Lock-Free Transactional Adjacency List

Zachary Painter(✉), Christina Peterson, and Damian Dechev

University Of Central Florida, Orlando, FL 32816, USA
zacharypainter@knights.ucf.edu

Abstract. Adjacency lists are frequently used in graphing or map based applications. Although efficient concurrent linked-list algorithms are well known, it can be difficult to adapt these approaches to build a high-performance adjacency list. Furthermore, it can often be desirable to execute operations in these data structures transactionally, or perform a sequence of operations in one atomic step. In this paper, we present a lock-free transactional adjacency list based on a multi-dimensional list (MDList). We are able to combine known linked list strategies with the capability of the MDList in order to efficiently organize graph vertexes and their edges. We design our underlying data structure to be node-based and linearizable, then use the Lock-Free Transactional Transformation (LFTT) methodology to efficiently enable transactional execution. In our performance evaluation, our lock-free transactional adjacency list achieves an average of 50% speedup over a transactional boosting implementation.

1 Introduction

Lock-free data structures aim to fully utilize the computing resources of multi-core processors without the drawbacks of lock-based counterparts such as deadlock or priority inversion. However, lock-free data structures are difficult to design due to the consideration of all possible thread interleavings when reasoning about safety or liveness properties. Even more so are lock-free transactional data structures because in addition to the safety and liveness properties of traditional lock-free data structures, isolation must be preserved such that a series of operations appear to occur in one atomic step.

An adjacency list data structure maps graph nodes, or "vertexes," to other nodes by their connections, or "edges." Generally, if a vertex i is adjacent to another vertex j, then vertex j is contained in the sublist of vertex i. In order to implement such a data structure concurrently, one would need to overcome the challenges of traversing in multiple dimensions, organizing vertex and edge nodes, and properly disposing of all children of a vertex before deleting the vertex.

This research was supported by the National Science Foundation under NSF OAC 1440530, NSF CCF 1717515, and NSF OAC 1740095.

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 203–219, 2019.
https://doi.org/10.1007/978-3-030-35225-7_14

Previous work on lock-free linked list data structures are designed for sets and queues. Since elements of these abstract data types do not account for relationships between elements, they are unsuitable to be directly used for an adjacency list data structure. An adjacency list data structure needs to support operations that can insert and remove vertexes and edges, as well as check whether a vertex or edge is contained in the list. Additional synchronization is required to ensure that an operation that deletes a vertex i does not modify or remove nodes that are currently part of i's sublist of adjacent nodes. Further synchronization is required to ensure that two operations are able to simultaneously modify the sublist of a vertex despite those operations appearing to take place at the same vertex.

A lock-free adjacency list provides atomicity at the granularity of an individual operation. However, in some cases one may want to perform a sequence of operations such that the entire sequence appears to take place in one atomic step. One such case is during the deletion of a vertex, in which case it must first be guaranteed that all edges from that vertex have already been deleted. In such a case, a sequence of operations such as the following would be useful.

```
1: if isEmpty(vertex.List) then
2:     Delete(vertex);
```

This code should be able to verify that a given node's sublist is empty before deleting that node. Unfortunately, this operation fails to complete its goal. Since the composition of the methods is not atomic, another thread a could insert an edge between the time that thread b reads that the list of edge nodes is empty, and thread b deleting the vertex, thus invalidating the operation.

In order to perform a series of operations such as those previously mentioned, all involved operations need to appear to take place in a single atomic step. Additionally, if any operation fails, it must appear as though none of the operations took place. Some implementations, such as Transactional Boosting [9], use fine-grained locking in order to create a transactional data structure from an underlying concurrent data structure. This, however, reduces the performance of the data structure, and negates any lock-free progress guarantee the underlying data structure might have had. Software Transactional Memory (STM) can also be used to create transactional data structures from existing ones. Unfortunately, this approach also creates significant performance loss. In an STM data structure, transactions maintain a list of read and write locations. If a transaction's read and write set overlaps with another transaction's write set, those transactions conflict. In the case of a conflict, one of the transactions must abort. This results in a significant amount of unnecessary aborts, as conflicts detected in this way do not necessarily correspond to high-level semantic conflicts. These excessive aborts can severely limit the degree of concurrency when executing transactions on a data structure.

In this paper, we present a high performance lock-free transactional adjacency list. The primary goal of the data structure presented in this work is to (1) implement a lock-free adjacency list base data structure, and (2) enable transactional execution of operations in this data structure.

In order to achieve the first goal, we implement lock-free adjacency list using a lock-free linked list of vertexes, where each vertex contains a pointer to a Multi-Dimensional List (MDList) [24] to allow fast lookup of edges. We depict the adjacency list structure in Fig. 1. An MDList guarantees a worst-cast search time complexity of $O(\log N)$, an improvement over a worst-cast search time complexity of $O(N)$ provided by design alternatives such as a linked list or skiplist. A skiplist provides an average search time complexity of $O(\log N)$, but has a worst-cast search time complexity of $O(N)$ if shortcuts to the node of interest do not exist. We place all vertexes in the primary linked list, and all adjacent edges to that vertex as a node in its associated MDList. This allows us to take maximum advantage of the multi-dimensional property of the MDList, while also easily organizing the relative locations of each vertex and their corresponding edges. Background details on the MDList are provided in Sect. 2.

We refer to elements in the primary linked list as vertexes, and elements in the sublist of a vertex as nodes. A node a contained in vertex b's associated MDList indicates that vertex a is adjacent to vertex b. When inserting or deleting a vertex, we traverse along the main list of vertexes, checking each key, until we find the location to insert or delete our vertex. While allocating the vertex we also allocate a new MDList for that vertex to point to.

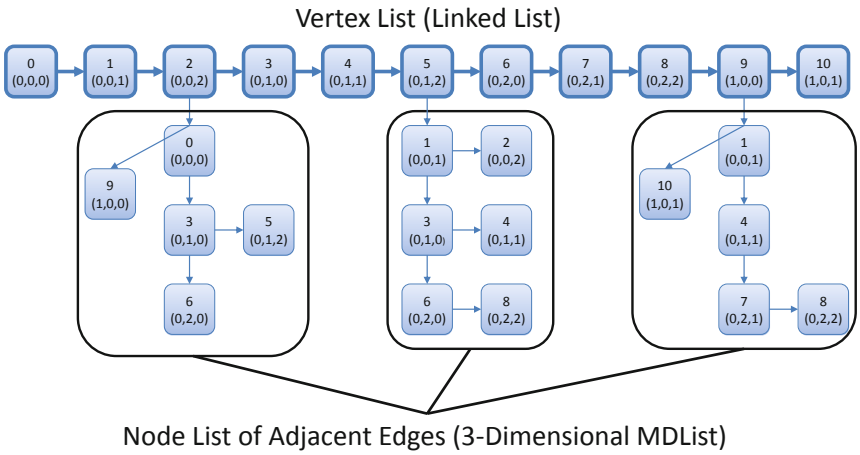


Fig. 1. Adjacency list structure

In order to achieve the second goal, we adopt Lock-Free Transaction Transformation (LFTT) [25] by storing descriptor objects within each node in both the main list and each MDList. LFTT uses high-level semantic conflict detection to avoid low-level read/write conflicts, and a logical rollback to avoid the

performance penalties of a physical rollback. Background details on LFTT are provided in Sect. 2.

The contribution made by this paper is as follows:

- To the best of our knowledge, this paper presents the only lock-free transactional adjacency list.
- This data structure experiences an average speedup greater than 50% when compared to similar approaches based on transactional boosting and STM.

2 Background

An MDList partitions a linked list into shorter lists organized in multi-dimensional space to improve search time. A node in a D-dimensional MDList comprises a key-value pair, a coordinate vector of integers $k[D]$, and an array of child pointers where the dth pointer links to a child node of dimension d. A list of arbitrary dimension D is formally defined as follows.

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 $d' \in [d, D)$ [24].*

Given a key range of $[0, N)$ in a D-dimensional space, the maximum number of keys in each dimension is $b = \lceil \sqrt[D]{N} \rceil$. The mapping of an integer key to its D-dimension vector coordinates is performed by converting the key to a b-based number and using each digit as an entry in the vector coordinates. Each node is associated with a coordinate vector k, where a dimension d node shares a coordinate prefix of length d with its parent. The following definition provides the criteria for which nodes are ordered in their D-dimensional list.

Definition 2. *Given a non-root node of dimension d with coordinate $k = (k_0, ..., k_{D-1})$ and its parent with coordinate $k' = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i, \forall i \in [0, d) \wedge k_d > k'_d$ [24].*

The search for a node is performed by starting at the 0-dimension and traversing all nodes at this dimension until either a node with the same 0th coordinate as the key of interest is reached, or the current node being traversed has a greater 0th coordinate than the key of interest. If a node with a 0th coordinate identical to the key of interest exists, then the search advances to the next dimension d. The search will continue advancing dimensions given that a node with the same dth coordinate as the key of interest is found. The search terminates when either a node with the same coordinates as the key of interest is found, or no node exists with the same dth coordinate as the key of interest.

The worst-case time complexity of a search in an MDList is $O(D \cdot b)$, where b is the maximum number of nodes in a dimension. Replacing b in the worst-cast time complexity, we have $O(D \cdot b) = O(D \cdot \sqrt[D]{N})$. If we choose $D \propto \log N$, then $O(D \cdot \sqrt[D]{N}) = O(\log N \cdot \sqrt[\log N]{N}) = O(\log N \cdot 2) = O(\log N)$.

Insertion into an MDList is performed by splicing and child adoption. Splicing consists of updating the new node's child pointer to point to the predecessor's child, and updating the predecessor's child pointer to point to the new node. Child adoption is necessary when the dimension of an old child has changed due to the insertion of a new node, where the old child will be adopted as a higher dimension child of the new node. Deletion of a node in an MDList is performed by updating the predecessor's child point to point to the child of the node to be deleted. In the case of a deletion, the child of the node to be deleted is adopted as a lower dimension child of the predecessor.

Lock Free Transactional Transformation (LFTT) is a methodology for creating transactional data structures from lock-free node-based data structures. LFTT handles conflicts between operations by utilizing descriptor objects referenced by each node. These transaction descriptors contain all information necessary for an arbitrary thread to perform any given operation or sequence of operations belonging to a transaction. For a thread to perform an operation at a node as part of a transaction, it is must first create a reference to its transaction descriptor in the node. If there already exists a transaction descriptor at that node, a conflict between two transactions accessing the same node has been detected. LFTT resolves these conflicts by having the thread that finds an existing transaction descriptor at a node help complete the conflicting transaction by executing all remaining operations that are part of that transaction, thus eventually causing the conflicting transaction to either succeed or fail. Once the transaction referenced by the transaction descriptor at a node is complete, a thread may place a reference to its own transaction descriptor in the node and proceed with its operation.

LFTT additionally handles the recovery of failed transactions through its transaction descriptors. A transaction descriptor may be marked as *committed*, indicating that all operations that are part of the transaction have been successfully completed. Alternatively, a transaction descriptor may be marked *aborted*, indicating that none of the operations in the transaction should occur. LFTT is able to avoid the need to physically undo already completed operations that are part of an aborted transaction by interpreting the logical status of a node based on its transaction descriptors status. A nodes status in the list is interpreted inversely if it is part of an aborted transaction. This results in the *appearance* that all completed operations that are part of an aborted transaction have been undone.

3 Lock-Free Transactional Adjacency List

The primary challenge in creating a lock-free transactional adjacency list is its multi-dimensional structure, which poses a major challenge to performing transactional synchronization for non-commutative operations. InsertEdge and DeleteEdge create a relation between two vertexes by adding or removing a node from the sublist of an existing vertex. Any InsertEdge or DeleteEdge operation occurring at vertex j would have their outcome affected by a transaction that modifies vertex j. As a result, two edge operations occurring at the

same vertex are able to commute, while an edge operation and an operation that modifies the vertex itself are not. The DeleteVertex method requires special consideration. The case in which a transaction deletes a vertex at which one or multiple threads are performing an edge operation must be prevented. A DeleteVertex operation on a vertex should help complete all pending edge operations currently accessing the MDList contained at that vertex. Simultaneously, any subsequent operations attempting to access that MDList will first help complete the pending DeleteVertex.

The constants provided by LFTT are detailed in Algorithm 1. We introduce a *currentOpid* field to each descriptor to track the current progress of each transaction. The IsNodePresent, IsKeyPresent, ExecuteOps, MarkDelete, LocatePred, and pointer marking operations are provided in Lock-Free Transactional Transformation [25].

Algorithm 1. LFTT Definitions

```
1: enum TxStatus
2:     Active
3:     Committed
4:     Aborted
5: enum OpType
6:     InsertVertex
7:     DeleteVertex
8:     InsertEdge
9:     DeleteEdge
10:    Find
11: struct Operation
12:    OpType type
13:    int key
14: struct Desc
15:    int size
16:    TxStatus status
17:    int currentOpid
18:    Operation ops[]
19: struct NodeDesc
20:    Desc* desc
21:    int opid
22: struct Node
23:    NodeDesc* info
24:    int key
25:    MDList* list
26:    ...
```

Algorithm 2. Update Info Pointer

```
1: function UpdateInfo(Node* n, NodeDesc* info, bool wantkey)
2:     NodeInfo *oldinfo ← n.info
3:     if IsMarked(oldinfo) then
4:         Do_Delete(n)
5:         return retry;
6:     if oldinfo.desc ≠ info.desc then
7:         if oldinfo.desc.ops[oldinfo.opid] == DeleteVertex & oldinfo.desc.currentOpid == oldinfo.opid then
8:             ExecuteOps(oldinfo.desc, oldinfo.opid)
9:         else
10:            ExecuteOps(oldinfo.desc, oldinfo.opid+1)
11:    else if oldinfo.opid >= info.opid then
12:        return success
13:    haskey ← IsKeyPresent(oldinfo)
14:    if (!haskey & wantkey) || (haskey & !wantkey) then
15:        return fail
16:    if info.desc.status ≠ Active then
17:        return fail
18:    if CAS(&n.info, oldinfo, info) then
19:        return success
20:    else
21:        return retry
```

Algorithm 2 contains the UpdateInfo operation provided by Lock-Free Transactional Transformation [25], which has been modified in the following way to allow for a special case regarding DeleteVertex. At line 2.6 we check the info pointer at node n. If a different operation is currently taking place at node n, that operation must be completed before the desired operation can begin. At line 2.7 we check if the operation that occurred at node n was a DeleteVertex operation. If so, we check whether the DeleteVertex operation is pending. The *currentOpid* variable stores what step the transaction is currently on. If this value is equal to the value of the operation that occurred at node n, then the DeleteVertex operation is not complete and the current thread should use the descriptor object to attempt to delete the vertex. For all other operations, the presence of an info pointer at node n indicates that the operation described by *n.info* is already complete. Thus, ExecuteOps is called on the next operation in the transaction.

Algorithm 3. Transformed Delete Vertex

```
1: function DeleteVertex(int vertex, NodeDesc* nDesc)
2:     Node *curr ← head
3:     Node *pred ← NULL
4:     while true do
5:         LocatePred(pred, curr, vertex)
6:         if IsNodePresent(curr, vertex) then
7:             ret ← (UpdateInfo(curr, nDesc, true) == success)
8:             if ret then
9:                 MDList *list ← curr.list
10:                ret ← list.FinishDelete(list.head, 0, nDesc)
11:        else
12:            ret ← false
13:        if ret then
14:            return true
15:        else
16:            return false
```

Algorithm 4. Finish Pending DeleteVertex Operation

```
1: function FinishDelete(MDList::Node* n, int dc, NodeDesc* nDesc)
2:     while true do
3:         if UpdateInfo(n, nDesc, true) == success then
4:             Break
5:         else
6:             return false
7:     for i ∈ [dc,DIMENSION) do
8:         MDList::Node *child ← n.child[i]
9:         CAS(&n.child[i], child, Set_Mark(child))
10:        if child! = NULL then
11:            ret ← FinishDelete(child, i, nDesc)
12:            if ret == false then
13:                return false
14:        else
15:            return true
```

3.1 Adjacency List Operations

This adjacency list supports 5 operations: InsertVertex, DeleteVertex, InsertEdge, DeleteEdge, and Find. The InsertVertex operation adds a vertex to a primary linked list of vertexes. The InsertEdge operation adds a node to a specific vertex's sublist, thus establishing that node as adjacent to the specified vertex. The DeleteVertex and DeleteEdge operations are the inverses of their counterparts. The Find operation searches for a node within the sublist of vertex j, returning whether or not that node shares an edge with vertex j.

Algorithm 5. Find Vertex Operation

```
1: function FindVertex(int vertex, NodeDesc* nDesc, int opid)
2:     Node *curr ← head
3:     Node *pred ← NULL
4:     while true do
5:         LocatePred(pred, curr, vertex)
6:         if IsNodePresent(curr, vertex) then
7:             NodeDesc *cDesc ← curr.info
8:             if cDesc != nDesc then
9:                 ExecuteOps(cDesc.desc,cDesc.opid+1)
10:            if IsKeyPresent(cDesc) then
11:                if nDesc.desc.status != ACTIVE then
12:                    return NULL
13:                else
14:                    return curr
15:        else
16:            return NULL
```

Algorithm 3 details the DeleteVertex operation. DeleteVertex traverses the main list of vertexes by calling LocatePred on line 3.5. If the node with the target key already exists, then LocatePred will return when *curr* points to the node with that key, otherwise, *curr* will point to the logical successor of the node to be deleted. We check for the case that the node with the desired key already exists on line 3.6. We then call UpdateInfo to attempt to redirect the $info$ pointer. If this succeeds, we must then call FinishDelete on the vertex's *list* object. FinishDelete traverses *list* calling UpdateInfo on all the nodes it contains. Additionally, we must mark the next pointer of all nodes as they are traversed, which will interrupt competing InsertEdge operations that have already begun inserting their node on line 6.15, causing them to re-traverse. The goal of this operation is to logically delete all edges adjacent to the vertex to be deleted. This process allows all pending transactions occurring within the sublist to commit due to the call to UpdateInfo at line 4.3. Once it can be guaranteed that all nodes within the vertex's list are deleted, the operation is complete. Physical deletion is later done by using Compare-And-Swap to change *pred.next* to point to *curr.next*, thus removing the vertex from the main list.

The InsertVertex algorithm is similar to DeleteVertex. InsertVertex traverses the list using LocatePred, but can only succeed if its value

is not already in the list (!IsNodePresent($curr, vertex$)). In this case, it allocates a new vertex and inserts it into the list using Compare-And-Swap to change $pred.next$ to $curr$.

Algorithm 5 details the main method used to help the insertion of edge nodes. To begin, it searches the list until it finds the correct vertex node and verifies that it is logically in the list, and that no other transaction currently holds the $info$ pointer. If another thread does hold the $info$ pointer, the thread will help complete that transaction at line 5.9. Otherwise, the function returns a pointer to the node.

Algorithm 6. Insert key:edge at target vertex

```
1:  function InsertEdge(int vertex, int edge, NodeDesc* nDesc, int opid)
2:      while true do
3:          Node  *currVertex ←
4:          FindVertex(vertex, nDesc, opid))
5:          if currVertex == NULL then
6:              return false
7:          Node *pred ← NULL
8:          Node *currEdge ← currVertex.list.head
9:          while true do
10:             currVertex.list.LocatePred(pred, currEdge)
11:             if IsNodePresent(currEdge, edge) then
12:                 return (UpdateInfo(currEdge, nDesc, false) == success)
13:             else
14:                 MDList::Node *n ← new MDList::Node
15:                 n.info ← nDesc
16:                 return currVertex.list.Do_Insert(n)
```

Algorithm 7. Delete key:edge at target vertex

```
1:  function DeleteEdge(int vertex, int edge, NodeDesc* nDesc, int opid)
2:      while true do
3:          Node  *currVertex ←
4:          FindVertex(vertex, nDesc, opid))
5:          if currVertex == NULL then
6:              return false
7:          Node *pred ← NULL
8:          Node *currEdge ← currVertex.list.head
9:          while true do
10:             currVertex.list.LocatePred(pred, currEdge)
11:             if IsNodePresent(currEdge, edge) then
12:                 return (UpdateInfo(currEdge, nDesc, true) == success)
13:             else
14:                 return false
```

Algorithm 6 details the insertion of a node into an MDList in order to create an edge with a vertex. InsertEdge begins by calling FindVertex to get the proper vertex node for insertion. If the node exists, then we traverse the MDList pointed to by the vertex to find the proper location to insert the new edge node. Once the traversal is complete, insertion is done the same way as in InsertVertex.

Algorithm 7 details the deletion of a node in an MDList in order to remove an edge with a vertex. DeleteEdge traverses to the target vertex using the same logic as InsertEdge. Once it has acquired a valid vertex, it traverses the MDList looking for the target edge node to delete. If the target node is found in the MDList, deletion is done by updating the $info$ pointer of the target node.

4 Correctness

The lock-free transactional adjacency list is designed for the correctness property strict serializability. According to conclusion by Herlihy et al. [9], a committed transaction is strictly serializable given that a data structure contains linearizable operations and obeys commutativity isolation.

4.1 Definitions

According to Herlihy et al. [9], a *history* is a sequence of instantaneous events. Events occur during the transition of a transactions status between pending, committed, and aborted.

Definition 3. *A history h is strictly serializable if the committed series of transactions is equivalent to a legal history in which all transactions executed sequentially in the order they commit.*

Definition 4. *Two method calls I,R and I',R' commute if: for all histories h, if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same abstract state.*

Operations are said to commute if executing them in any order yields the same abstract state. The commutativity of adjacency list operations are as follows, assuming vertexes x,y and nodes i,j:

InsertVertex(x) $\leftrightarrow$ InsertVertex(y), x $\neq$ y

DeleteVertex(x) $\leftrightarrow$ DeleteVertex(y), x $\neq$ y

InsertVertex(x) $\leftrightarrow$ DeleteVertex(y), x $\neq$ y

InsertEdge(x, i) $\leftrightarrow$ InsertEdge(x, j), i $\neq$ j

InsertEdge(x, i) $\leftrightarrow$ InsertEdge(y, i), x $\neq$ y

DeleteEdge(x, i) $\leftrightarrow$ DeleteEdge(x, j), i $\neq$ j

DeleteEdge(x, i) $\leftrightarrow$ DeleteEdge(y, i), x $\neq$ y

InsertEdge(x, i) $\leftrightarrow$ DeleteEdge(x, j), i $\neq$ j

InsertEdge(x, i) $\leftrightarrow$ DeleteEdge(y, i), x $\neq$ y

FindVertex(x) $\leftrightarrow$ InsertVertex(x)/*false* $\leftrightarrow$ DeleteVertex(x)/*false*

FindEdge(x, i) $\leftrightarrow$ InsertEdge(x, i)/*false* $\leftrightarrow$ DeleteEdge(x, i)/*false*

Rule 1. ***Linearizability:*** *For any history h, two concurrent invocations I and I' must be equivalent to either the history $h \cdot I \cdot R \cdot I' \cdot R'$ or the history $h \cdot I' \cdot R' \cdot I \cdot R$*

Rule 2. ***Commutativity Isolation:*** *For any non-commutative method calls* $I_1,_1 \in T_1$ *and* $I_2, R_2 \in T_2$*, either* T_1 *commits or aborts before any additional method calls in* T_2 *are invoked, or vice-versa.*

To meet the specifications of the correctness condition linearizability, we identify an operation's linearization points. Furthermore, we will identify an operation's decision points and state-read points. The decision point of an operation occurs the moment the outcome of the operation is decided atomically. A state-read point occurs when the deciding state of the data structure is read.

Lemma 1. *The adjacency list operations* InsertVertex*,* DeleteVertex*,* InsertEdge*,* DeleteEdge*, and* Find *are linearizable.*

Proof. In the DeleteVertex operation, execution can branch at multiple points. Beginning at 3.6, if the vertex to be deleted is not found, the operation returns a *fail* status. The state-read point of this execution occurs during traversal, when the thread reads *pred.next* and does not find a node with the desired key. If the vertex is successfully found, but the operation returns $fail$ at line 2.15 or 2.17, then the state-read point occurs when $oldinfo.desc.status$ and $info.desc.status$ are read, respectively. Following a successful logical status update, the decision point is when the CAS operation at line 2.18 succeeds.

The code path for FinishDelete, in which all nodes in the vertex's sublist are acquired by the transaction, is identical to the code path followed by DeleteVertex because of the call to UpdateInfo at line 4.3. Thus, the state-read and decision points for FinishDelete are the same as the respective cases in DeleteVertex. The code path for the physical deletion of the vertex is linearizable because Do_Delete, which is provided by the base data structure, is linearizable.

The same reasoning applies to the InsertVertex, InsertEdge, DeleteEdge and Find operations because they share the same UpdateInfo procedure for updating the logical status of a node.

Lemma 2. *The adjacency list operations* InsertVertex*,* DeleteVertex*,* InsertEdge*,* DeleteEdge*, and* Find *satisfy the commutativity isolation rule.*

Proof. As previously shown, commuting operations are those that access different vertexes, or those that access different nodes within the same vertex so long as no operation is operating on that vertex. This means that commuting operations must either operate on different vertexes or operate on different nodes rooted at the same vertex without operating on the vertex itself. Let T_1 denote a transaction that currently accesses vertex *n1*. If another transaction T_2 were to access *n1*, it must first perform ExecuteOps for T_1 which will either commit or abort T_1 before it is finished executing. Alternatively, let T_1 denote a transaction that currently accesses node *m1* stored in the sublist of vertex *n1*. If a transaction T_2 were to try to access vertex *n1* it would first perform ExecuteOps for T_1 when it traverses to node *m1* during the call to FinishDelete at 2.10, which will either commit or abort T_1.

Theorem 1. *The transformed lock-free adjacency list is strictly serializable*

Proof. Following Lemmas 1 and 2, we can claim that the lock-free adjacency list is strictly serializable due to the conclusions by Herlihy and Koskinen [9].

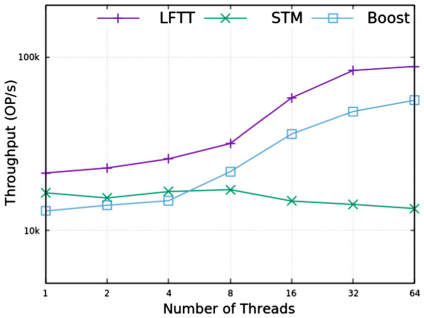
5 Experimental Evaluation

We compare the scalability and performance of our lock-free transactional adjacency list to related approaches based on transactional boosting [9] and NOrec Rochester Software Transactional Memory [13]. We create a related approach using transactional boosting by converting the lock-free transactional adjacency list's base data structure operations transaction boosting methodology. Additionally, an undo log is maintained per-thread for rollbacks in the boosted implementation.

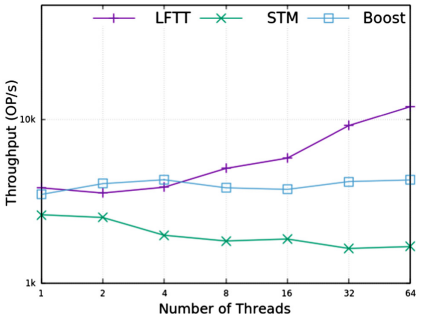
We evaluate the performance of these implementations using varying compositions of adjacency list operations. The compositions of operations are selected to highlight "vertex" operations and "edge" operations separately, as well measuring the effects of non-commutative or expensive operations like DeleteVertex. Each test consists of a series of fixed-size transactions made up of InsertVertex, DeleteVertex, InsertEdge, DeleteEdge and Find operations on random keys. The tests are performed on two systems; a 64-core NUMA system containing 4 AMD opteron 6272 16 core CPUs @2.1 GHz, and a 12-core system containing an Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.7 ghz.

Figure 2 shows the performance results of the 64-core NUMA system. Figure 3 shows the results for the 12-core system. Throughput is measured in terms of operations per second. Only operations that are part of a committed transactions are counting in the calculation of throughput in order to measure the performance impact of various conflict detection and rollback schemes. The x-axis represents the number of threads running the test. In each figure, graph (a) shows a work-load dominated by operations occurring at vertexes, whereas graph (b) represents a work-load made up of relatively more operations occurring at edges. This test measures the performance impact of non-commutative operations such as DeleteVertex and InsertEdge as well as the performance impact of rollbacks on lengthy operations such as DeleteVertex. Each thread executed 20,000 transactions with a key range of 500.

In Fig. 2, the difference between the lock-free transactional adjacency list, denoted 'LFTT,' the transactional boosting implementation, denoted 'Boost,' and the Software Transactional Memory implementation, denoted 'STM' is shown. In the boosting implementation, threads must acquire locks on nodes for each operation. In the case of DeleteVertex, threads may need to acquire a number of locks equal to the size of the vertex's sublist. In this case, the lock-free algorithm has the advantage of only needing to allocate a single descriptor object for the entire transaction. Additionally with regards to Boost, the cost of rolling back aborted operations is very high in operations like DeleteVertex. Not only must the vertex be restored after an aborted transaction, but all

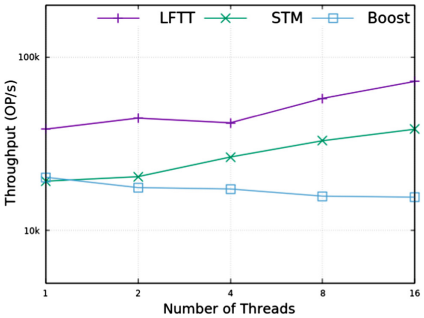


(a) 40% InsertVertex, 40% DeleteVertex, 10% InsertEdge, 10% DeleteEdge

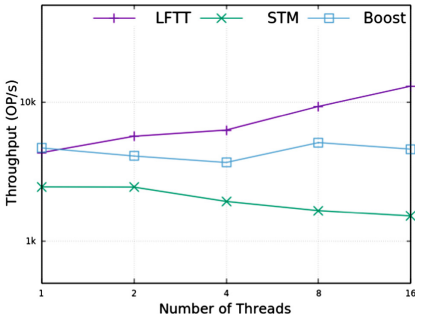


(b) 20% InsertVertex, 20% DeleteVertex, 25% InsertEdge, 25% DeleteEdge, 10% Find

Fig. 2. Performance Results



(a) 40% InsertVertex, 40% DeleteVertex, 10% InsertEdge, 10% DeleteEdge



(b) 20% InsertVertex, 20% DeleteVertex, 25% InsertEdge, 25% DeleteEdge, 10% Find

Fig. 3. Performance Results

nodes from the vertex's sublist must be re-added using InsertEdge. This creates a very low performance for aborted transactions in transactional boosting. Because of LFTT's logical status update, the lock-free transactional adjacency list is able to rollback these operations in a single atomic step. Similarly, STM experiences a heavy performance loss due to its high number of spurious aborts. STM is very likely to detect a conflict between operations like DeleteVertex, which modify a great number of nodes, despite there being no semantic conflict between transactions. These results are highly similar to the ones gathered from the 12-core system displayed in Fig. 3.

In general, the lock-free transactional adjacency list outperforms transactional boosting implementation by an average of 50%, and frequently outperforms RSTM by as much as 150%.

6 Related Work

Transactions can be enabled in similar data structures using related approaches such as STM or Transactional Boosting. We focus our discussion on transactional Lists and Skiplists, which provide similar node-based store and search time complexities.

6.1 Transactional Memory

Transactional memory is a programming paradigm initially proposed by Herlihy et al. [11] intended to simplify concurrent programming by allowing user-specified blocks of code to be executed in hardware, exhibiting both atomicity and isolation. Software transactional memory, proposed by Shavit et al. [18], was developed to facilitate transactional programming without hardware transactional memory support. Herlihy et al. [10] present DSTM, an application programming interface for obstruction-free STM designed to support dynamic-sized data structures. Dalessandro et al. [2] present NOrec, a low-overhead STM that utilizes a single global sequence lock shared with the transactional mutex lock system, an indexed write set, and value-based conflict detection to provide features such as livelock freedom, full compatibility with existing data structure layouts, and starvation avoidance mechanisms. Dice et al. [4] present Transactional Locking II (TL2), an STM algorithm that uses a novel version-clock validation to guarantee that user code operates only on consistent memory states. Other STM designs include [6,13,16]. STM implementations rely on low-level conflict detection to enable transactions. These implementations generally suffer from high spurious abort counts, making them less desirable for concurrent data structures.

Initial performance experiments were performed with Hardware Transactional Memory (HTM) by Dice et al. [3]. Intel introduced Transactional Synchronization Extensions (TSX) to the x86 instruction set architecture of the Intel 4th Generation Core^{TM}Processors [23]. IBM introduced HTM in the Power ISA [1]. Both implementations offer a best-effort HTM, which means that there is no guarantee provided that a hardware transaction will commit to memory. The disadvantage of a best-effort strategy is that HTM may experience frequent aborts due to data access conflicts, hardware interrupts, limited transactional resources, or false sharing due to unrelated variables mapping to the same cache line [12].

Herlihy et al. [9] present transactional boosting, a methodology for transforming highly-concurrent linearizable objects into highly-concurrent transactional objects. Transactional boosting uses a high-level semantic conflict detection to allow commutative operations in separate transactions to proceed concurrently using the thread-level synchronization of the base linearizable data structure; non-commutative operations require transaction-level synchronization through the acquisition of an abstract lock. If a transaction aborts, it recovers the correct abstract state by invoking the inverse operations recorded in the undo log.

6.2 Linked Lists

Valois [22], Harris [7], Michael [14], and Fomitchev et al. [5] present individual algorithms for a lock-free linearizable linked list based on the Compare-And-Swap operation. Valois' algorithm addresses the problem of (1) a concurrent deletion and insertion on an adjacent cell, and (2) a concurrent deletion and deletion on an adjacent cell, by requiring that every normal node in the list have an auxiliary node with only a next field as both its predecessor and successor. The auxiliary nodes prevent the undesirable circumstance of performing an insertion or deletion on a node adjacent to a node to be deleted. Harris' algorithm uses the bit-stealing technique to logically mark a node for deletion. A lazy approach is taken for the physical deletion in which a delete operation attempts to physically delete a node once using Compare-And-Swap. If Compare-And-Swap fails, then the physical deletion is left for other threads to perform if they traverse the logically deleted node. Michael's algorithm is compatible with efficient lock-free memory management methods, including IBM freelists [21] and the safe memory reclamation method [15]. Fomitchev et al.'s algorithm uses backlinks that are set when a node is deleted to allow a node to backtrack to a predecessor that is not undergoing a deletion. An MDList provides a worst-case search time complexity of $O(\log N)$ an improvement over the $O(N)$ worst-cast search time complexity provided by a linked list.

Transactional linked list implementations based on transactional boosting use coarse-grained locking to ensure that non-commutative method calls are never allowed to execute simultaneously. The underlying linked list algorithm's linearizability is preserved during this process to handle thread level synchronization. Rollbacks are performed by calling a method's inverse operation, which causes a performance loss for aborted transactions. Zhang and Dechev [24] present a lock-free transactional linked list alongside LFTT which takes advantage of a node based conflict detection scheme to preserve the underlying algorithm's lock-freedom. This approach additionally reduces the performance hit of rollbacks by introducing a logical status update scheme capable of aborting a transaction in a single atomic step. LFTT provides transformation templates for the set abstract data type, which does not account for operations in which elements are related to each other.

6.3 Skiplists and Queues

Sundell et al. [20] present a lock-free priority queue based on a lock-free skiplist adapted from Lotan et al. [17]. Fomitchev et al. [5] use their lock-free linked list design [5] to implement a lock-free skiplist. Each node is augmented with a pointer to the next lower level and a pointer to the base level. Herlihy et al. [8] present a lock-free skiplist based on an algorithm developed by Faser [6]. Skiplists eliminate global rebalancing and provide a logarithmic sequential search time on average, but the worst-case search time is linear with respect to the input size. An MDList improves upon the skiplist by providing a worst-case logarithmic sequential search time.

Spiegelman et al. [19] presented a transactional skiplist that uses STM-like techniques combined with node locking in an attempt to reduce overhead and false aborts. Spiegelman et al. additionally present a transactional queue using a pessimistic lock-based approach. In this queue, the execution of ENQUEUE operations are deferred to the final phase of the transaction, the commit phase, in order to avoid keeping track of the current head of the queue. Meanwhile, DEQUEUE operations acquire a lock on the queue until their transaction is complete. Zhang and Dechev [24] preserved lock-freedom in their algorithm by transforming a skiplist using LFTT which, again, offers a performance improvement on transaction rollbacks.

7 Conclusion

In this paper we introduced an efficient lock-free adjacency list algorithm based on MDList, then enabled transactions using the LFTT methodology. We allowed for multiple threads to concurrently modify nodes rooted at the same vertex thus increasing the amount of operations that commute. When compared to similar implementations based on related approaches, our algorithm experiences performance gains across several compositions of methods.

References

1. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust architectural support for transactional memory in the power architecture. In: ACM SIGARCH Computer Architecture News, vol. 41, pp. 225–236. ACM (2013)
2. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: ACM Sigplan Notices, vol. 45, pp. 67–78. ACM (2010)
3. Dice, D., Lev, Y., Moir, M., Nussbaum, D., Olszewski, M.: Early experience with a commercial hardware transactional memory implementation. Sun Microsystems Technical report (2009)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006). https://doi.org/10.1007/11864219_14
5. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, pp. 50–59. ACM (2004)
6. Fraser, K.: Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory (2004)
7. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45414-4_21
8. Herlihy, M., Lev, Y., Shavit, N.: A lock-free concurrent skiplist with wait-free search. Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts (2007)
9. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207–216. ACM (2008)

10. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM (2003)
11. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures, vol. 21. ACM (1993)
12. Intel. Intel 64 and IA-32 architectures optimization reference manual (2016)
13. Marathe, V.J., et al.: Lowering the overhead of nonblocking software transactional memory. In: Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT) (2006)
14. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 73–82. ACM (2002)
15. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 21–30. ACM (2002)
16. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197. ACM (2006)
17. Shavit, N., Lotan, I.: Skiplist-based concurrent priority queues. In: 2000 Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS 2000, pp. 263–268. IEEE (2000)
18. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
19. Spiegelman, A., Golan-Gueta, G., Keidar, I.: Transactional data structure libraries. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016), vol. 51, pp. 682–696. ACM (2016)
20. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. In: 2003 Proceedings of the International Parallel and Distributed Processing Symposium, pp. 11–pp. IEEE (2003)
21. Treiber, R.K.: Systems programming: coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
22. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222. ACM (1995)
23. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In: 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11. IEEE (2013)
24. Zhang, D., Dechev, D.: An efficient lock-free logarithmic search data structure based on multi-dimensional list. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 281–292. IEEE (2016)
25. Zhang D., Dechev, D.: Lock-free transactions without rollbacks for linked data structures. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 325–336. ACM (2016)