap and updates the pointer to the heap using atomic operations. This serves as a fault tolerance scheme to avoid deadlocks rather than exploit fine-grained parallelism. Both of these approaches, like all heap-based algorithms, suffer from sequential bottlenecks and contention due to the compact memory layout of the data structure. We omit detailed discussion on more heap-based approaches as empirical evidence collected on modern multi-core platforms shows that they are outperformed by recent skiplist-based structures [23], [25].

Pugh [20] designs a concurrent skiplist with per-pointer locks, where an update to a pointer must be protected by a lock. Shavit [24] discovers that the highly decentralized skiplist is suitable for shared-memory systems and presents a bounded priority queue based on Pugh's algorithm using fixed bins. However, this approach only supports a small set of fixed priorities whereas our approach supports an arbitrary range of keys. Shavit and Lotan [23] also propose the first unbounded concurrent priority queue based on a skiplist. Their locking approach employs logical deletion (by marking the target) that kept a specialized pointer to the current minimal item, and each DeleteMin operation has to traverse the lowest level list until it finds an unmarked item. A lock-free adaptation of this algorithm was later presented by Herlihy and Shavit [13]. Sundell and Tsigas [25] present the first linearizable lock-free priority queue based on a skiplist. They guarantee linearizability by forcing threads to help physically remove a node before moving past it. Herlihy et al. [12] propose an optimistic concurrent skiplist that simplifies previous work and allows for easy proof of correctness while maintaining comparable performance. Recently, Linden and Jonsson [16] propose a skiplist-based lock-free priority queue that addresses the sequential bottleneck of the DeleteMin operation. Their design uses a logical deletion scheme similar to Shavit and Lotan's approach described above, but provides significant performance improvement by performing physical deletions in batch. The drawback was that the logically deleted nodes have to always form a prefix of the lowest level list, and physical deletions cannot pass ongoing insertions.

As recognized by the above researches, the DeleteMin operation presents the biggest scalability challenge for concurrent priority queues. Concurrent data structures that are strictly adherent to linearizability often pay a price for performance and scalability. Relaxed semantics for concurrent data structures are proposed to provide considerable performance benefit when tight synchronization is not absolutely necessary [1], [10]. Henzinger et al. [10] proposed a formalization of semantics relaxation for concurrent priority queues. Alistarh et al. [2] present a relaxed algorithm by allowing deletion threads to randomly pick a node within a certain priority range. Wimmer et al. [27] study the performance of semantically relaxed priority queues for task scheduling. While the above approaches study the trade-offs of semantic relaxation, this paper tries to exploit the performance benefit of relaxing the consistency criteria.

The proposed MDList also bears some similarity to trie data structures [8], [26] in that the keys are ordered by their prefixes: a node always shares the same key prefix with its parent nodes. Lock-free trie designs such as the concurrent tries [19] and the skiptrie [18] provides log-logarithm search time, but their algorithm cannot be readily transfered to a priority queue implementation. One difficulty is that the minimum item in a trie is stored in a leaf node, so its

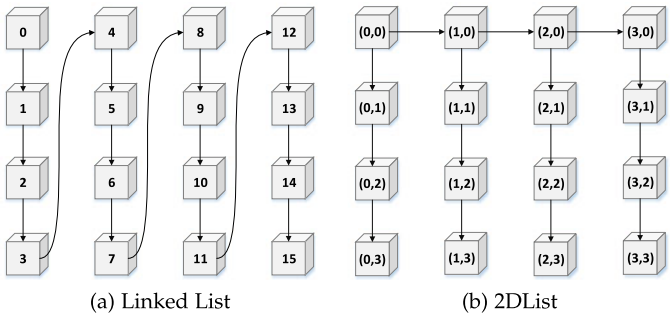


Fig. 1. From linked list to 2DList.

retrieval has to be done through repetitive search. Designing linearizable DeleteMin is also challenging in a trie, because new minimum items can be concurrently inserted at higher level while the DeleteMin operation proceeds to lower levels. MDList, on the other hand, behaves like a heap where the root node always holds the item with the smallest key.

3 Multi-Dimensional List

The core idea of a multi-dimensional list is to partition a linked list into shorter lists and rearrange them in a multi-dimensional space to facilitate search. Just like a point in a D-dimensional space, a node in a D-dimensional list can be located by a D-dimensional coordinate vector. The search operation examines one coordinate at a time and locates correspondent partitions by traversing nodes that belong to each dimension. The search time is bounded by the dimensionality of the data structure and logarithmic time is achieved by choosing D to be a logarithm of the key range.

3.1 From Linked List to 2DList

An ordered linked list keeps its nodes sorted linearly as if they were attached to a one-dimensional line. To find an existing node or insert a new node, one starts from the head and examines each node consecutively. The search time is linear with respect to the list capacity. The linked list illustrated in Fig. 1a has 16 nodes partitioned into four columns. This arrangement reveals a property concealed by the typical one-dimensional representation: the nodes are sorted along the columns as well as across the rows, and each column contains a unique range of nodes that have greater keys than the nodes in the previous column. Because a column is substantially shorter than the entire list, we can locate a node much faster if we firstly determine which column the node resides, and search for it within the column. However, a one-dimensional linked list lacks two essential properties required by the above search scheme: 1) shortcut links among columns which allow the search operation to switch from one column to another; 2) a function that maps keys into key ranges covered by each column.

We show a sample construction of a 2DList that satisfies these requirements in Fig. 1b. We replace the pointers linking the bottom nodes to the top nodes in Fig. 1a by links among the top nodes so that the top of each column form a one-dimensional linked list. This essentially converts the linked list into a tree, but we find it more intuitive to visualize it as a multi-dimensional list when discussing the algorithms in the following sections. We also compute a two-dimensional vector $\mathbf{k} = (k_0, k_1)$ based on the integer key, which serves as a node's unique coordinates in the two-dimensional space. We discuss the mapping from integer keys to vectors in more detail in Section 3.4. The nodes are arranged such that the top nodes of each column are ordered by k_0, while every node in one column shares the same k_0 and is ordered by k_1. We call the top nodes *dimension 0 nodes* (because they are ordered by the 0th element in the coordinate vector), and the nodes in each column *dimension 1 nodes*. To search for a node, for example $(2, 3)$, we begin by traversing the dimension 0 nodes as if we were traversing a one-dimension linked list. The difference is that instead of examining the actual key we examine the k_0 of each node and compare it against the target's k_0. We stop at node $(2, 0)$, and continue to traverse the third column, which consists of dimension 1 nodes. We look for the node with $k_1 = 3$ and eventually locate the target. The average and worst-case search times are reduced to a square root of the original times, and we can further improve them by extending the 2DList into higher dimensions.

3.2 Definition

We generalize the construction of the 2DList to a list of arbitrary dimensions and give the following definition of our multi-dimensional list.

Definition 1. *A D-dimensional list is a rooted tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, and each child is assigned a unique dimension of $d' \in [d, D)$.*

In an ordered multi-dimensional list, we associate every node with a coordinate vector $\mathbf{k}$, and determine the order among nodes lexicographically based on $\mathbf{k}$. A dimension d node should share a coordinate prefix of length d with its parent. The following requirement prescribes the exact arrangement of nodes according to their coordinates.

Definition 2. *Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, \ldots, k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, \ldots, k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i,\ \forall\, i \in [0, d) \wedge k_d > k'_d$.*

A fundamental property is that a high-dimensional list can be decomposed into low-dimensional lists: every dimension 0 node in a D-dimensional list can be seen as the root node of a $(D-1)$-dimensional list. If we repeat the decomposition process recursively, we obtain multi-dimensional lists with fewer and fewer dimensions and eventually we arrive at a single node. This is analogous to affine subspaces, or flats, in a $\mathbb{Z}^D_{\geq 0}$-affine space. The design of the search algorithm detailed in Section 3.5, is based on prefix traveling the of the affine space.

3.3 Structures

We define the structures of a sequential MDList in Algorithm 1. The dimension of an MDList is denoted by a constant integer D and the range of the keys by N. The class of MDList itself also contains a pointer to the head (root) node. The functions presented in the following sections are

member functions of the MDList class, which means they have access to the class fields by default. A node in MDList contains a key-value pair, an array $k[D]$ of integers as the coordinate vector, and an array of child pointers in which the dth pointer links to a dimension d child node. Since nodes of high dimensions have less children than nodes of low dimensions, for a d dimension node it is possible to allocate a child array that fits only d pointers to reduce memory consumption. However, for simplicity, we choose to allocate child arrays of size D where D is the dimension of the MDList for all nodes. For a dimension d node, only the indices d through $D-1$ of its child array are valid while the rest are unused. We do not need to store the dimension of a node in the node, it is implicitly deduced by the search algorithm based on the index of the node's pointer in its parent's child array.

Algorithm 1. MDList Structures

```
1: class MDList
2:     const int D
3:     const int N
4:     Node* head
5: struct Node
6:     int key, k[D]
7:     void* val
8:     Node* child[D]
```

3.4 Vector Coordinates

There are two requirements for the function that maps a scalar key into a high dimensional vector: 1) it is injective or one-to-one (i.e. distinctness among the keys are preserved); 2) it is monotonic (i.e. the original order among the scalar keys are preserved in the vectors, which are decided by Definition 2). Additionally, it would be beneficial if the vector coordinates are uniformly distributed so that each dimension of the MDList holds comparable number of nodes. Uniform distribution minimizes the number of nodes in each dimension and consequently optimize the search time. A counter example would be mapping all keys to the first coordinate (i.e. $f(k) \mapsto (k, 0, \ldots, 0)$), which essential degenerate MDList to a linked list because nodes are not populated to higher dimensions. There are infinitely many functions that meet the above requirements. In Algorithm 2, we present a simple method that uniformly maps integer keys within the range $[0, N)$ into a coordinate vector.

Algorithm 2. Mapping from Integer to Vector

```
1: function KeyToCoord (int key)
2:     int basis ← ⌈∛N⌉, quotient ← key, k[D]
3:     for i ∈ (D, 0] do
4:         k[i] ← quotient mod basis
5:         quotient ← ⌊quotient ÷ basis⌋
6:     return k
```

We first compute the maximum number of nodes in each dimension based on the range of the keys N. In practice, the range of keys is usually known prior to the execution either explicitly through the user specification of the user application or implicitly through the key's data type. For example, if the keys are 32-bit integers and we choose the dimension of MDList to be 8, then in each dimension there are at most 16 ($\sqrt[8]{2^{32}}$) keys. We can then obtain the coordinate vector by converting the base of the original key. In the above case, each digit in the 16-based number corresponds to a coordinate in an 8-dimension vector. Given a key of 1,000, its 16-based representation is `0x3E8` and the coordinates would be (0,0,0,0,0,3,E,8). In general, to uniformly distribute keys within range $[0, N)$ in a D-dimension space, the maximum number of keys in each dimension is $b = \lceil \sqrt[D]{N} \rceil$. The mapping from an integer key to its vector coordinates is essentially converting it to an b-based number and using each digit as an entry in the D-dimension vector.

Algorithm 3. Search for a Node with Coordinates **k**

```
1: function SEARCH (int[] k)
2:     Node*, curr ← head, int d ← 0
3:     while d < D do
4:         while curr ≠ NIL and k[d] > curr.k[d] do
5:             curr ← curr.child[d]
6:         if curr = NIL or k[d] < curr.k[d] then return NIL
7:         else d ← d + 1
8:     return curr
```

Like previous skiplist-based algorithms [7], [16], [25], we focus on unique integer keys in this paper. Duplicate priorities can be handled by building keys so that only some bits represent real priorities and the rest bits are chosen to distinguish unique instances of the same priority. For example, if the application desires 65,536 priority levels one can use 32-bit integer keys and dedicate the lower 16 bits as unique identifiers.

3.5 Search

We outlined the search process for a simple 2DList in Section 3.1. Following the same methodology, we present the search algorithm for a D-dimensional list in Algorithm 3. The SEARCH function returns the node containing the coordinates **k** if it exists. Given a D-dimensional list, we search for an element by traversing the dimension d nodes that do not overshoot the node containing the coordinates being searched for (line 3.4).[1] When it is not possible to make further progress at the current dimension, the search advances to the next dimension (line 3.7). According to the condition on lines 3.4 and 3.6, the search only increases the dimension if $curr$ node exists and $k[d] = curr.k[d]$, which guarantees that those pivot nodes always share the same coordinate prefix as the node being searched for. The outer while loop terminates when the search has visited all of the nodes in the highest dimension (line 3.3). If so, $curr$ must be immediately in front of the node that contains the desired coordinates , i.e. the search must have exhaustively compared every coordinate of $curr$ with **k** and they match. Otherwise, the search failed to proceed to the highest dimension (early termination on line 3.6), and the target node is not in the list. Fig. 2 illustrates a 3DList with up to four nodes in each dimension. To search a node, say $(3, 3, 2)$. The traverse begins at the root node and proceeds through all dimension 0 nodes. It then increases the search dimension d

1. Throughout the paper, we use $a.b$ to denote line b in algorithm a.

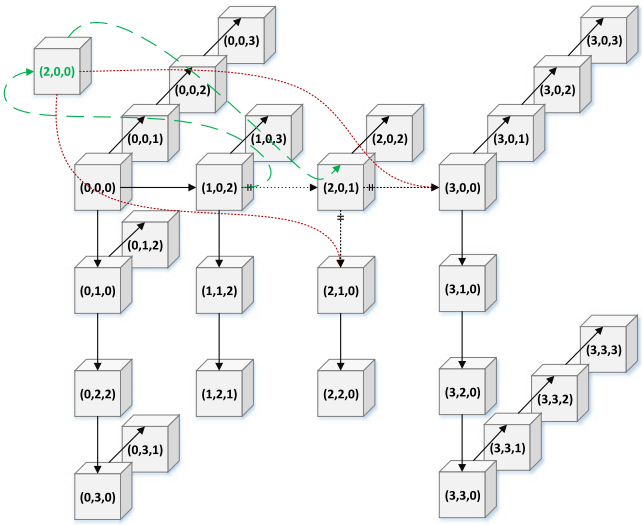


Fig. 2. INSERT operation in a 3DList.

and continues to traverse the 2DList rooted at $(3,0,0)$. The search further increases dimension and continues to traverse the 1DList rooted at $(3,3,0)$ before reaching the target node.

It is straightforward to deduce that the worst-case time complexity of the search algorithm is $\mathcal{O}(D \cdot M)$ where M is the maximum number of nodes in one dimension. For keys within the range of $[0, N)$, if we uniformly map them into the D-dimensional vectors using Algorithm 2, M is bounded by $\sqrt[D]{N}$. This gives $\mathcal{O}(D \cdot \sqrt[D]{N})$, which is equivalent to $\mathcal{O}(\log N)$, if we choose $D \propto \log N$ (Note that $\sqrt[\log N]{N} = 2$). This serves as a guideline for choosing D in sequential scenarios. For example, a 32DList that holds two nodes in each dimension provides comparable performance to that of balanced search trees with 32-bit integer keys.

3.6 Insertion and Deletion

A unique property of MDList, which makes it suitable for concurrent accesses, is the locality of its insertion and deletion: each operation requires updating at most two consecutive nodes in the data structure. For brevity, we outline the sequential insertion and deletion in this section. The pseudo code is presented in Section 4 when we explain in detail the concurrent versions of the algorithms.

The insertion operation involves two steps: node splicing and child adoption. In the first step, we search and splice as depicted by the dashed green arrows in Fig. 2. The appropriate insert position of the new node is determined by using a modified version of the SEARCH algorithm, which keeps a record of the predecessor and the dimension of the new node (the index of the new node in its predecessor's child array). Splicing involves pointing to the old child of the predecessor from the new node and updating the child pointer of the predecessor. In Fig. 2, we insert a new node $(2,0,0)$ between its predecessor $(1,0,2)$ and the predecessor's old child $(2,0,1)$. The new node becomes a dimension 0 child of its predecessor and the old child becomes a dimension 2 child of the new node. The second step is needed when the dimension of the old child has been changed due to the insertion of a new node. This extra child adoption process ensures that the nodes which are no longer accessible through the old child can be reached through the new node. According to Definition 1 the dimension of a node's children should be no less than the dimension of the node itself. If the dimension of a node increases from d to d', its children within the range of $[d, d')$ must be adopted. As marked by the dotted red arrows in Fig. 2, the new node takes over two children $(3,0,0)$ and $(2,1,0)$ from the old child $(2,0,1)$, the dimension of which increased from 0 to 2.

The head node (n_{head}) of an MDList is always associated with the smallest key. The DELETEMIN operation should discard n_{head} by identifying the node with the next smallest key (n_{next}), and setting it to be the new head node. According to Section 3.4, n_{next} is the first non-empty child of n_{head} when we reversely traverse n_{head}'s child array from index $D-1$ to 0. After becoming the new head node, n_{next} has a dimensionality of 0. The deletion operation needs to adopt children in a similar way to that in an insertion operation: if n_{next}'s dimensionality decreased from d to 0, n_{next} inherits n_{head}'s children within the range of $[0, d)$.

4 LOCK-FREE PRIORITY QUEUE

The proposed MDList provides convenient support for concurrent INSERT operations because of its decentralized structure. As mentioned in Section 3.6, each insertion takes one or two steps and updates no more than two consecutive nodes. We guarantee lock-free progress in the node splicing step by using a single-word COMPAREANDSWAP (CAS) atomic synchronization primitive to atomically swing the child pointer in a similar way to that of a lock-free linked list [9]. To provide for lock-free progress in the child adoption step, we need to announce the operation globally using descriptor objects. This allows the interrupting threads to help finish the adoption in case the insertion thread is preempted.

The DELETEMIN operation described in Section 3.6 poses a scalability challenge for concurrent executions because constant child adoption incurs heavy contention on the head node. To avoid such performance penalties, our lock-free DELETEMIN operations mark nodes for logical deletion [6], and perform physical deletions in batch at a later stage when memory accesses are less congested.

4.1 Data Structure

We define the structures of the lock-free priority queue in Algorithm 4. The descriptor object ADOPTDESC is a data structure that stores context for a pending child adoption operation [13]. The *adesc* field in the NODE structure signifies an interrupted child adoption task, and guides other threads to help finish it. The HEADNODE is a NODE with an additional version number. We keep track of the version of the head node to help the insertion operation decide on a proper position for rewinding the deletion stack. We employ the pointer marking technique described by Harris [9] to mark adopted child nodes as well as logically deleted nodes. The macros for pointer marking are defined in Algorithm 5. F_{adp} and F_{prg} flags are co-located with the *child* pointers while F_{del} flag is co-located with the *val* field. A node is considered logically deleted when its *val* field has been marked. The *deletion stack*, STACK, consists of an array of D node pointers and a head pointer. The pointer at index $D-1$ points to the last known logically deleted node, while pointers at indices $[0, D-1)$ point to its parents at previous

dimensions. All nodes in the stack form a path through which the next unmarked node can be reached. The priority queue is initialized with a dummy head node, which has the minimal key 0. The node array in the deletion stack is initially filled with the dummy head.

Algorithm 4. Priority Queue Structures

```
1: struct Node
2:     int key, k[D]
3:     void* val
4:     Node* child[D]
5:     AdoptDesc* adesc
6:     struct HeadNode
7:     PUBLIC: Node
8:     int ver
9: struct Stack
10:    Node* node[D]
11:    HeadNode* head
12: struct AdoptDesc
13:    Node* curr
14:    int dp, dc
15: class PriorityQueue
16:    const int D, N, R
17:    HeadNode* head
18:    Stack* stack
```

Algorithm 5. Pointer Marking

```
1: const int F_adp ← 0x1, F_prg ← 0x2, F_del ← 0x1
2: define SetMark p, m (p | m)
3: define ClearMark p, m (p & ~m)
4: define IsMarked p, m (p & m)
```

4.2 Invariants

We list the invariants that are satisfied by the concurrent priority queue object at all times. A brief proof is sketched after explaining the algorithms in details. By a *node*, we refer to an object of type *Node* that has been allocated and successfully linked to an existing node. We denote the $head$ node with $ver = i$ by $head_i$, and the set of nodes that are reachable from $head_i$ by L_i. At any point of time, the set of all nodes can be denoted by $L = \bigcup_{i=0}^{m} L_i$ where $m = head.ver$.

Invariant 1 states that if a node has no pending child adoption task, its dimension d child must have d invalid child slots leaving $D - d$ valid ones.

Invariant 1. $\forall n, n' \in L, \textsc{ClearMark}(n.child[d], F_{adp} | F_{prg}) = n' \wedge n.adesc = \text{NIL} \Rightarrow \forall i \in [0, d), \textsc{IsMarked}(n'.child[i], F_{adp}) = \textbf{true}$.

Invariant 2 states that any node in L_i can be reach by following a series child pointers with non-decreasing dimensionality.

Invariant 2. $\forall n \in L_i, \exists p = \{d_0, d_1, \ldots, d_m\} : d_0 \leq d_1 \leq \cdots \leq d_m \wedge head_i.child[d_0].child[d_1]...child.[d_m] = n$.

Invariant 3 states that the ordering property described by Definition 2 is kept at all times.

Invariant 3. $\forall n, n' \in L, n.child[d] = n' \Rightarrow n.key < n'.key \wedge \forall i \in [0, d)\ n.k[i] = n'.k[i] \wedge n.k[d] < n'.k[d]$.

Invariant 4 specify the condition for the shared $stack$ to be valid.

Invariant 4. $stack.head \underbrace{.child[0]...child[0]}_{0\ or\ more\ copies} = stack.node[0] \wedge \forall d \in [1, D),\ stack.node[d-1] \underbrace{.child[d]...child[d]}_{0\ or\ more\ copies} = stack.node[d]$.

4.3 Concurrent DeleteMin

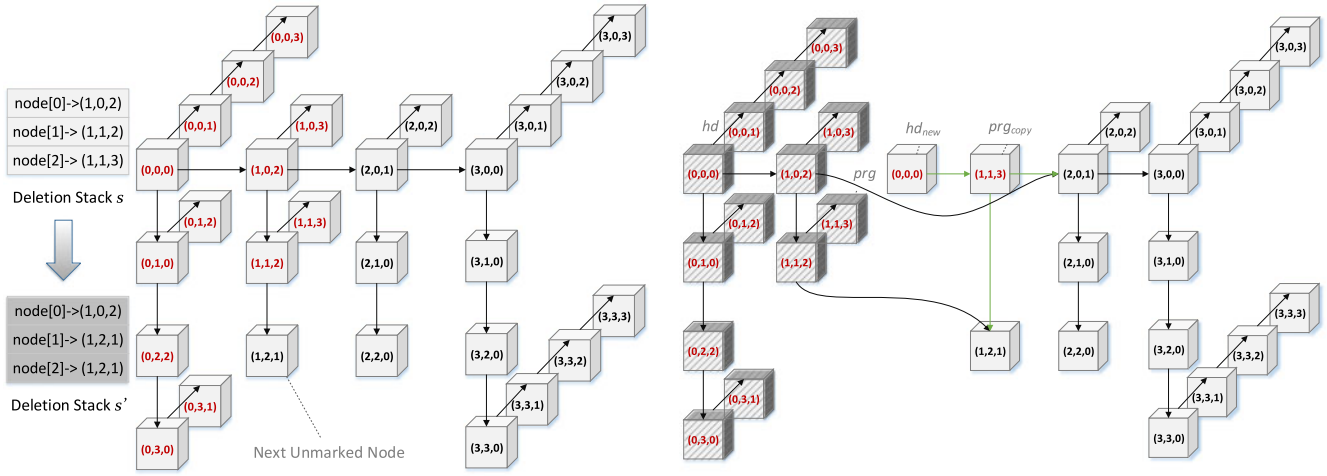
One limitation with previous logical deletion approaches is that they need to traverse all logically deleted nodes before reaching the next unmarked node [16], [23]. This proves to be troublesome especially for an MDList-based priority queue. Since an MDList is a rooted tree, reaching for the minimum node is done by a traversing procedure similar to a depth first search. As the search extends to higher dimensions, it traverses exponentially more nodes, which slows down logical deletion. Additionally, a naive implementation of the search algorithm based on recursion may lead to poor performance because of the overhead of function calls. We use a *deletion stack* to hint the DeleteMin operation about the position of next smallest node. The advantage of employing the deletion stack is twofold: 1) it reduces node traversal by providing hints about the position of the next smallest node; 2) it converts the typically recursion-based search algorithm into an iteration-based one.

4.3.1 Logical Deletion

Fig. 3a illustrates an example of how the deletion stack s is used in a logical deletion operation. Nodes in red are marked for deletion. To find the next unmarked node, the DeleteMin operation starts the search from the last entry of the stack instead of the head node. It reads $s.node[2] = (1, 1, 3)$ and examines the dimension 2 child $s.node[2].child[2]$. If the next smallest node exists, its earliest possible position should be $s.node[2].child[2]$. This is because $s.node[2]$ has the largest key among known marked nodes, and its smallest children should be assigned with the highest dimensionality according to Definition 2. In this example, node (1,1,3) does not have any child, so the operation traces back the stack to find the parent node of (1,1,3). It reads $s.node[1] = (1, 1, 2)$ and examines the child $s.node[1].child[1] = (1, 2, 1)$, which is the next unmarked node. After marking node (1,2,1) for deletion, the operation needs to submit an updated deletion stack s' to reflect its progress. s' is obtained by replacing the last two entries in s with the newly marked node (1,2,1).

Algorithm 6 lists the concurrent DeleteMin operation for a D-dimensional list. The operation backtracks the stack s until it reaches the top (line 6.6) or finds an entry $s.node[d]$ whose child $s.node[d].child[d]$ is not marked (line 6.22). In concurrent executions, $s.node[d].child[d]$ might have been marked by competing DeleteMin threads. If this is the case, as detected by the condition check on line 6.15, the operation updates the local copy of the deletion stack (line 6.17)[2] and

2. We use indexing notion $[a : b]$ to address elements within the range of $[a, b)$.



(a) Logical Deletion and Deletion Stack (b) Purge Node $(1, 1, 3)$

Fig. 3. Concurrent DeleteMin (logically marked nodes are in red; physically removed nodes are shaded).

starts anew (line 6.18). Note that the operation does not need to examine $s.node[d].child[d'],\ \forall d' \in [d+1, D)$ because it must have examined them in previous iterations given that s is valid (Invariant 4). After successfully marking a node, the operation updates the shared stack (line 6.24) using CAS.

Algorithm 6. Concurrent DeleteMin

```
1:  function (DeleteMin)
2:    Node* min ← NIL
3:    Stack* s_old ← stack, s ← newStack
4:    *s ← *s_old
5:    int d ← D − 1
6:    while d > 0 do
7:      Node* last ← s.node[d]
8:      FinishInserting(last, d, d)
9:      Node* child ← last.child[d]
10:     child ← ClearMark(child, F_adp|F_prg)
11:     if child = NIL then
12:       d ← d − 1
13:       continue
14:     void* val ← child.val
15:     if IsMarked(val, F_del) then
16:       if ClearMark(val, F_del) = NIL then
17:         s.node[d : D] ← child
18:         d ← D − 1
19:       else
20:         s.head ← ClearMark(val, F_del)
21:         s.node[0 : D] ← s.head, d ← D − 1
22:     else if CAS(child.val, val, SetMark(val, F_del)) then
23:       s.node[d : D] ← child, min ← child
24:       CAS(&stack, s_old, s)
25:       if marked_node > R and not_purging then
26:         Purge (s.head, s.node[D − 1])
27:       break
28:     return min
```

4.3.2 Physical Deletion

The physical deletion operation, or Purge, discards logically deleted nodes in batch and prepare them for memory reclamation. The existence of purge is not needed for correct execution of our priority queue algorithm, but timely purges ensure efficient execution of the logical deletion operations. We initiate a physical deletion if the logical deletion operation had to traverse too many marked nodes before reaching the unmarked node (line 6.25). The threshold R should be sufficient large such that the purge takes advantage of batch processing, but not too large to hinder the progress of logical deletion operations. We give more details on choosing appropriate threshold R in Section 6.

In Fig. 3b, we purge the group of logically deleted nodes by introducing a new dummy head node (hd_{new}). The layout of both the purged nodes (shaded ones) and the remaining nodes (black ones), as well as the linkage between the two groups, are preserved. This effectively produces two valid MDLists rooted at hd and hd_{new} respectively. Subsequent insertion operations will start from hd_{new}, but ongoing insertions that started from hd can still finish. The placeholder node prg_{copy}, which is a copy of prg, is necessary to maintain the dimensionality of the remaining nodes. Without prg_{copy}, node (1,2,1) has to be promoted as the dimension 0 child of hd_{new} in the new MDList. This alternation to the layout of the remaining nodes causes inconsistent dimensionality, e.g., node (1,2,1) is still the dimension 1 child of node (1,1,2) when accessed through the old MDList head hd.

Algorithm 7 lists the detailed purge operation. Given the last node to purge (prg), the LocatePivot function[3] identifies for each dimension $d \in [0, D)$ a pivot node (pvt) such that $\forall i \in [0, d),\ pvt.k[i] = prg.k[i] \wedge pvt.k[d] \leq prg.k[d]$. The pivot node ($pvt$) divides the d-dimensional sublist such that itself along with its predecessors will be purged while its successors will be kept. $pvt.child[d]$ is linked to either hd_{new} or prg_{copy} in a manner that preserves its current dimensionality. To prevent $pvt.child[d]$ from being changed by concurrent insertions, we employ the same pointer marking technique as the child adoption process using a different flag F_{prg} (line 7.27).

3. For clarity of presentation, we use the notion of *inline* functions, which have implicit access to caller's local variables without explicit argument passing.

Algorithm 7. Batch Physically Deletion

```
1: function PURGE(HeadNode* hd, Node* prg)
2:   if hd ≠ head then return
3:   Node* hd_new ← new Node, prg_copy ← new Node
4:   *prg_copy ← *prg, prg_copy.child[0 : D] ← NIL
5:   hd_new.val ← F_del, hd_new.ver ← hd.ver + 1
6:   for d ← 0, pvt ← hd; d < D do
7:     if !LOCATEPIVOT() then
8:       pvt ← hd, d ← 0
9:       continue
10:    if pvt = hd then
11:      hd_new.child[d] ← child
12:      prg_copy.child[d] ← F_del
13:    else
14:      prg_copy.child[d] ← child
15:      if d = 0 or prg_copy.child[d − 1] = F_del then
16:        hd_new.child[d] ← prg_copy
17:    d ← d + 1
18:  hd.val ← SETMARK(prg, F_del))
19:  prg.val ← SETMARK(hd_new, F_del))
20:  head ← hd_new
21: inline function LOCATEPIVOT()
22:   while prg.k[d] > pvt.k[d] do
23:     FINISHINSERTING(pvt, d, d)
24:     pvt ← CLEARMARK(pvt.child[d], F_adp|F_prg)
25:   repeat
26:     child ← pvt.child[d]
27:   untilCAS(&pvt.child[d], child, SETMARK(child, F_prg)
        or ISMARKED(child, F_adp|F_prg)
28: if ISMARKED(child, F_prg) then
29:   child ← CLEARMARK(child, F_prg)
30:   return true
31: else return false
```

After the purge, nodes between hd and prg can be safely reclaimed when all threads have concluded their operations on the old MDList rooted at hd. We use an epoch-based garbage collector [6], [20] to reclaim the memory of dynamically allocated nodes. This kind of garbage collection is light weight and efficient, but it requires that the objects which are longer reference to be explicitly added to a garbage list. Since we only process the physical deletion in the purge function, the thread performing purge are solely responsible for putting purged nodes on the list. Each purge creates a garbage list identified by the version of the head being purged. Every time a thread starts an operation it observes the current head version: for INSERT it is $head.ver$; for DELETEMIN it is $stack.head.ver$. Once all threads have observed the same head version v, the garbage list associated with $v-1$ can be safely reclaimed. For the descriptor objects and deletion stacks, we employ reference courting [17] because those are transient objects that do not reference each other. We could reuse them as quickly as possible so that the heap need not to accommodate defunct-but-unusable descriptors and stacks.

4.3.3 Preservation of Invariants

DELETEMIN preserves Invariant 1.

Proof. Both logical deletion and physical deletion operations do not modify nodes' F_{adp} flags. □

DELETEMIN preserves Invariant 2.

Proof. Logical deletion does not alter linkage among nodes. Physical does not alter existing links, so invariant holds for $n \in L_i,\ where\ i = hd.ver$. It introduces new links that keep the invariant for $n \in L_{i+1},\ where\ i+1 = hd_{new}.ver$ because $\forall d \in [0, D),\ pvt.child[d] = hd_{new}.child[d] \vee pvt.child[d] = prg_{copy}.child[d]$. □

DELETEMIN preserves Invariant 3.

Proof. Logical deletion does not alter linkage among nodes. Physical deletion preserves the ordering property because of the condition check on line 7.22. □

DELETEMIN preserves Invariant 4.

Proof. The invariant trivially holds on line 6.21. On lines 6.23 and 6.17, The algorithm sets $s.node[d:D]$ to $child$. The invariant holds because $child = s.node[d].child[d]$. □

4.4 Concurrent Insert

Algorithm 8 lists the concurrent INSERT function. LOCATEPRED, which is an extension of the sequential search function (Algorithm 3), determines the target position by locating the new node's immediate parent $pred$ and child $curr$ (line 8.11 and 8.20). We link $pred.child[dp]$ to the new node on line 8.16. The new node should be inserted as the dimension dp child of the $pred$ node, while a non-empty $curr$ node will become the dimension dc child of the new node. Regardless of the dimensionality of the data structure, $pred$ and $curr$ are the only nodes that is updated by an insertion. The CAS-based loop restarts when the CAS fails under three circumstances: 1) the desired child slot has been updated by a competing insertion; 2) the desired child slot has been marked invalid by a child adoption process; and 3) the desired child slot has been marked invalid by a purge operation. The deletion stack needs to be rewound (line 8.18) if the new node was inserted into a position that cannot be reached by subsequent DELETEMIN operations. We explain this synchronization mechanism in details in Section 4.4.2.

Algorithm 8. Concurrent Insert

```
1: function INSERT(int key, void* val)
2:   Stack* s ← new Stack
3:   Node* node ← new Node
4:   node.key ← key, node.val ← val
5:   node.k[0 : D] ← KEYTOCOORD(key)[0 : D]
6:   node.child[0 : D] ← NIL
7:   while true do
8:     Node* pred ← NIL, curr ← head
9:     dp ← 0, dc ← 0
10:    s.head ← curr
11:    LOCATEPRED()
12:    if dc = D then
13:      break
14:    FINISHINSERTING(curr, dp, dc)
15:    FILLNEWNODE()
16:    if CAS(&pred.child[dp], curr, node) then
17:      FINISHINSERTING (node, dp, dc)
18:      REWINDSTACK()
19:      break
20: inline function LOCATEPRED()
```

```
21:   while dc < D do
22:     while curr ≠ NIL and node.k[dc] > curr.k[dc] do
23:       pred ← curr, dp ← dc
24:       FinishInserting (curr, dc, dc)
25:       curr ← curr.child[dc]
26:     if curr = NIL or node.k[dc] < curr.k[dc] then
27:       break
28:     else
29:       s.node[dc] ← curr, dc ← dc + 1
30: inline function FillNewNode()
31:   node.adesc ← NIL
32:   if dp < dc then
33:     node.adesc ← new AdoptDesc
34:     node.adesc.curr ← curr
35:     node.adesc.dp ← dp, node.adesc.dc ← dc
36:   node.child[0 : dp] ← F_adp
37:   node.child[dp : D] ← NIL
38:   node.child[dc] ← curr
```

4.4.1 Helping with Child Adoption

The FinishiInserting function in Algorithm 9 performs child adoption on a given node n. Child adoption is necessary when the insertion of n demotes its immediate child node ($curr$) to a higher dimension (line 8.32). As described in Section 3.6, n needs to adopt multiple children $curr.child[i],\ i \in [dp, dc)$. Before a child pointer can be copied, we must safeguard it so that it cannot be changed by concurrent insertions while the copy is in progress. This is done by setting the F_{adp} flag in the child pointers (line 9.10). Once the flag is set either by this thread or by other threads, the function proceeds to copy the pointer to n (line 9.12). Finally, the descriptor field in n is cleared to designate the operation's completion.

Algorithm 9. Child Adoption

```
1: function FinishInserting (Node* n, int dp, int dc)
2:   if n = NIL then
3:     return
4:   AdoptDesc* ad ← n.adesc
5:   if ad = NIL or dc < ad.dp or dp > ad.dc then
6:     return
7:   Node* child, curr ← ad.curr
8:   int dp ← ad.dp, dc ← ad.dc
9:   for i ∈ [dp, dc) do
10:    child ← FetchAndOr(&curr.child[i], F_adp)
11:    child ← ClearMark(child, F_adp)
12:    CAS(&n.child[i], NIL, child)
13:  n.adesc ← NIL
```

The child adoption task is usually completed by the thread that just successfully inserted node n (line 8.17). However, helping is enforced if another thread tries to read $n.child[d]$, $d \in [n.adesc.dp, n.adesc.dc)$ before the insertion thread could finish the child adoption task. For example, on line 8.24 LocatePred helps $curr$ node prior to reading $curr.child[dc]$. The search process can continue without restart because the child adoption appears transparent to traversing operations, i.e., the helping task does not alter already traversed nodes. In our algorithms, all accesses to a node's child pointers are lead by a call to FinishInserting, which enforces child adoption and ensures consistency.

4.4.2 Synchronizing Insert with DeleteMin

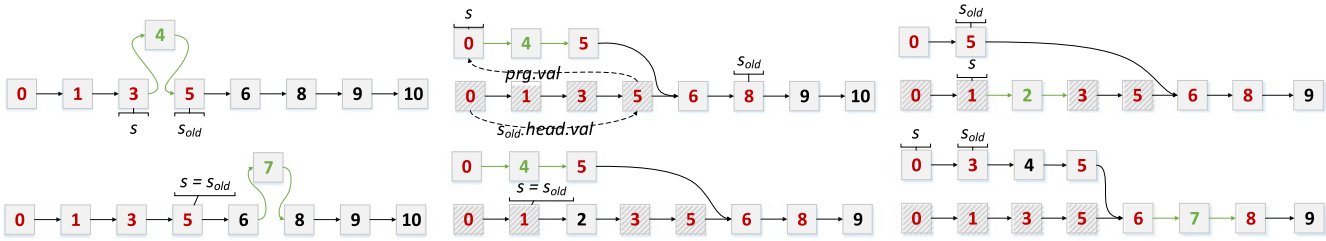
In our design, Insert and DeleteMin are synchronized through the stack rewind mechanism. Intuitively, a deletion thread "advances" the deletion stack while an insertion thread "rewinds" the stack when it detects that the stack points to a position beyond the new node. The insertion rewinds the stack to a position before the newly inserted node so that it is made visible to future deletion threads. In contrast with the previous approach [16], in which logically deleted nodes are required to form a prefix, our approach allows insertion threads to proceed optimistically without blocking deletion threads until the new nodes are in place.

Algorithm 10 determines the exact position of rewound stack s based on the temporal relation between s and s_{old} (as ordered by head version $head.ver$), and categorizes the rewinds into three scenarios as shown in Fig. 4. For simplicity, we depict logical ordering of nodes rather than the physical layout of the MDList. Fig. 4a illustrate the first case, in which the head node is unchanged (line 10.4). If s_{old} points to a node with larger key than that of the new node, we rewind the stack to the new node's predecessor (line 10.6). Otherwise, deletion threads have yet to reach the new node and we can skip the rewind except for the first iteration, in which case we update the stack as it is to make sure the new node will not be skipped by concurrent deletions (line 10.7).

Algorithm 10. Rewind Deletion Stack

```
1: inline function RewindStack()
2:   Stack* s_old ← stack
3:   repeat
4:     if s.head.ver = s_old.head.ver then
5:       if node.key ≤ s_new.node[D − 1].key then
6:         s.node[dp : D] ← pred
7:       else if first iteration then *s ← *s_new
8:       else break
9:     else if s.head.ver > s_old.head.ver then
10:      Node* prg ← ClearMark(s_old.head.val)
11:      if prg.key ≤ s_old.node[D − 1].key then
12:        s.head ← ClearMark(prg.val, F_del)
13:        s.node[0 : D] ← s.head
14:      else if first iteration then *s ← *s_old
15:      else break
16:    else
17:      Node* prg ← ClearMark(s.head.val)
18:      if prg.key ≤ node.key then
19:        s.head ← ClearMark(prg.val, F_del)
20:        s.node[0 : D] ← s.head
21:      else s.node[dp : D] ← pred
22:  until CAS(&stack, s_new, s) or Is Marked(node.prg, F_del)
```

In the second case, s is more recent than s_{old}, which means $s_{old}.head$ has been purged. As shown in the top diagram of Fig. 4b, we need to rewind the stack to the new head given by $prg.val$ to ensure the new node is reachable (line 10.12). During a purge operation, we duplex nodes' val fields to store two convenient pointers (dashed arrows in Fig. 4b): one in the purged head node pointing to the last purged node (line 7.18); another in the last purged node pointing to the new head node (line 7.19). The DeleteMin



(a) $s.head.ver = s_{old}.head.ver$ (b) $s.head.ver > s_{old}.head.ver$ (c) $s.head.ver < s_{old}.head.ver$

Fig. 4. Three stack rewind scenarios (black nodes are normal; red nodes are logically deleted; shaded nodes are purged).

operation will switch to the new head if it detects the node has been purged (line 6.19). We thus keep the stack as it is if s_{old} has not reached the last purged node. This is depicted in the bottom diagram of Fig. 4b.

In the last case, s_{old} is more recent (line 10.9). If the new node is inserted before the last purged node, as shown in the top diagram of Fig. 4c, it cannot be reached through s_{old}. We rewind the stack to the predecessor. On the other hand, if the new node is inserted after the last purged node (bottom of Fig. 4), we rewind the stack to the next head ($prg.val$) regardless the position of the s_{old} to ensure reachability.

4.4.3 Preservation of Invariants

INSERT preserves Invariant 1.

Proof. By observing the statements at lines 8.36 and 9.10 we see that the F_{adp} flags are properly initialized before linking a new node to its predecessor and updated properly whenever a child is adopted. □

INSERT preserves Invariant 2.

Proof. At the start, the structure contains a single dummy $head$ node and the invariant holds trivially. Any new node is initially placed at a position reachable from $head$ because the node traversed by LOCATEPRED form a consecutive path p'. Note the condition checks in (lines 8.22 and 8.26), we have $i < j \Rightarrow d_i \leq d_j \; \forall d_i, d_j \in p'$. Though the path may be altered by subsequent insertions, we do not unlink nodes from the data structure. The claim follows by noting that an insertion either adds a new node to p' or replaces an existing node in p', i.e., $p' \subseteq p$. □

INSERT preserves Invariant 3.

Proof. Initially the invariants trivially holds. The linkage among nodes can be changed by insertion, child adoption. Insert preserves the invariants because the condition checks on lines 8.22 and 8.26 guarantee that $\forall i \in [0, dp)\ pred.k[i] = node.k[i] \wedge pred.k[dp] < node.k[dp]$. Child adoption preserves the invariant because $\forall i \in [dp, dc)\ node.k[i] = curr.k[i] < curr.child[i].k[i]$. □

INSERT preserves Invariant 4.

Proof. The invariant holds trivially on line 10.13 and 10.20. The updates on lines 10.6 and 10.21 keep the stack entries within the range of $(dp : D)$ valid by assigning them the same value $pred$. The entries within the range of $[0, dp]$ are valid because LOCATEPRED updates them by following a path starting from $s.head$ (line 8.10) such that $\forall dc \in [1, dp],\ s.node[dc] = s.node[dc-1].\underbrace{child[dc]...child[dc]}_{0\ or\ more\ copies}$ (lines 8.25 and 8.29). □

5 CORRECTNESS

In this section, we sketch a proof of the main correctness property of the presented priority queue algorithm, which is quiescent consistency. Additionally, we discuss how stricter correctness conditions such as linearizability can be guaranteed through the use of time-stamps. We explain the relaxation of the DELETEMIN operation when the concurrent object is executed without quiescent periods. We begin by defining the *abstract state* of a sequential priority queue and then show how to map the internal state of our concrete priority queue object to the abstract state. We denote the abstract state of a sequential priority queue to be a set P. Equation (1) specifies that an INSERT operation grows the set if the key being inserted does not exist. Equations (2) and (3) specify that a DELETEMIN operation shrinks a non-empty set by removing the key-value pair with the smallest key

$$\text{INSERT}(\langle k, v\rangle) = \begin{cases} P \cup \{\langle k, v\rangle\} & \forall \langle k', v'\rangle \in P\ ,\ k' \neq k \\ P & \exists \langle k', v'\rangle \in P : k' = k \end{cases} \quad (1)$$

$$\text{DELETEMIN}() = \begin{cases} P \setminus \langle k, v\rangle & P \neq \emptyset \\ \emptyset & P = \emptyset \end{cases} \quad (2)$$

$$\text{where } \forall \langle k', v'\rangle \in P,\ k' \neq k \Rightarrow k' > k. \quad (3)$$

5.1 State Mapping

Now we consider the concurrent priority queue object. We show that the unmarked nodes which are accessible through the deletion stack form a strictly well-ordered set that is equivalent to P. Recall that we define the set of all nodes as $L = \bigcup_{i=0}^{m} L_i$ where $m = head.ver$ in Section 4.2.

Lemma 1. *At any time, nodes in L_i, including those marked for logical deletion, form an MDList that complies with Definition 1.*

Proof. Invariant 1 shows that for any node n with dimension d, only children with dimension greater or equal to d is accessible, thus the dimension of a node is always no greater than the dimensions of its children. □

Lemma 2. *Logically deleted nodes appear transparent to traversing operations.*

Proof. Note that the *val* field of a logically deleted node is marked by flag F_{del}, which renders the key-value pair stored in that node obsolete. Follow Invariant 2, a logically deleted key-value pair still occupy a valid node in the structure before it is physically removed. Its location withing the data structure is still consistent with its embed coordinates, making it a valid routing node. □

Lemma 3. $\forall d_1, d_2 \in [0, D), d_1 < d_2 \Rightarrow \forall i \in [0, d_1], stack.node[d_1].k[i] = stack.node[d_2].k[i] \wedge stack.node[d_1].k[d_2] \leq stack.node[d_2].k[d_2]$.

Proof. Initially, the stack contains a dummy head node and the invariant holds. Following Invariant 4, $stack.node[d_1] = stack.node[d_2] \vee \exists\{i_0, i_1, \ldots, i_m\} : d_1 < i_0 \leq i_1 \leq \cdots \leq i_m \leq d_2 \wedge stack.node[d_1].child[i_0].child[i_1]...child.[i_m] = stack.node[d_2]$. The lemma follows by observing Invariant 3. □

Lemma 4. $\forall d \in [0, D), stack.node[d].child[d].key > stack.node[D-1].key$.

Proof. Following Invariant 3, $stack.node[d].child[d].k[d] > stack.node[d].k[d]$. Following Lemma 3, $stack.node[D-1].k[d] = stack.node[d].k[d]$. □

In summary, Lemma 4 states that the deletion stack splits the nodes in L_i into two groups: those that are reachable by the DeleteMin algorithm and those that are not. We define the latter by $M_i = \{n | n \in L_i \wedge n.key \leq stack.node[D-1].key\}$, and $M = \bigcup_{i=0}^{m} M_i$. We also define the set of logically deleted nodes by $S = \{n | n \in L \wedge \text{IsMarked}(n.val, F_{del}) = \textbf{true}\}$. Together with Lemmas 1 and 2, the abstract state can then be defined as $P \equiv L \setminus M \setminus S$.

5.2 Quiescent Consistency and Linearizability

We now sketch a proof that our algorithm is a quiescently consistent priority queue implementation that complies with the abstract semantics. Quiescent consistency states that method calls separated by a period of quiescence should appear to take effect according to their real time order [13]. Linearizability requires a method to take effect instantaneously at some point after the operations starts and before it ends [14]. We first identify the *linearization point* for each operation to show that the Insert operation is linearizable with respect to other concurrent Insert operations, and the DeleteMin operation is linearizable with respect to other DeleteMin operations.

Theorem 1. *A successful* Insert$(\langle k, v\rangle)$ *operation that does not overlap with any* DeleteMin*, i.e. following a period of quiescence, takes effect atomically at one statement.*

Proof. If an Insert operation returns on lines 10.8, 10.15, or the second condition on line 10.22, the deletion stack is not updated. The decision point for such an operation to take effect is when the CAS operation on line 10.22 succeeds. The remaining CAS operations in the child adoption process will eventually succeed according to Lemma 5. If an Insert operation needs to rewind the deletion stack, i.e. $\langle k, v\rangle \in M \Rightarrow \langle k, v\rangle \notin P$, the decision point for it to take effect is when the CAS operation on line 10.22 succeeds. The newly updated deletion stack ensures that $stack.node[D-1].key \leq key$, thus renders the new node reachable for DeleteMin operations, i.e. $M = M \setminus \{\langle k, v\rangle, \ldots\}$. Equation (1) holds in both cases because $L = L \cup \langle k, v\rangle \wedge \langle k, v\rangle \notin M$. □

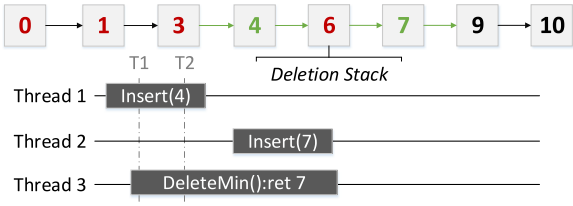


Fig. 5. Violation of real time order.

Theorem 2. *A* DeleteMin *that does not overlap with any* Insert *operation takes effect atomically at one statement.*

Proof. A DeleteMin operation updates the abstract state by growing S. The decision point for it to take effect is when the CAS operation on line 6.22 successfully marks a node for deletion, or on line 6.6 when it reaches the last node in the deletion stack without finding an unmarked node. In the first case, Equations (2) and (3) hold because $S' = S \cup \langle k, v\rangle \Rightarrow P' = P \setminus \{\langle k, v\rangle\}$. In the second case, $P' = P = \emptyset$. □

However, concurrent execution of DeleteMin and Insert operations may violate the real time ordering required quiescent consistency. Fig. 5 shows a such example. Thread 1 finishes its insertion before thread 2 starts it insertion, but thread 3, which concurrently executes DeleteMin, returns 7 because it reads the deletion stack at $T1$ before thread 1 is able to update the stack at $T2$. This is because only those nodes $\{n | n \notin M_i\}$ are reachable from $stack_i$. At the beginning of the DeleteMin operation, the deletion stack is read once on line 6.4. The subsequent traversal will not be aware of any unmarked nodes removed from M.

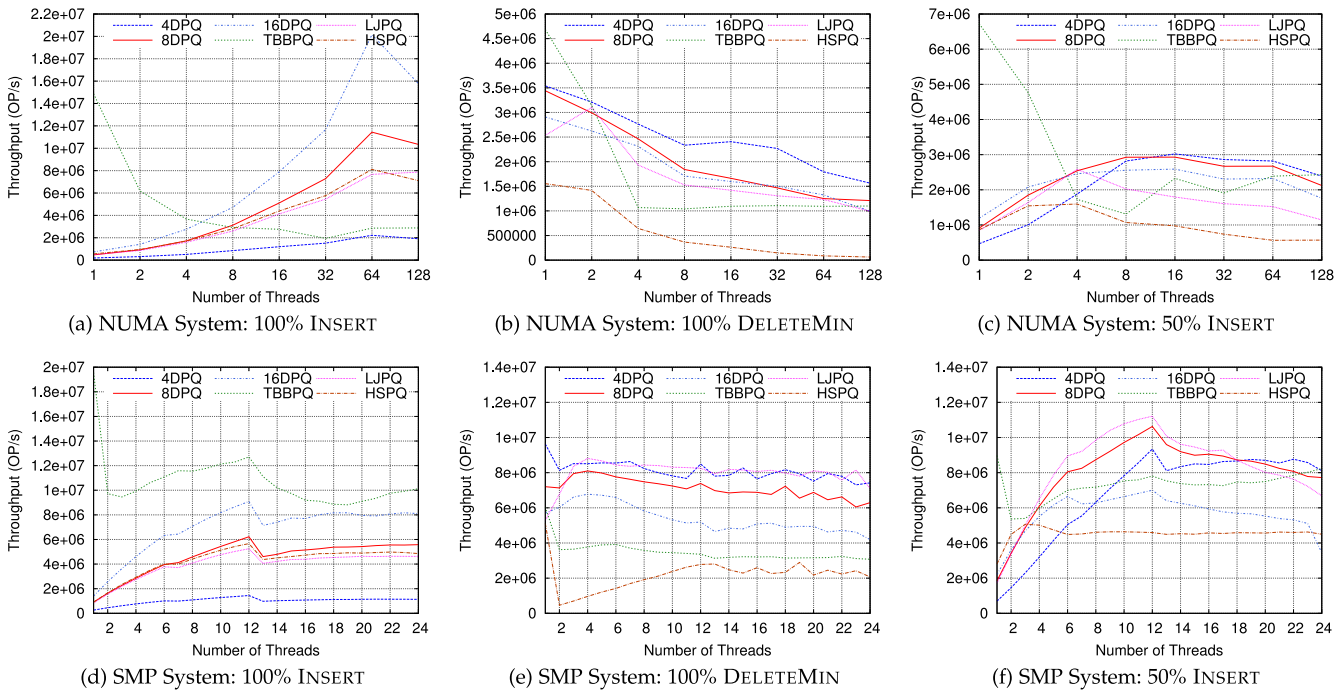
One method of designing a linearizable algorithm is the time-stamping mechanism adopted by Shavit and Lotan in [23], in which each deletion thread returns the minimal undeleted node among those inserted completely before it reads the deletion stack. After a node is inserted on line 8.16, it acquires a time-stamp. A deletion thread notes the time at which it reads the stack and only processes nodes with a smaller time-stamp. This time-stamping mechanism ensures that the DeleteMin operations see a consistent abstract state throughout the search process.

5.3 Lock Freedom

Our algorithm is lock-free because it guarantee that for every possibly execution scenario, at least one thread makes progress. We prove this by examining unbounded loops in all possible execution paths, which can delay the termination of the operations.

Lemma 5. *FinishInserting (Algorithm 9) and Purge (Algorithm 7) complete in finite steps.*

Proof. We observe that FinishInserting does not contain unbounded loop. For Purge, the `repeat` loops on line 7.27 are subject to fail and retry when another insertion thread concurrently updates $pvt.child[i]$. The number of retries is bounded by $\sqrt[D]{N}$, which is the maximum



(a) NUMA System: 100% INSERT (b) NUMA System: 100% DELETEMIN (c) NUMA System: 50% INSERT

(d) SMP System: 100% INSERT (e) SMP System: 100% DELETEMIN (f) SMP System: 50% INSERT

Fig. 6. Throughput of the Priority Queues (MDPQs are named by their dimensionality, e.g. 8DPQ is an MDPQ with eight dimensions).

number of nodes in each dimension. The `for` loop (line 7.6) is also unbounded. According to Invariant 2, the purged node can always be reached from the head through a sequence of child pointers and the number of reties is bounded by $D\sqrt[D]{N}$. □

Theorem 3. *INSERT and DELETEMIN operations are lock-free.*

Proof. Note that all shared variables are concurrently modified by CAS operations, and the CAS-based unbounded loops (lines 8.16, 6.22, and 10.22), only retry when a CAS operation fails. This means that for any subsequent retry, there must be one CAS that succeeded, which caused the termination of the loop. All reads of child pointer are preceded by FINISHINSERTING, which completes child adoption in finite steps to ensure consistency. Furthermore, our implementation does not contain cyclic dependencies between CAS-based loops, which means that the corresponding operation will progress. □

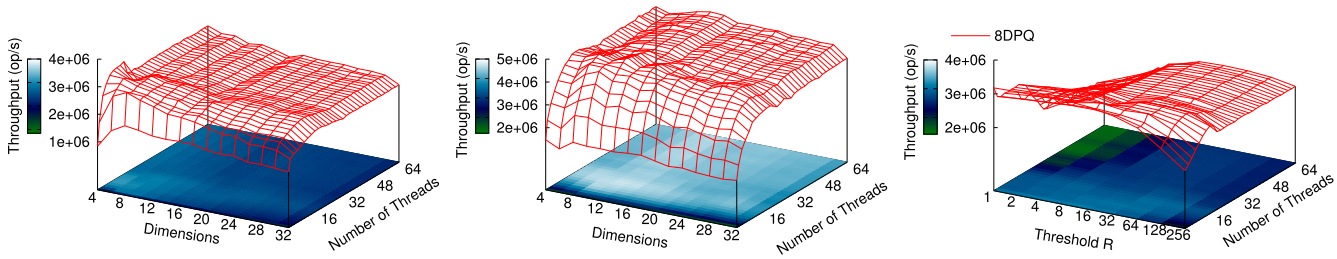
6 EXPERIMENTAL EVALUATION

We compare the performance of our algorithm (MDPQ) against Intel TBB's concurrent priority (TBBPQ) [22], Linden and Jonsson's linearizable priority queue (LJPQ) [16], and Herlihy and Shavit's quiescently consistent priority queue (HSPQ) [13]. Intel TBB is an established industry standard concurrent library. TBBPQ is based on an array-based heap and employs a dedicated aggregator thread to perform all operations. It is not lock-free because a halting aggregator thread prevents system wide progress. LJPQ is the best available linearizable priority queue that minimizes contention on the head node. HSPQ shares the same correctness guarantee as our algorithm. Both LJPQ and HSPQ are built on top of Fraser's [6] state of the art lock-free skiplist implementation and use the epoch-based garbage collection. TBBPQ allocates contiguous memory chunks with growing capacity but never free them until the termination of the object. For fair comparison of the algorithm themselves, we disable memory reclamation for all approaches and do not use any back-off strategies. We employ a micro-benchmark to evaluate the performance of these approaches for uniformly distributed unique 32-bit keys. This canonical evaluation method [9], [16], [23], [25] consists of a tight loop that randomly chooses to perform either an INSERT or a DELETEMIN operation. Each thread performs one million operations and we take the average from three runs. The tests are conducted on a 64-core NUMA system (four AMD opteron 6,272 CPUs with 16 cores per chip @2.1 GHz) and a 12-core SMP system (one Intel Xeon X5670 CPU with hyper-threading @2.9GHz). Both the micro-benchmark and the priority queue implementations[4] are compiled with GCC 4.7 with level three optimizations.

6.1 Throughput

Figs. 6a, 6b and 6c illustrate the throughput of the algorithms on the NUMA system. The y-axis represents the throughput measured by *operation per second*, and the x-axis represents the number of threads. In Fig. 6a, threads perform solely INSERT operations. We observe that the skiplist-based and MDList-based approaches explore fine-grained parallelism and exhibit similar scalability trends. The throughput increases linearly until 16 threads, and continues to increase at a slower pace until 64 threads. Because executions beyond 16 threads span across multiple chips, the performance growth is slightly reduced due to the cost of remote memory accesses. The executions are no longer fully concurrent beyond 64 threads, thus the overall throughput is capped and may even reduce due to context switching overhead. The performance of LJPQ are on par with HSPQ because they employs identical skiplist insertion

4. Source code can be requested from http://cse.eecs.ucf.edu/request.php?project=13.



(a) NUMA System: 50% INSERT (b) NUMA System: 75% INSERT (c) NUMA System: 50% INSERT

Fig. 7. Performance Impact of dimensionality and purge threshold.

algorithms. Our priority queue based on an 8DList (8DPQ) achieves a 24 percent speedup over LJPQ on 16 threads and a further 50 percent speedup on 64 threads. Each insertion in MDPQ modifies at most two consecutive nodes, incurring less remote memory access than skiplist-based approaches. When we increase the dimensionality of MDPQ to 16, we obtain on average a 50 percent throughput gain over 8DPQ. A 16DPQ contains at most four nodes in each dimension. In order to reach the target position the search operation traverses a maximum of 64 nodes comparing to 128 nodes in a 8DPQ. Further increases in dimensionality can result in diminishing returns for concurrent INSERT. The growing number of child pointers causes synchronization overhead that cancels the benefit of having less nodes in each dimension. TBBPQ, on the other hand, exhibits reduced throughput when increasing the number of threads. Its heap-based structure achieves high throughput on a single thread, but the aggregator effectively serializes all operations and limits the throughput in concurrent executions.

Fig. 6b shows the results for DELETEMIN operations. We fill the data structures with one million elements and measure the time it takes to dequeue them. The overall throughput decreases as the number of threads increases, which matches the observation that DELETEMIN is the sequential bottleneck of a priority queue algorithm. Throughput of HSPQ drops significantly because it initiates physical deletion for every logically deleted node. The physical deletion constantly swings pointers which causes heavy content on the head node. 8DPQ and LJPQ achieve comparable performance because they both employ batch physical deletion, which reduces the total number of pointer updates. 4DPQ outperforms LJPQ by 50 percent starting from 16 threads because the number of pointers, which the deletion needs to update, reduces as dimensionality decreases. TBBPQ is able to keep a constant throughput starting from 4 threads. As the DELETEMIN operation is intrinsically sequential, having a dedicated aggregator helps relieve contention by allowing other threads to wait while a single thread performs the update.

We show the results where threads randomly choose operations with a distribution of 50 percent INSERT and 50 percent DELETEMIN in Fig. 6c. As the level of concurrency increases, the overall throughput peaks at some point where the executions strike a balance between the increasing throughput of INSERT and the decreasing throughput of DELETEMIN. 8DPQ exhibits the best overall throughput, outperforming LJPQ by 50 percent starting from eight threads. Our algorithm allows insertion of new nodes into a position before a logically deleted node. The concurrent executions of INSERT and DELETEMIN guarantees quiescent consistency, which is relaxed compared to linearizability. This helps improve throughput over Linden's logical deletion scheme where deletion threads cannot proceed past an ongoing insertion [16]. 4DPQ achieves about 10 percent speedup over 8DPQ on 64 and 128 threads at the price of being slower when the number of threads is low.

Figs. 6d 6e and 6c show the throughput of the algorithms on the SMP system. In Fig. 6d MDList-based and Skiplist-based approaches exhibits similar scalability trend as on the NUMA system, with 16DPQ outperform LJPQ by as much as of 70 percent starting from 10 threads. TBBPQ performs well its contiguous heap structure is specifically optimized for Intel chips, which provides larger cache than the AMD chip. In Fig. 6e 4DPQ and LJPQ perform equally well. In Fig. 6f 8DPQ and LJPQ achieve comparable throughput, obtaining 30 percent speedup over TBBPQ and 50 percent over HSPQ with 12 threads. Executions beyond 12 threads are preemptive, and the overhead of context switching leads to a reduction of 8DPQ and LJPQ's throughput.

6.2 Tuning

In Figs. 7a and 7b, we sweep the dimension of MDPQ from 4 to 32 on the NUMA system. The lighter color on the color coded heat map indicates higher throughput. In Fig. 7a, we observe the max throughput converges towards four dimensions on most scale levels. In Fig. 7b we increase the number of insertions to create a write-dominated workload. This is to simulate the typical access pattern of best-first search algorithms, in which a worker thread insert multiple new nodes (search state) upon dequeuing the node with highest priority [3]. On all scale levels, we see that the throughput converges towards 12 dimensions. This implies that if the access pattern of the user application is taken into account the performance of MDPQ can be optimized without knowing how many threads are going to access the data structure. The dimensionality sweep also reveals that the overall throughput is capped with 16 threads. This implies that the algorithms with inherent sequential semantics, such the DELETEMIN operation, pose scalability challenges for NUMA systems.

In Fig. 7c we evaluate the performance impact of purge threshold R. We observe that the maximum throughput converges towards $32 \leq R \leq 64$. Note that $R \leq 4$ greatly limits the throughput under high concurrency. Generally, smaller R values cause more frequent purges, which incurs overhead. Larger R values result in more undeleted nodes, which slows down logical deletion. Good R values strike a balance between these two scenarios.

In summary, MDPQ provides scalable performance under high concurrency. The locality of its operations makes it suitable for NUMA architectures where remote

memory access incurs considerable performance penalties. On an SMP system with low levels of concurrency, MDPQ performs better than the state of the art skiplist-based approaches under write-dominate workload and equally well under mixed workload. MDPQ can also adapt to different workload: a high-dimensional priority queue behaves more like a tree and speeds up insertions; a low-dimensional priority queue behaves more like a linked list and speeds up deletions.

7 Conclusion

In this paper, we introduced a lock-free priority queue design based on a novel multi-dimensional list. The proposed MDList guarantees logarithmic worst-case search time by mapping keys into high dimensional coordinates. We exploited spatial locality to increase the throughput of the Insert operation, and adopted quiescent consistency to address the sequential bottleneck of the DeleteMin operation. When compared to the best available skiplist-based and heap-based algorithms, our algorithm achieved an average of 50 percent speedup on a 64-core NUMA system. We plan to further evaluate our approach on SMP system with more cores, such as Intel Xeon Phi, to verify potential performance gain under high concurrency. The performance of an MDList-based data structure can be tailored to different access patterns by changing its dimensionality. This implies the possibility of a workload aware algorithm. Furthermore, an MDList provides a scalable alternative to skiplists and search trees, which opens up opportunities for implementing other multiprocessor data structures, such as dictionaries and sparse vectors.

Acknowledgments

Deli Zhang is the corresponding author.

References

[1] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Proc. 14th Int. Conf. Principles Distrib. Syst.*, 2010, pp. 395–410.
[2] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A scalable relaxed priority queue" in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 11–20.
[3] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *J. Artif. Intell. Res.*, vol. 39, no. 1, pp. 689–743, 2010.
[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, vol. 2. Cambridge, U.K.: MIT Press, 2001.
[5] F. Ellen, D. Hendler, and N. Shavit, "On the inherent sequentiality of concurrent objects," *SIAM J. Comput.*, vol. 41, no. 3, pp. 519–536, 2012.
[6] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, Cambridge Univ. Computer Laboratory, Cambridge, U.K., 2003. Also available as Tech. Rep. UCAM-CL-TR-579, 2004.
[7] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, p. 5, 2007.
[8] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, 1960.
[9] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. 15th Int. Conf. Distrib. Comput.*, 2001, pp. 300–314.
[10] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Proc. 40th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2013, pp. 317–328.
[11] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 5, pp. 745–770, 1993.
[12] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A provably correct scalable concurrent skip list," in *Proc. Conf. Principles Distrib. Syst.*, 2006, p. 440.
[13] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Amsterdam, The Netherlands: Elsevier, 2012.
[14] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[15] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott, "An efficient algorithm for concurrent priority queue heaps," *Inf. Process. Lett.*, vol. 60, no. 3, pp. 151–157, 1996.
[16] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proc. 17th Int. Conf. Principles Distrib. Syst.*, 2013, pp. 206–220.
[17] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," DTIC Document, Ft. Belvoir, VA, USA, Tech. Rep. TR-599, 1995.
[18] R. Oshman and N. Shavit, "The skiptrie: low-depth concurrent search without rebalancing," in *Proc. ACM Symp. Principles Distrib. Comput.*, 2013, pp. 23–32.
[19] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 151–160, 2012.
[20] W. Pugh, "Concurrent maintenance of skip lists," Univ. Maryland at College Park, College Park, MD, USA, Tech. Rep. CS-TR-2222 UMIACS, 1990.
[21] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
[22] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2007.
[23] N. Shavit and I. Lotan, "Skiplist-based concurrent priority queues," in *Proc. IEEE 14th Int. Parallel Distrib. Process. Symp.*, 2000, pp. 263–268.
[24] N. Shavit and A. Zemach, "Scalable concurrent priority queue algorithms," in *Proc. 18th Annu. ACM Symp. Principles Distrib. Comput.*, 1999, pp. 113–122.
[25] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 609–627, 2005.
[26] D. E. Willard, "Log-logarithmic worst-case range queries are possible in space θ (n)," *Inf. Process. Letters*, vol. 17, no. 2, pp. 81–84, 1983.
[27] M. Wimmer, D. Cederman, F. Versaci, J. L. Träff, and P. Tsigas, "Data structures for task-based priority scheduling," *arXiv preprint arXiv:1312.2501*, 2013.

Deli Zhang is currently working toward the PhD degree at the EECS Department, University of Central Florida. He is also a research assistant in the Computer Software Engineering-Scalable and Secure Systems Lab at UCF. His research interests include developing nonblocking data structures and algorithms, applying nonblocking synchronization in existing performance critical applications.

Damian Dechev is an assistant professor at the EECS Department, University of Central Florida and the founder of the Computer Software Engineering-Scalable and Secure Systems Lab at UCF. He specializes in the design of scalable multiprocessor data structures and algorithms and has applied them in the design of real-time embedded space systems at NASA JPL and the HPC data-intensive applications at Sandia National Labs.