An Efficient Lock-free Logarithmic Search Data Structure Based on Multi-dimensional List

Deli Zhang   Damian Dechev
Department of Computer Science
University of Central Florida
Orlando, FL 32817, USA
de-li.zhang@knights.ucf.edu   *dechev@cs.ucf.edu*

***Abstract*—Logarithmic search data structures, such as search trees and skiplists, are fundamental building blocks of many applications. Although the self-balancing binary search trees are among the most ubiquitous sequential search data structures, designing non-blocking rebalancing algorithms is challenging due to the required structural alternation, which may stall other concurrent operations. Skiplists, which probabilistically create multiple levels of shortcuts in an ordered list, provide practical alternatives to balanced search trees. The use of skiplists eliminates the need of rebalancing and ensures amortized logarithmic sequential search time, but concurrency is limited under write-dominated workload because the linkage between multiple distant nodes must be updated.**

In this paper, we present a linearizable lock-free dictionary design based on a multi-dimensional list (MDList). A node in an MDList arranges its child nodes by their dimensionality and order them by coordinate prefixes. The search operation works by first generating a one-to-one mapping from the scalar keys to a high-dimensional vectors space, then uniquely locating the target position by using the vector as coordinates. Our algorithm guarantees worst-case search time of $\mathcal{O}(\log N)$ where N is the size of key space. Moreover, the ordering property of the data structure is readily maintained during mutations without rebalancing nor randomization. In our experimental evaluation using a micro-benchmark, our dictionary outperforms the state of the art approaches by as much as 100% when the key universe is large and an average of 30% across all scenarios.

***Keywords*-Concurrent Data Structure, Dictionary, Lock-free, Multi-dimensional List, Search Tree, Skiplist**

I. Introduction

Non-blocking data structures are pivotal to leverage the performance of parallel applications on current multi-core and future many-core systems. Recent years have seen rising interests in the concurrent logarithmic search data structures that implement the *dictionary* abstract data type. A dictionary stores a set of key-value pairs where the keys are drawn from a totally ordered universe. It is defined only by its semantics that specify three canonical operations: INSERT which adds a key-value pair, DELETE which removes the value associated with a specific key, and FIND which returns the associated value if the specific key is in the set. In-memory dictionaries can be implemented on top of either binary search trees (BSTs) or skiplists [1].

This work is supported by the National Science Foundation under ACI Award No. 1440530.

Binary search trees are among the most ubiquitous sequential data structures. Despite recent research efforts, designing a self-balancing non-blocking BST is challenging and remains an active topic [2], [3], [4]. One difficulty is to devise a correct and scalable design for predecessor query, which serves as the subroutine for all three operations to locate the physical node containing the target key. When executing the predecessor query concurrently with write operations, the physical location of the target key in the tree might have changed before the search finishes. This is especially troublesome when a predecessor query fails to reach the target node, under which circumstance it has to decide whether the target element is absent, or the target element's physical location has been changed by some concurrent updates [5]. The problem stems from the lack of one-to-one mapping between the *logical ordering* of keys and the physical layout of a BST. For example, the BST in Figure 1a and 1b differ in layout but represent the same ordering for integers $\{1, 6, 7\}$. If a predecessor query looking for node 6 in tree (a) gets suspended when examining node 7 and another thread transforms the tree into (b) by deleting node 4, the resumed operation would falsely conclude that node 6 does not exist. Well-orchestrated synchronization techniques, such using a leaf-oriented BST [6], and embedding logical ordering [5], have been proposed to address the issue, but they pose additional space or time overhead. Another difficulty is to cope with sequential bottleneck of rebalancing. BSTs provide logarithmic access time complexity when they are height balanced. Rebalancing is triggered to maintain this invariant immediately after the height difference exceeds a certain threshold. For concurrent accesses by a large number of threads, frequent restructuring induces contention. Mutating operations need to acquire not only exclusive access to a number of nodes to guarantee the atomicity of their change, but also locks to ensure that no other mutation affects the balance condition before the proper balance is restored [6]. Relaxed balance [6] and lazy rebalancing [7] have been suggested to alleviate contention in lock-based BSTs, but designing efficient lock-free rebalancing operations remains an open topic.

In recent research studies [8], [9], [10], non-blocking dictionaries based on skiplists are gaining momentum. A skiplist [1] 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. Skiplist-based concurrent dictionaries have a distributed memory structure that allows concurrent accesses to different parts of the data structure efficiently with low contention. However, INSERT and DELETE operations on skiplists involve updating shortcut links in distant nodes, which incurs unnecessary data dependencies among concurrent operations and limits the overall throughput. Besides, due to the nature of randomization, skiplists may exhibit less than ideal linear worst-case search time.

In this paper, we present a linearizable lock-free dictionary implementation based on a multi-dimensional list (MDList) using single-word COMPAREANDSWAP (CAS) primitives. An MDList [11] stores ordered key-value pairs in nodes and guarantees worst-case sequential search time of $\mathcal{O}(\log N)$ where N is the size of the key universe. The search works by injectively mapping a scalar key into a high dimensional vector space, then uniquely locating the target node using the vector as coordinates. The dimensionality D of an MDList is defined as the dimensionality of its vector coordinates. A node in an MDList shares a coordinate prefix with its parent node. The search is done through prefix matching rather than key comparison. Unlike previous prefix-based search data structures [12], [13], an MDList partitions the key universe in a way such that 1) the nodes sharing a common coordinate prefix form a sub-list; and 2) a node store links to at most D sub-lists arranged by the length of their shared coordinate prefixes. As a result, an MDList provides efficient concurrent accesses to different partition of the data structure, and its ordering invariant is readily maintained during insertions and deletions without rebalancing nor randomization. The proposed dictionary 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.

- Nodes are ordered by coordinate prefix, which eliminates the need of rebalancing and randomization.
- Physical layout is deterministic and independent of execution histories, which provides a unique location for each key, and simplifies the FIND algorithm.
- Each INSERT and DELETE operation modifies at most two consecutive nodes, which allows concurrent updates to be executed with minimal interference.

In our experimental evaluation, we compare our algorithm

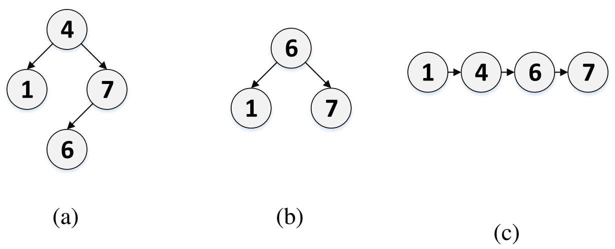


Figure 1: BSTs have various layouts for the same logical ordering (a and b). The linked list (c) has deterministic layout that is independent of execution histories.

with the best available skiplist-based [14], [10] and BST-based dictionaries [4], [15], [6] using a micro-benchmark on two hardware platforms. The result shows that on average our algorithm outperforms the alternative approaches by 30%. It exhibits excellent scalability by obtaining as much as 100% speedup over the best alternatives under high levels of concurrency with large key universes. We also show that the dimensionality of an MDList-based dictionary can be tuned to fit different application scenarios: a high-dimensional dictionary behaves more like a tree and speeds up insertions; a low-dimensional dictionary behaves more like an ordered linked list and speeds up deletions.

The rest of the paper is organized as follows. In Section II, we briefly summarize the MDList definitions and sequential algorithms. We present the concurrent FIND, INSERT, and DELETE operations of our lock-free dictionary in Section II. We reason about its correctness and progress properties in Section IV. The performance evaluation and result analysis is given in Section V. In Section VI, we review and compare our approach with a number of existing concurrent BST, skiplist and trie data structures. We conclude the paper in Section VII.

II. MULTI-DIMENSIONAL LINKED LIST

An MDList conveniently lends support to designing high-performance lock-free FIND, INSERT and DELETE operations because of its distributed and deterministic layout. The core idea of an MDList is to partition a linked list into shorter sub-lists and rearrange them in a multi-dimensional space to facilitate search. For brevity, we provide the definition of an MDList as below but refer readers to the authors' previous work [11] for detailed explanation on MDList intuitions and sequential algorithms.

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

Algorithm 1 Lock-free Dictionary Data Structure

```
1: class Dictionary
2:     const int $D$
3:     const int $N$
4:     Node* $head$
5: struct Node
6:     int $key, k[D]$
7:     void* $val$
8:     Node* $child[D]$
9:     AdoptDesc* $adesc$
10: struct AdoptDesc
11:     Node* $curr$
12:     int $dp$
13:     int $dc$
```

Algorithm 2 Pointer Marking

```
1: int $F_{adp} \leftarrow$ 0x1, $F_{del} \leftarrow$ 0x2, $F_{all} \leftarrow F_{adp}|F_{del}$
2: define SetMark($p$, $m$) ($p \mid m$)
3: define ClearMark($p$, $m$) ($p$ & $\sim m$)
4: define IsMarked($p$, $m$) ($p$ & $m$)
```

among nodes lexicographically based on $\mathbf{k}$. A dimension d node 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 $\mathbf{k} = (k_0, ..., k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i,\ \forall\ i \in [0, d) \land k_d > k'_d$.*

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 search time is achieved by choosing D to be a logarithm of the key range. To map a scalar key to a high-dimensional vector, one can choose any injective and monotonic function. In this paper, we employ KEYTOCOORD, a simple mapping function based on numeric base conversion [11]. This function maps keys uniformly to the vector space, which optimizes the average search path length for random inputs. For a key in range $[0, N)$, KEYTOCOORD converts it to the base-$\lceil \sqrt[D]{N} \rceil$ representation, and treats each digit as one coordinate. For example, given key 1234, when $N = 2^{32}$ and $D = 8$ we have $(1234)_{10} = (4D2)_{16}$. Thus the key 1234 will be mapped to vector (0,0,0,0,0,4,D,2).

III. LOCK-FREE DICTIONARY

We define the structure of the concurrent dictionary and the auxiliary descriptor object in Algorithm 1. The dimension of the dictionary is denoted by a constant integer D and the range of the keys by N. A node contains a key-value pair, an array $k[D]$ of integers that caches the coordinate vector to avoid repetitive computation, a descriptor for the child adoption process (detailed in Section III-B), and an array of child pointers in which the dth pointer links to a dimension d child node. A descriptor [16] is an object that stores operation context used by helping threads to finish a delayed operation. For a dimension d node, only

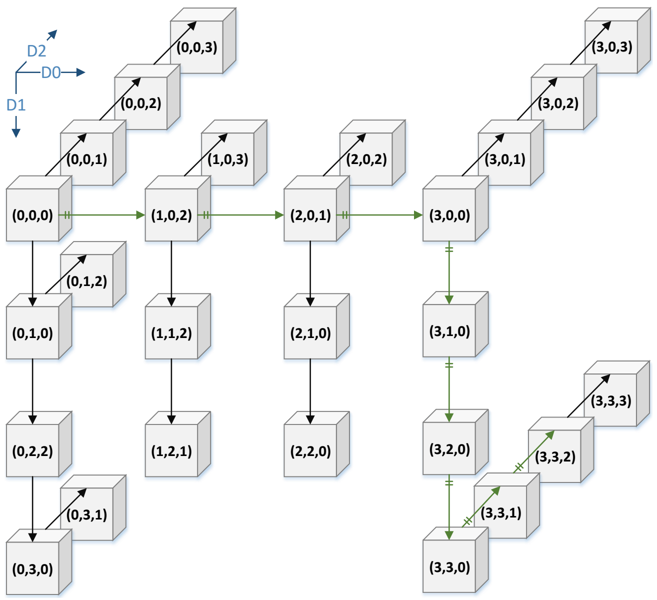


Figure 2: FIND operation in a 3DList ($D = 3, N = 64$)

the indices in $[d, D)$ of its child array are valid and the rest are unused [1]. The dictionary is initialized with a dummy head node, which has the minimal key 0. We employ the pointer marking technique [17] to mark adopted child nodes as well as logically deleted nodes. The macros for pointer marking are defined in Algorithm 2. F_{adp} and F_{del} flags are co-located with the *child* pointers.

A. Concurrent Find

We illustrate the FIND operation on an example 3DList in Figure 2. To locate the node with key 62 (coordinates (3,3,2)), the FIND operation traverse 3 sub-lists following the path highlighted by the green arrows. The worst-case time complexity of the search algorithm is $\mathcal{O}(D \cdot M)$ where M is the maximum number of nodes in each dimension. If we use previously described KEYTOCOORD to uniformly map the keys into the D-dimensional vectors, 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 $\sqrt[\log N]{N} = 2$).

We list the concurrent FIND function in Algorithm 3. The search begins from the head (line 3.4[2]). It then invokes LOCATEPRED listed in Algorithm 4 to perform the predecessor query. For clarity of presentation, we use the notion of **inline** functions, which have implicit access to the caller's local variables without explicit argument passing. The LOCATEPRED function is an extension of the sequential MDList search function [11]. Given a coordinate vector k, it tries to determine its immediate parent $pred$ and child $curr$.

[1]Since nodes of higher dimensions have less children, for a d dimension node it is possible to allocate a child array of size d to reduce memory consumption. In this paper, we demonstrate the use of constant size child array for simplicity.

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

Algorithm 3 Concurrent Find

```
1: function FIND(int $key$)
2:    Node* $curr$, $pred$
3:    int $dp$, $dc$
4:    $pred \leftarrow$ NIL, $curr \leftarrow head$, $dp \leftarrow 0$, $dc \leftarrow 0$
5:    LOCATEPRED(KEYTOCOORD($key$))
6:    if $dc = D$ then
7:       return $curr.val$
8:    else
9:       return NIL
```

Algorithm 4 Predecessor Query

```
1: inline function LOCATEPRED(int $k[\,]$)
2:    while $dc < D$ do
3:       while $curr \neq$ NIL and $k[dc] > curr.k[dc]$ do
4:          $pred \leftarrow curr$, $dp \leftarrow dc$
5:          $ad \leftarrow curr.adesc$
6:          if $ad \neq$ NIL and $dp \in [ad.dp,\ ad.dc]$ then
7:             FINISHINSERTING($curr$, $ad$)
8:          $curr \leftarrow$ CLEARMARK($curr.child[dc], F_{all}$)
9:       if $curr =$ NIL or $k[dc] < curr.k[dc]$ then
10:         break
11:      else
12:         $dc \leftarrow dc + 1$
```

In the case that the node with the target coordinates already exists, $curr$ points to the node itself. Together with the dimension variables dp and dc, they amount to the context for inserting or deleting nodes. On line 4.7, prior to reading $curr$'s child LOCATEPRED helps finish any child adoption tasks in order to acquire the up-to-date values. Since a child adoption process updates the children indexed from dp to dc, the function must help $curr$ node if it intends to read the child node in this range (line 4.6). The helping task does not alter already traversed nodes, so the search process can continue without restart.

B. Concurrent Insert

The lock-free INSERT operation is derived from the sequential MDList algorithm [11], in which each insertion takes one or two steps and updates no more than two consecutive nodes. Figure 3 illustrates the insertion of key 32 (coordinates (2,0,0)). In the first step, we update the predecessor node: the new node is spliced with its predecessor on dimension 0 (as marked the by green arrows), and the old child of the predecessor becomes a dimension 2 child of the new node. In the second step, we update the successor node if its dimensionality has changed during the first step. In this case, the new node adopts child nodes of its successor between dimension $[0, 2)$ (as marked by the red arrows). We guarantee lock-free progress in the node splicing step by atomically updating predecessor's child pointer using CAS [17]. To provide for lock-free progress in the child

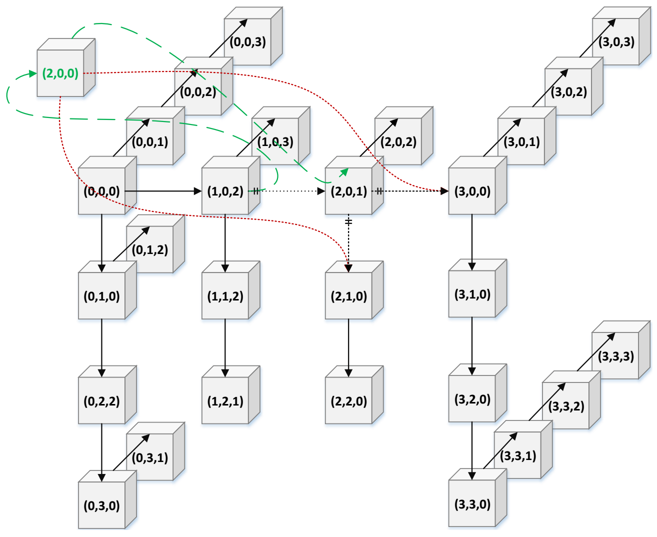


Figure 3: INSERT operation in a 3DList ($D = 3, N = 64$)

adoption step, we need to announce the operation globally using descriptor objects [16]. This allows the interrupting threads to help finish the operation in case the insertion thread is preempted.

We list the concurrent INSERT function in Algorithm 5. After locating the target position (line 5.12), the function updates the child pointer of the predecessor node (line 5.21). The dimension of the node being inserted is kept in dp and the dimension of the child in dc (line 4.4 and 4.12). The new node should be inserted at the dimension dp child of the $pred$ node, while a non-empty $curr$ node will become the dimension dc child of the new node. The code between lines 5.13 and 5.15 reads the $adesc$ fields from $curr$ and tests if helping is needed. Like in LOCATEPRED, the insertion need to help $curr$ node if it is going to adopt children from $curr$ node.

Prior to atomically updating the link to the new node, we fill the remaining fields of the new node (line 5.27 and 5.35). If the new node needs to adopt children from $curr$ node, we use an adopt descriptor to store the necessary context (line 5.30). The pointers within the range $[0, dp)$ of the new node's child array are marked with F_{adp}. This effectively invalidates these positions for future insertions. The pointers within the range $[dp, D]$ are set to NIL meaning they are available for attaching child nodes. $curr$ node is conditionally linked to the new node on line 5.34. dc can be set to D on either line 5.19 or line 4.12. In the first case, $curr$ must be logically deleted, and the new node is immediately in front of $curr$. By not linking it with the new node, we effectively discard $curr$. In the second case, the new node has the same key as $curr$ and we essentially update the associated value by replacing $curr$ with the new node. On line 5.21, the single-word CAS atomic synchronization primitive is used to update the $pred$'s child pointer. The CAS would fail under three circumstances: 1) the desired child

Algorithm 5 Concurrent Insert

```
1: function INSERT(int $key$, void* $val$)
2:     Node* $node$                                   ▷ the new node
3:     Node* $pred$, $curr$          ▷ new node's parent and child
4:     int $dp$, $dc$         ▷ dimension of node in pred and curr
5:     AdoptDesc* $ad$  ▷ descriptor for child adoption task
6:     $node \leftarrow$ new Node
7:     $node.key \leftarrow key$, $node.val \leftarrow val$
8:     $node.k[0:D] \leftarrow$ KEYTOCOORD($key$)$[0:D]$
9:     $node.child[0:D] \leftarrow$ NIL
10:    while true do
11:        $pred \leftarrow$ NIL, $curr \leftarrow head$, $dp \leftarrow 0$, $dc \leftarrow 0$
12:        LOCATEPRED($node.k$)
13:        $ad \leftarrow curr \neq$ NIL ? $curr.adesc$ : NIL
14:        if $ad \neq$ NIL and $dp \neq dc$ then
15:            FINISHINSERTING($curr$, $ad$)
16:        if ISMARKED($pred.child[dp], F_{del}$) then
17:            $curr \leftarrow$ SETMARK($curr, F_{del}$)
18:            if $dc = D - 1$ then
19:                $dc \leftarrow D$
20:        FILLNEWNODE( )
21:        if CAS(&$pred.child[dp]$, $curr$, $node$) then
22:            if $ad \neq$ NIL then
23:                FINISHINSERTING($node$, $ad$)
24:            break
25:
26: inline function FILLNEWNODE( )
27:    $ad \leftarrow$ NIL
28:    if $dp \neq dc$ then
29:        $ad \leftarrow$ new AdoptDesc
30:        $ad.curr \leftarrow curr$, $ad.dp \leftarrow dp$, $ad.dc \leftarrow dc$
31:    $node.child[0:dp] \leftarrow F_{adp}$
32:    $node.child[dp:D] \leftarrow$ NIL
33:    if $dc < D$ then
34:        $node.child[dc] \leftarrow curr$
35:    $node.adesc \leftarrow ad$
```

slot has been updated by another competing insertion; 2) the desired child slot has been invalidated by a child adoption process; and 3) the desired child slot has been marked for logical deletion. If any of the above cases is true, the loop restarts. Otherwise, the insertion proceeds to finish its own child adoption process.

The FINISHINSERTING function in Algorithm 6 performs child adoption on a given node n with the descriptor ad. This is a helping procedure that must correctly handle duplicate and simultaneous executions. The function first reads the adoption context from the descriptor into its local variables. It then transfers $curr$ node's children within the range of $[dp, dc)$ to n. Before a child pointer can be copied, we must safeguard it so that it cannot be changed while the copy is in progress. This is done by setting the F_{adp} flag in the child pointers (line 6.5). Once the flag is set, the function

Algorithm 6 Child Adoption

```
1: function FINISHINSERTING(Node* $n$, AdoptDesc* $ad$)
2:     Node* $child$, $curr \leftarrow ad.curr$
3:     int $dp \leftarrow ad.dp$, $dc \leftarrow ad.dc$
4:     for $i \in [dp, dc)$ do
5:         $child \leftarrow$ FETCHANDOR(&$curr.child[i]$, $F_{adp}$)
6:         $child \leftarrow$ CLEARMARK($child$, $F_{adp}$)
7:         if $n.child[i] =$ NIL then
8:             CAS(&$n.child[i]$, NIL, $child$)
9:     CAS(&$n.adesc$, $ad$, NIL)
```

Algorithm 7 Concurrent Delete

```
1: function DELETE(int $key$)
2:     Node* $curr$, $pred$, $child$, $marked$
3:     int $dp$, $dc$
4:     while true do
5:         $pred \leftarrow$ NIL, $curr \leftarrow head$, $dp \leftarrow 0$, $dc \leftarrow 0$
6:         LOCATEPRED(KEYTOCOORD($key$))
7:         if $dc \neq D$ then
8:             return NIL
9:         $marked \leftarrow$ SETMARK($curr$, $F_{del}$)
10:        $child \leftarrow$ CAS$_{val}$(&$pred.child[dp]$, $curr$, $marked$)

11:        if CLEARMARK($child$, $F_{del}|F_{adp}$) $= curr$ then
12:            if !ISMARKED($child$, $F_{del}|F_{adp}$) then
13:                return $curr.val$
14:            else if ISMARKED($child$, $F_{del}$) then
15:                return NIL
```

proceeds to copy the pointer to n (line 6.8). Finally, the descriptor field in n is cleared to designate the operation's completion.

C. Concurrent Delete

The sequential DELETE and INSERT operations on an MDList [11] works reciprocally: the former may promote a node to a lower dimension while the latter may demote a node to a higher dimension. This works well for sequential algorithms, but in concurrent execution where threads help each other, bidirectional change of nodes' dimension incurs contention and synchronization issues. Consider a node n with an active child adoption descriptor (i.e., $n.adesc \neq$ NIL). When concurrency level is high, several threads may read this descriptor and proceed to help finish the adoption by marking some children on node $n.adesc.curr$ as invalid. One of them will eventually succeeds in finish the helping (as observed by setting $n.adesc \leftarrow$ NIL), but we have no way to know *if all threads have finished the helping*. If a DELETE operation promotes the node $n.adesc.curr$ by re-enabling the invalid child pointers, an unfinished helping process may erroneously disable them again. Additional synchronization is required to prevent threads from interfering with each other when they perform helping task on the same node. We found the simplest solution is to *keep dimensionality change unidirectional.*

Our lock-free DELETE operation is thus asymmetrical in the sense that it dose not remove any node from the data structure nor alter the nodes' dimensionality. It only marks a node for logical deletion [17], and leaves the execution of physical removal to subsequent INSERT operations. When a new node (n_n) is inserted immediately before a logically deleted node (n_d), n_n expunge n_d from the data structure by skipping n_n and linking directly into all of the child nodes of n_d. Since the physical deletion is embedded in the insertion operation, we reduce the interaction between insertion and deletion operations to a minimal and achieved an overall simple design of lock-free dictionary. This may sound counterintuitive, but the list-like partition strategy of MDList allow us to efficiently discard nodes by simply skipping links. Since a logically deleted node only gets purged when an insertion take place immediately in front of it, there will be a number of zombie nodes. We thus trade memory consumption for scalability. We thus trade memory consumption for scalability.

In Algorithm 7, the concurrent DELETE operation traverses the dictionary starting from the head looking for target node. It shares the same CAS-based loop as the INSERT function. The process terminates on line 7.8 if it fails to find the target node. Otherwise, it marks the target node for logical deletion using CAS 7.10. A node is considered logically deleted once the pointer in its parent's child array is marked with F_{del}. The CAS$_{val}$ returns the value store on the address before the update. It is considered successful if the return value $child$ is equal to the expected value $curr$, which is detected by the conditional statement on line 7.11 and 7.12. Otherwise, the function checks if the target node has already been marked for deletion by examine the F_{del} flag on $child$ (line 7.14). If so, the function returns. Finally, the child pointer in $pred$ must have been updated by concurrent insertions, DELETE would start anew from the head.

IV. CORRECTNESS

In this section, we sketch a proof of the main correctness property of the presented dictionary algorithm, which is linearizability [18]. A concurrent object is linearizable if all of its operations appear to take effect instantaneously at some point after the operations start and before they end. We begin by defining the *abstract state* of a sequential dictionary and then show how to map the internal state of our concrete dictionary object to the abstract state. We denote the abstract state of a sequential dictionary to be a totally ordered set P. Equation 1 specifies that an INSERT operation grows the set if the key being inserted does not exist. Equation 2 specifies that a DELETE operation shrinks a non-empty set by removing the key-value pair with the specific key.

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

$$\textsc{Delete}(k) = \begin{cases} P \setminus \{\langle k,v\rangle\} & \langle k,v\rangle \in P \\ P & \langle k,v\rangle \notin P \end{cases} \tag{2}$$

A. Invariants

Now we consider the concurrent dictionary object. By a *node*, we refer to an object of type **Node** that has been allocated and successfully linked to an existing node (line 5.21). We denote the set of nodes that are reachable from $head$ by L. The following invariants are satisfied by the concrete dictionary object at all times. 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. $\forall n, n' \in L, \textsc{ClearMark}(n.child[d], F_{adp} \mid F_{del}) = n' \wedge n.adesc = NIL \implies \forall i \in [0, d), \textsc{IsMarked}(n'.child\ [i], F_{adp}) =$ ***true***

Proof: By observing the statements at line 5.31 and 6.5 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. ■

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

Invariant 2. $\forall n \in L, \exists p = \{d_0, d_1, ..., d_m\} : d_0 \leq d_1 \leq ... \leq d_m \ \wedge \ head.child[d_0].child[d_1]...child.[d_m] = n$

Proof: At the start, the structure contains a 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 (Algorithm 4) form a consecutive path p'. Note the condition checks in (line 4.3 and 4.9), we have $i < j \implies d_i \leq d_j\ \forall d_i, d_j \in p'$. Though subsequent insertions may alter the path, they 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'. ■

Lemma 1. *At any time, nodes in L, 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. Follow Invariant 2, logically deleted key-value pairs still occupy valid nodes in the structure before they are physically removed. ■

We now show that the nodes *without* deletion marks form a well-ordered set that is equivalent to P. Invariant 3 states that the ordering property described by Definition 2 is kept at all times.

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

Proof: Initially the invariants trivially holds. The linkage among nodes is only changed by insertion, and child adoption. Insert preserves the invariants because the condition checks on line 5.3 and 5.9 guarantee that $\forall i \in [0, dp)\ pred.k[i] = node.k[i] \land pred.k[dp] < node.k[dp]$. Child adoption preserves the invariant because $\forall i \in [dp, dc)\ node.k[i] = curr.k[i] < curr.child[i].k[i]$. ■

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

Proof: Note that a pointer to logically deleted nodes is marked by flag F_{del}, which renders the key-value pair stored in that node obsolete. However, the node's location withing the data structure is still consistent with its embed coordinates, making it a valid routing node. The traversing operation treat logically deleted nodes transparently by explicitly clearing the pointer markings on line 4.8. ■

Let us define the set of logically deleted nodes by $S = \{n | n' \in L \land n'.child[d] = \text{SETMARK}(n, F_{del})\}$. Following Lemma 1 and 2, the abstract state can then be defined as $P \equiv L \setminus S$.

B. Linearizability

We now sketch a proof that our algorithm is a linearizable dictionary implementation that complies with the abstract semantics by identifying *linearization points* for each operation. The concurrent operation can be viewed as it occurred atomically at its linearization point in the execution history. Additionally, we use the notion of decision points and state-read points to facilitate our reasoning [9]. The decision point of an operation is define as the atomic statement that finitely decides the result of an operation, i.e. independent of the result of any subsequent instruction after that point. A state-read point is define as the atomic statement where the state of the dictionary, which determines the outcome of the decision point, is read.

Theorem 1. *A* FIND(k) *operation takes effect atomically at one statement.*

Proof: A find operation may return v if $\langle k, v\rangle \in P$, or NIL otherwise. The decision point for the first case is when the `while` loop terminates on line 4.2. The node with k must exist in P because the coordinates up to dimension D have been exhaustively examined. The state-read point is line 4.8 when $curr$ is read from child pointers. The subsequent execution branches to line 4.12 after comparing $curr.k$ with k. As the coordinate field k cannot be changed after initialization, the state of the dictionary immediate before passing the state-read point must have been $\langle k, v\rangle \in P$. The decision point for the second case is when the condition check on line 4.9 fails. The state-read point is when the value of $curr$ is read on line 4.8. For both case, the linearization point is the state-read point on line 4.8 ■

Theorem 2. *An* INSERT$(\langle k, v\rangle)$ *operation takes effect atomically at one statement.*

Proof: An INSERT operation returns on line 5.24; given a legal key it must succeed by either adding a new node or replacing an existing node. The decision point for both cases to take effect is when the CAS operation on line 5.21 succeeds. The remaining atomic primitives in the child adoption process will be executed at least once and successfully complete through the use of helping mechanism. Equation 1 holds for the first case because $L = L \cup \langle k, v\rangle$. It holds for the second case because $L = L \cup \langle k, v\rangle \setminus \langle k, v'\rangle$. ■

Theorem 3. *A* DELETE(k) *operation takes effect atomically at one statement.*

Proof: If $\langle k, v\rangle \in P$, a successful DELETE$(k)$ operation updates the abstract state by growing S. The decision point for it to take effect is when the CAS operation on line 7.10 successfully marks a node for deletion. Equations 2 holds because $S' = S \cup \langle k, v\rangle \implies P' = P \setminus \{\langle k, v\rangle\}$. The decision points for a DELETE(k) operation to fail are on line 7.7 when it cannot find the node with the target key, and on line 7.14 when the target node has been logically deleted by a competing deletion operation. The state-read point for the former is on line 4.8, which causes LOCATEPRED to abort before reaching the highest dimension. The state-read point for the latter is on line 7.10, where $child$ is read. The linearization points for failed deletions are either of the state-read points. In these cases, Equations 2 holds because $S' = S \implies P' = P$. ■

C. 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 3. FINISHINSERTING *(Algorithm 6) and* LOCATEPRED *(Algorithm 4) complete in finite steps.*

Proof: We observe that there is no unbounded loop in Algorithm 6. The `for` loop on line 6.4 is bounded by the dimensionality of data structure D, which in practice is a small number. For LOCATEPRED, the `while` loop (line 4.2) is also bounded by D. The inner unbounded `while` loop (4.3) ends after at most $\sqrt[D]{N}$ retries, which is the maximum number of nodes in each dimension. The maximum number of nodes to be examined by LOCATEDPRED is thus $D \cdot \sqrt[D]{N}$. ∎

Theorem 4. FIND *operations are wait-free.*

Proof: The FIND operations invoke LOCATEPRED and does not contain additional loops. The theorem holds by following Lemma 3 ∎

Theorem 5. 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 (line 5.21, and 7.10 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. ∎

V. EXPERIMENTAL EVALUATION

We compare the performance of our algorithm (MDLIST) to the following concurrent skiplists and BSTs that are available to us in C/C++ implementations.

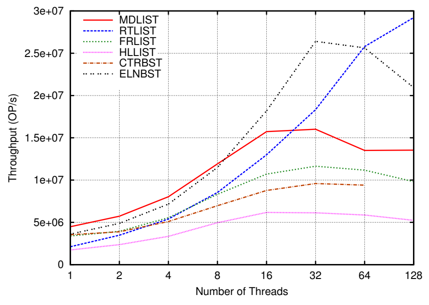
1) The rotating skiplist (RTLIST) by Dick et al. [14]. Their implementation is build upon the C port of Crain's no hot-spot skiplist [19], which employs a background thread to maintain balance and handle physical deletions.
2) Herlihy's lock-free skiplist [16] (HLLIST) with lazy deletions. The tested version is derived from Wicht's C++ implementation [20].
3) Fraser's publicly available lock-free skiplist [10] (FRLIST), which includes several optimizations not found in HLLIST. It is often considered as the most efficient skiplist implementation.
4) The lock-based implementation of Bronson et al.'s relaxed AVL tree [6] (BRNBST) derived from Wicht's C++ implementation [20].
5) The lock-free unbalanced BST (ELNBST) based on Ellen et al's algorithm [21]. The implementation is derived from Wicht's C++ implementation [20].
6) The Citrus tree [15] (CTRBST) by Arbel et al, which employs the novel synchronization mechanism *read copy update* (RCU). RCU allows updates to produce a copy of the data they write so the read-only accesses can proceed without acquiring locks. Code is available directly from the authors.

The original implementation of FRLIST, RTLIST and our algorithm used the epoch based garbage collection proposed by Fraser [10], while BRNBST, ELNBST and HLLIST used hazard pointers [22]. CTRBST, on the other hand, did not employ any memory management scheme. For fair comparison of the algorithms themselves, we disabled memory reclamation for all approaches but use thread-caching malloc [3] as a scalable alternative to the standard library malloc. We employ a micro-benchmark to evaluate the throughput of these approaches for uniformly distributed keys. This canonical evaluation method [17], [14], [3] consists of a tight loop that randomly chooses to perform an INSERT, a DELETE or a FIND operation in each iteration. Each thread performs one million operations and we take the average from four runs. As done in [3], [2] we pre-populate the data structures to half capacity to ensure consistent result. Both the micro-benchmark and the dictionary implementations are compiled using GCC 4.7 with `O3` optimizations. The tests are conducted on a 64-core NUMA system (4 AMD opteron CPUs with 16 cores per chip @2.1 GHz) and a 12-core SMP system (1 Intel Xeon 6-core CPU with hyper-threading @2.9GHz).

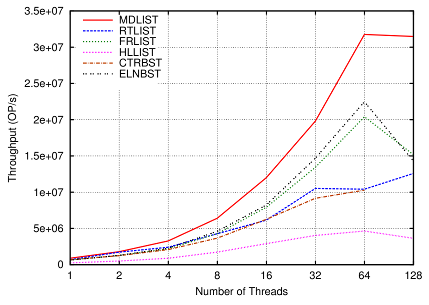
Figures 4, 5 and 6 illustrate the algorithms' throughput on the NUMA system. The y-axis represents the throughput measured by *operation per second*, and the x-axis represents the number of threads in logarithmic scale. Like the evaluations in [3], [23], we consider three different workload distributions: a) write-dominated with 50% insertion, 50% deletion; b) mixed workload with 20% insertion, 10% deletion and 70% find; c) read-dominated with 9% insertion, 1% deletion and 90% find. We also consider three ranges of keys, 1000 (1K), 1 million (1M) and 1 billion (1G). The size of key space affects the height of search trees, but it does not affect the tower height of a skiplist nor the dimension of an MDList as those are chosen by users prior to execution.

Figures 4a, 4b and 4c depicted the write-dominated situation. We observe that the both skiplist-based and BST-based dictionaries were able to explore fine-grained parallelism and exhibit similar scalability trends. The overall throughput increases almost 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. For small key space of 1K keys (Figure 4a, RCU-based Citrus trees stand out against the rest. This is because most insertions would not update the data structures due to the existence duplicated keys, and they essentially become read-only operations for which RCU is optimized. MDLIST has slightly larger overhead for traversing operations if it is under-populated. To locate an existing node, the prefix matching algorithm always needs to perform D comparison operations, whereas in a BST, the search algorithm could terminate much earlier because of shallow depth. For 1M

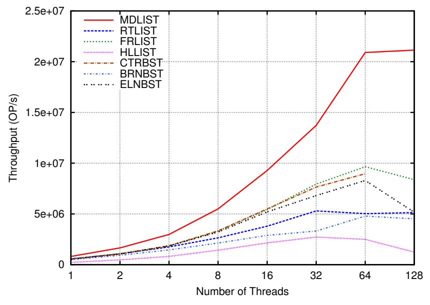
[3]http://goog-perftools.sourceforge.net/doc/tcmalloc.html



(a) 1K keys

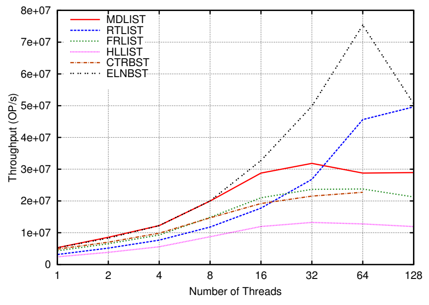


(b) 1M keys

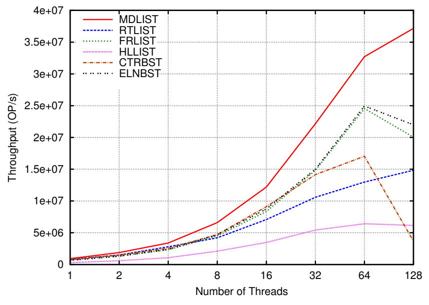


(c) 1G keys

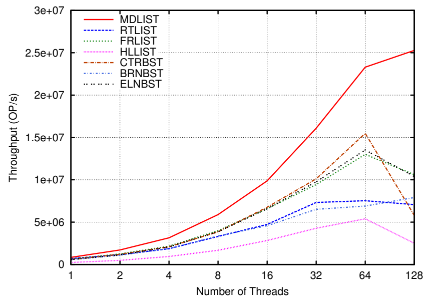
Figure 4: 50% INSERT, 50% DELETE, 0% FIND on the NUMA System



(a) 1K keys

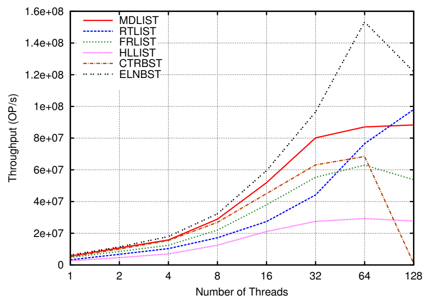


(b) 1M keys

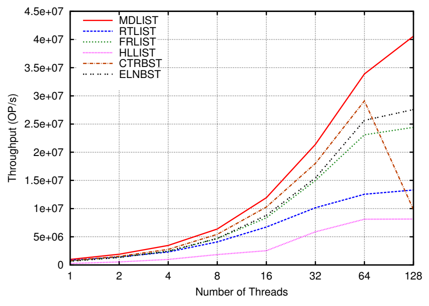


(c) 1G keys

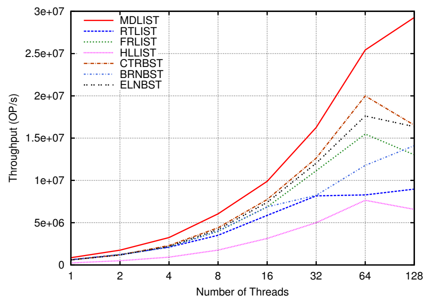
Figure 5: 20% INSERT, 10% DELETE, 70% FIND on the NUMA System



(a) 1K keys



(b) 1M keys

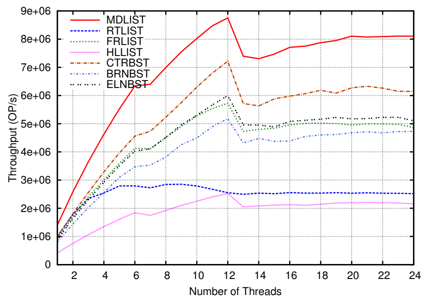


(c) 1G keys

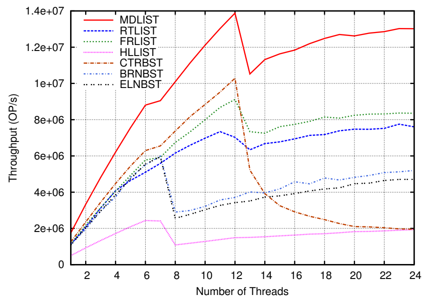
Figure 6: 9% INSERT, 1% DELETE, 90% FIND on the NUMA System

key space (Figure 4b, most approaches achieve similar throughput except for HLLIST. MDLIST outperform the other approaches by an average of 25%. It continues to excel in large key space with one billion keys. As shown by Figure 4c, MDLIST outperforms the best alternatives including FRLIST and CTRBST by as much as 100%. For larger key space, insertions are less likely to collide with an existing key. The size of the data structures grows much quicker too. Each insertion in MDLIST modifies at most two consecutive nodes, incurring less remote memory access than the skiplist-based and BST-based approaches.

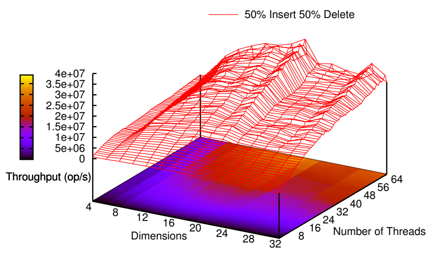
Figures 5a, 5b and 5c show the throughput for the mixed workload. The overall scalability trends for all three key ranges mimic those of the write-dominated workload respectively. In key spaces with 1M to 1G keys, MDLIST achieves 15% 30% speedup over FRLIST and ELNBST, which are two of the best skiplist-based and BST-based algorithms in mixed workload. Figures 6a, 6b and 6c show the throughput for the read-dominated workload. As the distribution of mutating operations further decreases to one tenth of the whole operations, the performance gaps among different algorithms begin to diminish. For 1G keys (Figure 6c, RTLIST, FRLIST, CTRBST, and ELNBST have almost identical performance up until 8 threads. This is because all the data structures being tested implement logarithmic search. Less writes means less interference between concurrent updates and traverse. The performance characteristics of difference algorithms thus converge towards an ideal straight line. They differ only in term of scalability at high levels of concurrency. CTRBST achieves the best scalability among the alternatives but is still 20% slower than MDLIST. Note that in Figure 6a CTRBST's performance degrades



(a) 4G keys, 100% INSERT



(b) 4M keys, 50% updates



(c) Dimension sweep

Figure 7: Throughput on SMP system and dimension sweep on NUMA

drastically when the benchmark spawn more threads than the number of available hardware cores. CTRBST employs a user level RCU library that uses global locks, which may delay running threads if the threads holding the locks get preempted.

Figure 7a and 7b show the throughput of the algorithms on the SMP system. The x-axis in these graphs is in linear scale. In Figure 7a, the executions consist solely of INSERT operations, which insert keys in the range of 32-bit integers. For all approaches, the overall system throughput peaks at 12 threads which is the maximum number of hardware threads. Executions beyond 12 threads are preemptive, and the throughput slightly dropped due to unbalanced load causing increasing amount of cache invalidation. MDLIST provides an average of 30% speedup over CTRBST on all levels of concurrency. While the performance of the BST-based approaches closely resembles each other, the performance of skiplist-based approaches varies. Notably, the throughput of RTLIST drops significantly beyond 6 threads. The impact of a dedicated maintenance is clearly visible on the SMP system, where the background thread has to compete with working threads for limited hardware cores. Because the background thread is the only thread that does physical deletion, the overall progress will stagnate once the it is suspended. In Figure 7b, we distribute the workload by having 30% insertions, 20% deletions and 50% searches. MDLIST outperforms FRLIST by as much as 60%. The throughput of CTRBST again drops drastically after exhausting all hardware threads.

In Figure 7c, we sweep the dimension of MDLIST from 4 to 32 on the NUMA system, and show that the algorithm achieves maximum throughput with 20 dimensions on 64 threads. On all scale levels, we see that the throughput converges towards 20 dimensions. This means that the way the dimensionality of an MDLIST affects its performance is independent from the number of threads. The performance of MDLIST can be optimized if the access pattern of the user application is taken into account.

Overall, MDLIST excels at high levels of concurrency with large key spaces. 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 concurrency, MDLIST performs equally well or even better than the state of the art skiplist-based and BST-based approaches.

VI. RELATED WORK

Concurrent key-value store data structures that implement abstract dictionaries have been extensively studied in the literature. Unordered dictionaries can be built upon non-blocking hash tables [24], which achieve $\mathcal{O}(1)$ amortized cost. Range queries are not attainable on such data structures because keys are not totally ordered. We thus focus our discussion on BSTs and skiplists, which provide totally ordered key-value store and logarithmic search time by retaining balance either explicitly or probabilistically.

A. *Search Trees*

Recent fine-grained locking implementations by Bronson et al. [6] and Afek et al. [25] take advantage of optimistic concurrently control and hand over hand validation to reduce synchronization overhead. The lock-based relaxed AVL tree by Crain et al. [7] further reduces contention by delaying tree rotation and employing a dedicated maintenance thread to remove logically deleted nodes. Drachsler et al. [5] proposed a relaxed balanced and an unbalanced lock-based BSTs in which the nodes store additional logical ordering data by pointing to its logical predecessor and successor. The search operation would traverse the logical chain to locate the target if it does not find the target after reaching a physical leaf node. In comparison, the proposed MDList has deterministic physical layout pertaining to the logical ordering, which allows for more straightforward and efficient search operations.

Fraser [10] presented a non-blocking BST using multi-word CAS, which is not available on existing multi-core chips and expensive to implement using CAS. The first practical lock-free linearizable BST design was given by Ellen et al. [21]. Their implementation was based on a leaf-oriented BST, where values are stored externally in leaf nodes and internal nodes were only used for routing. Howley and Jones [23] presented a lock-free node-oriented BST based on the same co-operative technique. Their algorithm has faster search operations than those in Ellen et al.'s algorithm because the search path is generally shorter in a node-oriented BST than in a leaf-oriented one. On the other hand, Ellen et al.'s DELETE operations, which avoid removing nodes with two children, are simpler and faster at the price of extra memory for internal nodes. Natarajan and Mittal [3] proposed a lock-free leaf-oriented BST, which marks edges instead of nodes. Their algorithm is more efficient than previous approaches because the mutating operations work on a smaller portion of the tree and execute fewer atomic instructions. Due to the complexity of rebalancing mentioned in Section I, all of the above non-blocking trees are not balanced, thus they subject to potential linear runtime for certain inputs. Brown et al. [2] proposed a general template for implementing lock-free linearizable down-trees. In order to orchestrate rebalancing among threads, their tree update routine atomically replaces a subgraph with a new connected subgraph using a set of multi-word synchronization primitives. As recognized by the above researches, non-blocking rebalancing presents the biggest challenge to practical non-blocking BSTs. The proposed MDList eliminates the need of balancing resulting in simpler and more efficient synchronization mechanisms.

B. Skiplists

Pugh [26] designed a concurrent skiplist with per-pointer locks, where an update to a pointer must be protected by a lock. Shavit [27] discovered that the highly decentralized skiplist is suitable for shared-memory systems and presents a concurrent priority queue based on Pugh's algorithm. In practice, probabilistic balancing is easier to implement and as efficient as deterministic balancing. This is why the concurrent map class of the Java standard library uses skiplists rather than search trees.Fraser [10] proposed a lock-free skiplist with an epoch based memory reclamation technique. It also employs the logical deletion technique, which was originally proposed by Harris [17], to mark pointers and delay physical removal. Sundell and Tsigas [9] presented a provably correct linearizable lock-free skiplist, which uses exponential back-off to alleviate contention. They guarantee linearizability by forcing threads to help physically remove a node before moving past it. Herlihy et al. [28] proposed an optimistic concurrent skiplist that simplifies previous work and allows for easy proof of correctness while maintaining comparable performance. Crain et al. [19] proposed a no hot spot skiplist that alleviates contention by localizing synchronization at the least contended part of the structure. Dick et al. [14] recently presented an improvement over existing skiplists algorithms. Their lock-free skiplist uses rotating wheels instead of the usual towers to improve locality of reference and speedup traversals. Skiplists have also been used to built a number of hybrid data structures. Spiegel et al. [29] combined a skiplist and a B-tree to produce a lock-free multi-way search tree, improving spatial locality of reference of skiplists by storing several elements in a single node.

C. Tries

Prokopec et al. [12] proposed a lock-free hash array mapped trie using both single-word and double-word CAS. They introduce intermediate nodes to solve the synchronization issue of updating the branching nodes. However, the intermediate nodes also become hot spots for contention because every expansion of a branching node requires a new branching node to be linked to the corresponding intermediate node through CAS. Oshman and Shavit [13] proposed the lock-free skiptrie using double-word CAS, which combines a shallow skiplist with a x-fast trie to store high level nodes. The MDList also bears some similarity to above mentioned tries in that the keys are ordered by their prefixes: a node always shares the same key prefix with its parent nodes. The major difference lies in the partition strategies: in a trie a node shares the same prefix with all of its children, but in an MDList a node shares prefixes of different lengths with each of its child. This leads to a constant branch factor for nodes in tries and reducing branching factors for bottom levels nodes in MDLists. Besides, retrieving the minimal key in a trie requires repetitive search, while MDList behaves like a heap where the root node always holds the smallest key. Memory wise, the values are stored in a trie's leaf nodes, whereas they are stored in an MDList's internal nodes.

VII. CONCLUSION

In this paper, we presented a simple and efficient lock-free dictionary design based on MDList. It maps keys into high dimensional vector coordinates and achieve deterministic layout that is consistent with nodes' logical ordering. We exploited spatial locality to increase the throughput of the INSERT operations, and adopted asymmetrical logical deletion to address the synchronization overhead of the DELETE operations. When compared to the best available skiplist-based and BST-based algorithms, our algorithm achieved performance gains in scenarios with medium to large key spaces. As much as 100% performance improvement is achieved on a 64-core NUMA system, and an average of 40% on a 12-core SMP system. The performance of our dictionary can be tailored to different access patterns by changing its dimensionality. We plan to improve the functionally of the dictionary by implementing concurrent range queries and iterator operations.

REFERENCES

[1] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[2] T. Brown, F. Ellen, and E. Ruppert, “A general technique for non-blocking trees,” in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014, pp. 329–342.

[3] A. Natarajan and N. Mittal, “Fast concurrent lock-free binary search trees,” in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014, pp. 317–328.

[4] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert, “The amortized complexity of non-blocking binary search trees,” in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 332–340.

[5] D. Drachsler, M. Vechev, and E. Yahav, “Practical concurrent binary search trees via logical ordering,” in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014, pp. 343–356.

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, “A practical concurrent binary search tree,” in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 257–268.

[7] T. Crain, V. Gramoli, and M. Raynal, “A contention-friendly binary search tree,” in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 229–240.

[8] J. Lindén and B. Jonsson, “A skiplist-based concurrent priority queue with minimal memory contention,” in *Principles of Distributed Systems*. Springer, 2013, pp. 206–220.

[9] H. Sundell and P. Tsigas, “Scalable and lock-free concurrent dictionaries,” in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004, pp. 1438–1445.

[10] K. Fraser, “Practical lock-freedom,” Ph.D. dissertation, Cambridge University Computer Laboratory, 2004.

[11] 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, March 2016.

[12] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, “Concurrent tries with efficient non-blocking snapshots,” in *Acm Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.

[13] R. Oshman and N. Shavit, “The skiptrie: low-depth concurrent search without rebalancing,” in *Proceedings of the 2013 ACM symposium on Principles of distributed computing*. ACM, 2013, pp. 23–32.

[14] I. Dick, A. Fekete, and V. Gramoli, “Logarithmic data structures for multicores,” 2014.

[15] M. Arbel and H. Attiya, “Concurrent updates with rcu: Search tree as an example,” in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC ’14. New York, NY, USA: ACM, 2014, pp. 196–205. [Online]. Available: http://doi.acm.org/10.1145/2611462.2611471

[16] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[17] T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *Distributed Computing*. Springer, 2001, pp. 300–314.

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

[19] T. Crain, V. Gramoli, and M. Raynal, “No hot spot non-blocking skip list,” in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 196–205.

[20] B. Wicht and C. Evequoz, “Binary trees implementations comparison for multicore programming,” Information and Communications Technology, University of applied sciences in Fribourg, Tech. Rep., 2012.

[21] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, “Non-blocking binary search trees,” in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 131–140.

[22] M. M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 6, pp. 491–504, 2004.

[23] S. V. Howley and J. Jones, “A non-blocking internal binary search tree,” in *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 161–171.

[24] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2002, pp. 73–82.

[25] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, “Cbtree: A practical concurrent self-adjusting search tree,” in *Distributed Computing*. Springer, 2012, pp. 1–15.

[26] W. Pugh, “Concurrent maintenance of skip lists,” Department of Computer Science, University of Maryland, Tech. Rep. CS-TR-2222.1, 1990.

[27] N. Shavit and A. Zemach, “Scalable concurrent priority queue algorithms,” in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, 1999, pp. 113–122.

[28] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, “A provably correct scalable concurrent skip list,” in *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[29] M. Spiegel and P. Reynolds, “Lock-free multiway search trees,” in *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 2010, pp. 604–613.