Lock-free Transactions without Rollbacks for Linked Data Structures

Deli Zhang
de-li.zhang@knights.ucf.edu

Damian Dechev
dechev@cs.ucf.edu

Department of Computer Science
University of Central Florida
Orlando, FL 32817, USA

ABSTRACT

Non-blocking data structures allow scalable and thread-safe accesses to shared data. They provide individual operations that appear to execute atomically. However, it is often desirable to execute multiple operations atomically in a transactional manner. Previous solutions, such as software transactional memory (STM) and transactional boosting, manage transaction synchronization in an external layer separated from the data structure's own thread-level concurrency control. Although this reduces programming effort, it leads to overhead associated with additional synchronization and the need to rollback aborted transactions.

In this work, we present a new methodology for transforming high-performance lock-free linked data structures into high-performance lock-free transactional linked data structures without revamping the data structures' original synchronization design. Our approach leverages the semantic knowledge of the data structure to eliminate the overhead of false conflicts and rollbacks. We encapsulate all operations, operands, and transaction status in a transaction descriptor, which is shared among the nodes accessed by the same transaction. We coordinate threads to help finish the remaining operations of delayed transactions based on their transaction descriptors. When transaction fails, we recover the correct abstract state by reversely interpreting the logical status of a node.

In our experimental evaluation using transactions with randomly generated operations, our lock-free transactional lists and skiplist outperform the transactional boosted ones by 40% on average and as much as 125% for large transactions. They also outperform the alternative STM-based approaches by a factor of 3 to 10 across all scenarios. More importantly, we achieve 4 to 6 orders of magnitude less spurious aborts than the alternatives.

CCS Concepts

•**Computing methodologies** → **Concurrent algorithms;**

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935780

Keywords

Transactional Data Structure, Lock-free, Transactional Memory, Transactional Boosting

1. INTRODUCTION

With the growing prevalence of multi-core systems numerous highly concurrent non-blocking data structures have emerged [8, 27, 31, 39]. Researchers and advanced users have been using libraries like LibCDS [1], Tervel [2] and Intel TBB [3], which are packed with efficient concurrent implementations of fundamental data structures. High level programming languages such as C#, JAVA and Scala also introduce concurrent libraries, which allow users who are unaware of the pitfalls of concurrent programming to safely take advantage of the performance benefits of increased concurrency. These libraries provide operations that appear to execute atomically when invoked individually. However, they fall short when users need to execute a sequence of operations atomically (i.e., compose operations in the manner of a transaction). For example, given a concurrent map data structure, the following code snippet implementing a simple ComputeIfAbsent pattern [11] is error prone.

```
if(!map.containsKey(key)) {
    value = ... // some computation
    map.put(key, value); }
```

The intention of this code is to compute a value and store it in the map, if and only if the map does not already contain the given key. The code snippet fails to achieve this since another thread may have stored a value associated with the same key right after the execution of containsKey and before the invocation of put. As a result, the thread will overwrite the value inserted by the other thread upon the completion of put. Programmers may experience unexpected behavior due to the violation of the intended semantics of ComputeIfAbsent. Many Java programs encounter bugs that are caused by such non-atomic composition of operations [36]. Because of such hazards, users are often forced to fall back to locking and even coarse-grained locking, which has a negative impact on performance and annihilates the non-blocking progress guarantees provided by some concurrent containers.

The problem of implementing high-performance transac-

[1] http://libcds.sourceforge.net/
[2] http://ucf-cs.github.io/Tervel/
[3] https://www.threadingbuildingblocks.org/

tional data structures [4] is important and has recently gained much attention [2, 11, 12, 13, 19, 21, 26]. We refer to a transaction as sequence of linearizable operations on a concurrent data structure. This can be seen as a special case of memory transactions where the granularity of synchronization is on the data structure operation level instead of memory word level. We consider a concurrent data structure "transactional" if it supports executing transactions 1) atomically (i.e., if one operation fails, the entire transaction should abort), and 2) in isolation (i.e., concurrent executions of transactions appear to take effect in some sequential order).

Software transactional memory (STM) [22,37] can be used to conveniently construct transactional data structures from their sequential counterparts: operations executed within an STM transaction are guaranteed to be transactional. Despite the appeal of straightforward implementation, this approach has yet to gain practical acceptance due to its significant runtime overhead [3]. An STM instruments threads' memory accesses by recording the locations a thread reads in a *read set*, and the locations it writes in a *write set*. Conflicts are detected among the *read/write sets* of different threads. In the presence of conflicts, only one transaction is allowed to commit while the others are aborted and restarted. Apart from the overhead of metadata management, excessive transaction aborts in the presence of data structure "hot-spots" (memory locations that are constantly accessed by threads, e.g., the head node of a linked list) limit the overall concurrency [21]. The inherent disadvantage of STM concurrency control is that *low-level memory access conflicts do not necessarily correspond to high-level semantic conflicts*.

In this paper, we present *lock-free transactional transformation*: a methodology for transforming high-performance lock-free *base data structures* into high-performance lock-free transactional data structures. Our approach is applicable to a large class of linked data structures—ones that comprise a set of data nodes organized by references. We focus our discussion here on the data structures that implement the set *abstract data type* with three canonical operations INSERT, DELETE, and FIND. Linked data structures are desirable for concurrent applications because their distributed memory layout alleviates contention [38]. The specification for the base data structures thus defines an *abstract state*, which is the set of integer keys, and a *concrete state*, which consists of all accessible nodes. Lock-free transactional transformation treats the base data structure as a *white box*, and introduces a new code path for transaction-level synchronization using only the single-word COMPARE-ANDSWAP (CAS) synchronization primitive. The two key challenges for high-performance data structure transaction executions are: 1) to efficiently buffer write operations so that their modifications are invisible to operations outside the transaction scope; and 2) to minimize the penalty of rollbacks when aborting partially executed transactions.

To overcome the first challenge, we employ a cooperative transaction execution scheme in which threads help each other finish delayed transactions so that the delay does not propagate across the system. We embed a reference to a *transaction descriptor* in each node, which stores the instructions and arguments for operations along with a flag indicating the status of the transaction (i.e., active, committed, or aborted). A transaction descriptor is shared among a group of nodes accessed by the same transaction. When an operation tries to access a node, it first reads the node's descriptor and proceeds with its modification only if the descriptor indicates the previous transaction has committed or aborted. Otherwise, the operation helps execute the active transaction according to the instructions in the descriptor.

To overcome the second challenge, we introduce *logical rollback*—a process integrated into the transformed data structure to interpret the *logical status* of the nodes. This process interprets the logical status of the nodes left behind by an aborted transaction in such a way that concurrent operations observe a consistent abstract state as if the aborted transaction has been revoked. The logical status defines how the concrete state of a data structure (i.e., set of nodes) should be mapped to its abstract state (i.e., set of keys). Usually the mapping is simple—every node in the concrete state corresponds to a key in the abstract state. Previous works on lock-free data structures have been using *logical deletion* [18], in which a key is considered removed from the abstract state if the corresponding node is bit-marked. Logical deletion encodes a binary logical status so that a node maps to a key only if its reference has not been bit-marked. We generalize this technique by interpreting a node's logical status based on the combination of transaction status and operation type. The intuition behind our approach is that operations can recover the correct abstract state by *inversely interpreting* the logical status of the nodes with a descriptor indicating an aborted transaction. For example, if a transaction with two operations INSERT(3) and DELETE(4) fails because key 4 does not exist, the logical status of the newly inserted node with key 3 will be interpreted as *not inserted*.

We applied lock-free transaction transformation on an existing lock-free linked list and a lock-free skiplist to obtain their lock-free transactional counterparts. In our experimental evaluation, we compare them against the transactional data structures constructed from transactional boosting, a word-based STM, and an object-based STM. We execute a micro-benchmark on a 64-core NUMA system to measure the throughput and number of aborts under three types of workloads. The results show that our transactional data structures achieve an average of 40% speedup over transactional boosting based approaches, an average of 10 times speedup over the word-based STM, and an average of 3 times over the object-based STM. Moreover, the number of aborts generated by our approach are 4 orders of magnitude less than transactional boosting and 6 orders of magnitude less than STMs.

This paper makes the following contributions:

- To the best of our knowledge, lock-free transactional transformation is the first methodology that provides both lock-free progress and semantic conflict detection for data structure transactions.
- We introduce a node-based conflict detection scheme that does not rely on STM nor require the use of an additional data structure. This enables us to augment linked data structures with native transaction support.
- We propose an efficient recovery strategy based on interpreting the logical status of the nodes instead of explicitly revoking executed operations in an aborted transaction.

[4] Also referred as *atomic composite operations* [11]

- Data structures transformed by our approach gain substantial speedup over alternatives based on transactional boosting and the best-of-breed STMs; Due to cooperative execution in our approach, the number of aborts caused by node access conflict is brought down to a minimum.
- Because our transaction-level synchronization is compatible with the base data structure's thread-level synchronization, we are able to exploit the considerable amount of effort devoted to the development of lock-free data structures.

The rest of the paper is organized as follows. In Section 2, we explain our methodology in details. We present the template for building transactional linked lists and skiplists in Section 3. We reason about the correctness and progress properties of our algorithms in Section 4. The performance evaluation is presented in Section 5. In Section 6, we review existing approaches on constructing transactional data structures. We conclude the paper in Section 7.

2. LOCK-FREE TRANSACTIONAL TRANSFORMATION

Any transactional data structure must cope with two tasks: conflict detection and recovery. In previous works [12, 21], locks were commonly used to prevent access conflict, and undo logs were often used to discard speculative changes when a transaction aborts. In this work we introduce *lock-free transactional transformation*: a methodology that streamlines the execution of a transaction by abolishing locks and undo logs. Our lock-free transactional transformation combines three key ideas: 1) node-based semantic conflict detection; 2) interpretation-based logical rollback; and 3) cooperative transaction execution. In this section, we explain these ideas and introduce the core procedures that will be used by transformed data structures: a) the procedure to interpret the logical status of a node; b) the procedure to update an existing node with a new transaction descriptor; and c) the transaction execution procedure that orchestrates concurrent executions.

For clarity, we list the constants and data type definitions in Algorithm 1. In addition to the fields used by the base lock-free data structure, we introduce a new field $info$ in NODE. NODEINFO stores *desc*, a reference to the shared transaction descriptor, and an index *opid*, which provides a history record on the last access (i.e., given a node n, $n.info.desc.ops[n.desc.opid]$ is the most recent operation that accessed it). A node is considered *active* when the last transaction that accessed the node had an active status (i.e., $n.info.desc.status$ = Active). A descriptor [24] is a shared object commonly used in lock-free programming to announce steps that cannot be done by a single atomic primitive. The functionalities of our transaction descriptor is twofold: 1) it stores all the necessary context for helping finish a delayed transaction; and 2) it shares the transaction status among all nodes participating in the same transaction. For set operations, we pack the high-level instructions including the operation type and the operand using only 8 bytes per operation. We also employ the pointer marking technique described by Harris [18] to designate logically deleted nodes. The macros for pointer marking are defined in Algorithm 2. The $Mark$ flag is co-located with the $info$ pointers.

Algorithm 1 Type Definitions

```
1: enum TxStatus
2:     Active
3:     Committed
4:     Aborted
5: enum OpType
6:     Insert
7:     Delete
8:     Find
9: struct Operation
10:    OpType type
11:    int key
12: struct Desc
13:    int size
14:    TxStauts status
15:    Operation ops[ ]
16: struct NodeInfo
17:    Desc* desc
18:    int opid
19: struct Node
20:    NodeInfo* info
21:    int key
22:    ...
```

Algorithm 2 Pointer Marking

```
1: int Mark ← 0x1
2: define SetMark(p) (p | Mark)
3: define ClearMark(p) (p & ~Mark)
4: define IsMarked(p) (p & Mark)
```

2.1 Node-based Conflict Detection

In our approach, conflicts are detected at the granularity of a node. If two transactions access different nodes (i.e. the method calls in them commute [21]), they are allowed to proceed concurrently. In this case, shared-memory accesses are coordinated by the concurrency control protocol in the base data structure. Otherwise, threads proceed in a cooperative manner as detailed in Section 2.4. For set data types, each node is associated with a unique key, thus our conflict detection operates at the same granularity as the abstract locking used by transactional boosting.

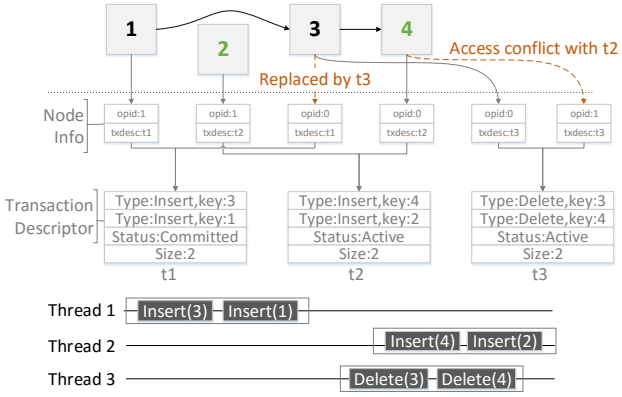


Figure 1: Transaction Execution and Conflict

We illustrate an example of node access conflict in Figure 1. At the beginning, Thread 1 committed a transaction $t1$ inserting keys 1 and 3. Thread 2 attempted to insert keys 4 and 2 in transaction $t2$, while Thread 3 was concurrently executing transaction $t3$ to delete keys 3 and 4. Thread 3 was able to perform its first operation by updating the $info$ pointer on node 3 with a new NODEINFO. However, it encounters a conflict when attempting to update node 4 because Thread 2 has yet to finish its second operation. To enforce serialization, operations must not modify an active node. In order to guarantee lock-free progress, the later

Algorithm 3 Logical Status

```
1: function IsNodePresent(Node* n, int key)
2:   return n.key = key
3: function IsKeyPresent(NodeInfo* info, Desc*desc)
4:   OpType op ← info.desc.ops[info.opid]
5:   TxStatus status ← info.desc.status
6:   switch (status)
7:     case: Active
8:       if info.desc = desc then
9:         return op = Find or op = Insert
10:      else
11:        return op = Find or op = Delete
12:    case: Committed
13:      return op = Find or op = Insert
14:    case: Aborted
15:      return op = Find or op = Delete
```

transaction must help carry out the remaining operations in the other active transaction.

Our synchronization protocol is *pessimistic* in that it assigns a node to a transaction as soon as an operation requires it, for the duration of the transaction. Moreover, node-based conflict detection effectively compartmentalizes the execution of transactions. Completed operations will not be affected should a later operation in the transaction experience contention and need to retry (most lock-free data structures use CAS-based retry loops). Each node acts as a checkpoint; once an operation successfully updates a node, the transaction advances one step towards completion. Data-based conflict detection, due to the lack of such algorithm-specific knowledge, has to restart the whole transaction upon conflict.

2.2 Logical Status Interpretation

To achieve isolation in transaction executions, a write operation needs to buffer or "hide" its update until the transaction commits; and to achieve atomicity it needs to revoke its modifications upon transaction abort. In the context of data structure transactions, existing strategies undo the operations by invoking their inverse operations [21]. This would incur a significant penalty because the compute cycles spent on the inverse operations do not contribute to the overall throughput and introduce additional contention. We approach the recovery task from another angle: an aborted transaction does not need to physically invoke the inverse methods; executed operations in an aborted transaction just need to *appear to have be undone*. We achieve this by having operations *inversely interpret* the logical status of nodes accessed by operations in an aborted transaction. Both physical undo and our logical undo reach the same goal of restoring the abstract state of the data structure.

In Algorithm 3, we list the function to interpret the logical status of a node according to the value of the transaction descriptor. Function IsNodePresent, verifies that a node associated with a specific key is present. This is a common test found in existing linked data structures. Given a node's presence, function IsKeyPresent verifies if the key should be logically included in the abstract state, and returns a boolean value based on the combination of operation type and transaction status. For a node most recently accessed by an Insert operation, its key is considered present if the transaction has successfully *committed*. On the contrary, according to the semantics of Delete, a successful operation must remove the key from the set. Thus for a node most recently accessed by a Delete operation, its key is considered present if the transaction has *aborted*. These two opposite interpretations also match previous observations that Insert and Delete are a pair of inverse operations [21]. Since the Find operation is read-only, no rollback is needed. The node's key is always present regardless the status of the transaction. A special case is the operations in an active transaction, which we treat as committed but are visible only to subsequent operations within the same transaction scope.

Algorithm 4 Update NodeInfo

```
1: function UpdateInfo(Node* n, NodeInfo* 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:     ExecuteOps(oldinfo.desc, oldinfo.opid + 1)
8:   else if oldinfo.opid ≥ info.opid then
9:     return success
10:  bool haskey ← IsKeyPresent(oldinfo)
11:  if (!haskey and wantkey) or (haskey and !wantkey)
   then
12:    return fail
13:  if info.desc.status ≠ Active then
14:    return fail
15:  if CAS(&n.info, oldinfo, info) then
16:    return success
17:  else
18:    return retry
```

2.3 Logical Status Update

As mentioned above, the logical status of a node depends on the interpretation of its transaction descriptor. In a transformed data structure, an operation needs to change a node's logical status before performing any necessary low-level node manipulations. This is done by updating the node's NodeInfo pointer as shown in Algorithm 4. Given a node n, the function UpdateInfo reads its current $info$ field (line 4.2 [5]), verifies its sanity, and attempts to update $n.info$ through the use of CAS (line 4.15). It returns a tri-state value indicating whether the operation succeeded, failed, or should be retried. We make sure that any other active transaction accessing n is completed by helping execute its remaining operations (line 4.7). However, we have to avoid helping the same transaction because of the hazard of infinite recursions. This is prevented by the condition check on line 4.6. We also skip the update and report success if the operation has already been performed by other threads (line 4.8). Due to the use of the helping mechanism, the same operation may be executed multiple times by different threads. The condition check on line 4.8 allows us to identify the node accessed by threads that execute the same transaction, and ensures consistent results. We validate the presence of the key on line 4.10 and test if the key's presence

[5]We denote line b from algroithm a by $a.b$

Algorithm 5 Transaction Execution

```
1: thread_local Stack helpstack
2: function ExecuteTransaction(Desc* desc)
3:   helpstack.Init()
4:   ExecuteOps(desc, 0)
5:   return desc.status = Committed

6: function ExecuteOps(Desc* desc, int opid)
7:   bool ret ← true
8:   set delnodes
9:   if helpstack.Contains(desc) then
10:    CAS(&desc.flag, Active, Aborted)
11:    return
12:  helpstack.Push(desc)
13:  while desc.status = Active and ret and opid < desc.size do
14:    Operation* op ← desc.ops[opid]
15:    if op.type = Find then
16:      ret ← Find(op.key, desc, opid)
17:    else if op.type = Insert then
18:      ret ← Insert(op.key, desc, opid)
19:    else if op.type = Delete then
20:      Node* del
21:      ret ← Delete(op.key, desc, opid, del)
22:      delnodes.Insert(del)
23:    opid ← opid + 1
24:  helpstack.Pop()
25:  if ret = true then
26:    if CAS(&desc.flag, Active, Committed) then
27:      MarkDelete(delnodes, desc)
28:  else
29:    CAS(&desc.flag, Active, Aborted)
```

(as required by deletions and finds), or lack of presence (as required by insertions), is desired on line 4.11. The boolean flag *wantkey* is passed on by the caller function to indicate if the presence of key is desired. The operation reports failure when *wantkey* contradicts *haskey*. Finally, we validate that the transaction is still active (line 4.13) to prevent a terminated transaction from erroneously overwriting $n.info$.

2.4 Transaction Execution

We consider the transaction execution model in which a transaction explicitly aborts upon the first operation failure. The ExecuteTransaction in Algorithm 5 is the entry point of transaction execution, which is invoked by the thread that initiates a transaction. The ExecuteOps function executes operations in sequence starting from *opid*. For threads that help to execute a delayed transaction, the *opid* could be in the range of [1, *desc.size*]. In each step of the while loop (line 5.13), the return value of the previous operation is verified. We require the operations to return a boolean value indicating if the executions are successful. A false value indicates the precondition required by the operation is unsatisfied and the transaction will abort. Once all operations complete successfully we atomically update the transaction status with a *Committed* flag (line 5.26). It is not necessary to retry this CAS operation as a failed CAS indicates that some other thread must have successfully updated the transaction status. The thread that successfully executed the CAS will be responsible for performing physical node deletion (line 5.27), which will be explained in Section 3.

By adopting cooperative transaction execution, our approach is able to eliminate the majority of aborts caused by access conflicts. Although rare, potential livelock is possible if two threads were to access two of the same nodes in opposite order. In such cases, both threads will be trapped in infinite recursions helping execute each other's transaction. We detect and recover from this hazard by using a per-thread *help stack*, which is a simple stack containing *Desc* pointers. This is similar to a function call stack, except it records the invocation of ExecuteOps. A thread initializes its help stack before initiating a transaction. Each time a thread begins to help another transaction, it pushes the transaction descriptor onto its help stack. A thread pops its help stack once the help completes. Cyclic dependencies among transactions can be detected by checking for duplicate entries in the help stack (line 5.9). We recover by aborting one of the transactions as shown on line 5.10.

2.5 Obstruction-free Approach

Our helping mechanism invokes a slight memory and performance overhead, due to the use and maintenance of the per-thread help stack. In situations of low contention, it may be beneficial to eliminate this overhead by disabling the helping mechanism. We introduce an alternative obstruction-free approach that disables the helping mechanism, and we compare it to our original lock-free approach.

Obstruction freedom guarantees that if a method executes without interruption from other threads, then it will finish in a finite number of steps. To obtain our obstruction-free approach, we adopt an aggressive contention management strategy. If a thread tries to update a node n while another active transaction is accessing n, then the current thread forcibly aborts the other transaction. This approach eliminates the overhead of the helping mechanism while still enforcing serialization.

In Algorithm **??**, we list the obstruction-free approach's methods for transaction execution. They are similar to the methods in the lock-free approach, but they do not create or maintain a help stack of descriptors. We list the obstruction-free UpdateInfo function in Algorithm **??**. Before performing its update on the target node n, the thread checks if another transaction is currently accessing n and aborts that transaction if it is still active (line **??**.**??**).

3. APPLICATION EXAMPLES

In this section we demonstrate the application of our lock-free transaction transformation on linked lists and skiplists based sets. The process involves two steps: 1) identify and encapsulate the base data structure's methods for locating, inserting, and deleting nodes; and 2) integrate the UpdateInfo function (Algorithm 4) in each operation using the templates provided in this section.

The first step is necessary because we still rely on the base algorithm and its concurrency control to add, update, and remove linkage among nodes. This is a refactoring process, as we do not alter the functionality of the base implementations. Although implementation details such as argument types and return values may vary, we need to extract the following three functions: Do_LocatePred, Do_Insert, and Do_Delete. We add a prefix Do_ to indicate these are the methods provided by the base data structures. For brevity,

Algorithm 6 Template for Transformed Insert Function

```
1: function INSERT(int key, Desc* desc, int opid)
2:   NodeInfo* info ← new NodeInfo
3:   info.desc ← desc, info.opid ← opid
4:   while true do
5:     Node* curr ← DO_LOCATEPRED(key)
6:     if ISNODEPRESENT(curr, key) then
7:       ret ← UPDATEINFO(curr, info, false)
8:     else
9:       Node* n ← new Node
10:      n.key ← key, n.info ← info
11:      ret ← DO_INSERT(n)
12:    if ret = success then
13:      return true
14:    else if ret = fail then
15:      return false
```

Algorithm 7 Template for Transformed Find Function

```
1: function FIND(int key, Desc* desc, int opid)
2:   NodeInfo* info ← new NodeInfo
3:   info.desc ← desc, info.opid ← opid
4:   while true do
5:     Node* curr ← DO_LOCATEPRED(key)
6:     if ISNODEPRESENT(curr, key) then
7:       ret ← UPDATEINFO(curr, info, true)
8:     else
9:       ret ← fail
10:    if ret = success then
11:      return true
12:    else if ret = fail then
13:      return false
```

we omit detailed code listings, but express the general functionality specifications. Given a key, DO_LOCATEPRED returns the target node (and any necessary variables for linking and unlinking a node, e.g., its predecessor). DO_INSERT creates the necessary linkage to correctly place the new node in the data structure. DO_DELETE removes any references to the node. Note that some lock-free data structures [10, 18] employ a two-phased deletion algorithm, where the actual node removal is delayed or even separated from the first phase of logical deletion. In this case, we only expect DO_DELETE to perform the logical deletion as nodes will be physically removed during the next traversal.

Algorithm 6 lists the template for the transformed INSERT function. The function resembles the base node insertion algorithm with a CAS-based while loop (line 6.4). The only addition is the code path for invoking UPDATEINFO on line 6.7. The logic is simple: on line 6.6 we check if the data structure already contains a node with the target key. If so, we try to update the node's logical status, otherwise we fall back to the base code path to insert a new node. Should any of the two code paths indicate a retry due to contention, we start the traversal over again.

The DELETE operation listed in Algorithm 8 is identical to INSERT except it terminates with failure when the target node does not exist (line 8.13). We also adopt a two-phase process for unlinking nodes from the data structure: deleted nodes will firstly be buffered in a local set in EXECUTEOPS, and when the transaction commits, the $info$ field of buffered

Algorithm 8 Template for Transformed Delete Function

```
1: function DELETE(int key, Desc* desc, int opid,
   ref Node* del)
2:   NodeInfo* info ← new NodeInfo
3:   info.desc ← desc, info.opid ← opid
4:   while true do
5:     Node* curr ← DO_LOCATEPRED(key)
6:     if ISNODEPRESENT(curr, key) then
7:       ret ← UPDATEINFO(curr, info, true)
8:     else
9:       ret ← fail
10:    if ret = success then
11:      del ← curr
12:      return true
13:    else if ret = fail then
14:      del ← NIL
15:      return false

16: function MARKDELETE(set delnodes, Desc* desc)
17:   for del ∈ delnodes do
18:     if del = NIL then
19:       continue
20:     NodeInfo* info ← del.info
21:     if info.desc ≠ desc then
22:       continue
23:     if CAS(del.info, info, SETMARK(info)) then
24:       DO_DELETE(del)
```

nodes will be marked (line 8.23) and consequently unlinked from the data structure by invoking the base data structures' DO_DELETE. One advantage of our logical status interpretation is that unlinking nodes from the data structure is optional, whereas in transactional boosting nodes must be physically unlinked to restore the abstract state. This opens up opportunities to optimize performance based on application scenarios. Timely unlinking deleted nodes is important for linked lists because the number of "zombie" nodes has linear impact on sequential search time. Leaving delete nodes in the data structure may be beneficial for skiplists because the overhead and contention introduced by unlinking nodes may outweigh the slight increase in sequential search time ($\mathcal{O}(logn)$).

The FIND operation listed in Algorithm 7 also needs to update the node's $info$ pointer. Without this, concurrent deletions may remove the node after FIND has returned and before the transaction commits. Since we have extracted the core functionality of interpreting and updating logical status into a common subroutine, the transformation process is generic and straightforward. To use a transformed data structure, the application should first initialize and fill a DESC structure, then invoke EXECUTETRANSACTION (Algorithm 5) with it as an argument. The allocation of the transaction descriptor contributes to most of the transaction execution overhead. We discuss this in more details in Section 5.

4. CORRECTNESS

We base our correctness discussion on the notion of commutativity isolation [21], which states that the history of committed transactions is *strictly serializable* for any transactional data structure that provides linearizable operations

and obeys commutativity isolation [6]. STM systems may prefer more strict correctness criteria, such as opacity [16], because they need to account for the consistency of intermediate memory access. In the case of data structure transactions, the intermediate computation is managed by linearizable methods, and only the end result of a transaction is accessible to users. Strict serializability [33], which is the analogue of linearizability [25] for transactions, provides enough semantics for such cases.

4.1 Definitions

We provide a brief recapitulation of the definitions and correctness rules from Herlihy and Koskinen's work [21]. A *history* of computation is a sequence of instantaneous events. Events associated with a method call include invocation I and response R. A single transaction running in isolation defines a *sequential history.* A *sequential specification* for a data structure defines a set of *legal histories* for that data structure.

Definition 1. *A history h is strictly serializable if the subsequence of h consisting of all events of committed transactions is equivalent to a legal history in which these transactions execute sequentially in the order they commit.*

Definition 2. *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.*

Commutativity identifies operations that have no dependencies on each other. Executing commutative operations in any order yields the same abstract state. The commutativity specification for set operations is as follows:

$$\begin{aligned} \textsc{Insert}(x) &\leftrightarrow \textsc{Insert}(y),\ x \neq y \\ \textsc{Delete}(x) &\leftrightarrow \textsc{Delete}(y),\ x \neq y \\ \textsc{Insert}(x) &\leftrightarrow \textsc{Delete}(y),\ x \neq y \\ \textsc{Find}(x) \leftrightarrow \textsc{Insert}(x)/false &\leftrightarrow \textsc{Delete}(x)/false \end{aligned} \tag{1}$$

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 $I_1, R_1 \in T_1$ 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.*

4.2 Serializability and Recoverability

We now show that lock-free transactional transformation meets the two above correctness requirements. We denote the concrete state of a set as an node set N. At any time, the abstract state observed by transaction T_i is $S_i = \{n.key \mid n \in N \wedge \textsc{IsKeyPresent}(n.info,\ desc_i)\}$, where $desc_i$ is the descriptor of T_i.

Linearizability requires that concurrent operations appear as if they took place instantaneously at some points between their invocations and responses. We show the transformed operations are linearizable by identifying their *linearization points.* Additionally, we use the notion of decision points and state-read points to facilitate our reasoning. The decision point of an operation is defined 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 defined as the atomic statement where the state of the dictionary, which determines the outcome of the decision point, is read.

Lemma 1. *The set operations* Insert, Delete, *and* Find *are linearizable.*

Proof. For the transformed Insert operation, the execution is divided into two code paths by the condition check on line 6.6. The code path on line 6.7 updates the existing node's logical status. Note that if the operation reports failure on line 4.12 and 4.14, no write operation will be performed to change the logical status of the node. The state-read point for the former case is when the previous transaction status is read from $oldinfo.desc.status$ on line 3.5. The state-read point for the later case is when the current transaction status is read from $info.desc.status$ on line 4.13. The abstract states S' observed by all transactions immediately after the reads are unchanged, i.e., $\forall i, S'_i = S_i$. For a successful logical status update, the decision point for it to take effect is when the CAS operation on line 4.15 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after the CAS is $i = d \implies S'_i = S_i \cup n.key$. For all other transactions $i \neq d \implies S'_i = S_i$. In all cases, the update of abstract states conforms to the sequential specification of the insert operation. The code path for physically adding linkage to the new node (line 6.11) is linearizable because the corresponding Do_Insert operation in the base data structure is linearizable.

The same reasoning process applies to the transformed Delete and Find operations because they share the same logical status update procedure with Insert. □

The commutativity isolation rule prevents operations that are not commutative from being executed concurrently.

Lemma 2. *Conflict detection in lock-free transactional transformation satisfies the commutativity isolation rule.*

Proof. As identified in Equation 1, two set operations commute if they access the different keys. Because of the one-to-one mapping from node to keys, we have $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. This means that two set operations commute if they access two different nodes. Let T_1 denotes a transaction that currently accesses node n_1, i.e., $n_1.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 were to access n_1, it must perform ExecuteOps for T_1 on line 4.7 because ExecuteOps always update the transaction status when it returns on line 5.26 or 5.29 (note that failed CAS also means the transaction status has been set, but another thread). We thus ensure that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. □

Theorem 1. *For a data structure generated by lock-free transactional transform, the history of committed transactions is strictly serializable.*

Proof. Follow Lemma 1, and 2, and the conclusion in Herlihy and Koskinen's work [21], the theorem holds. □

[6] We omit the discussion of rules on compensating actions and disposable methods because they are not applicable to our approach

Theorem 2. *For a data structure generated by lock-free transactional transformation, any history defines the same abstract state as a history with aborted transactions removed.*

Proof. Let T_1 be an aborted transaction with descriptor $desc_1$. We denote S as the abstract state immediately after T_1 aborts. Let history $h = F_1 \cdot F_1' \cdots F_2 \cdot F_2' \cdot F_x \cdots F_y'$ be the sequence of linearizable method calls after T_1 starts and until T_1 aborts, where $F_i, 1 \leq i \leq x$ denotes the method calls successfully executed by T_1, and $F_i', 1 \leq i \leq y$ denotes the method calls executed by other transactions. The interleaving of these method calls is arbitrary. Follow commutativity isolation in Lemma 2 we assure that the method calls after F_x must commute with F_x, thus we can swap them without changing the abstract state. By progressively doing this for $F_i, 1 \leq i \leq x$, we obtain an equivalent history $h = h' = F_1' \cdots F_2' \cdots F_y' \cdots F_1 \cdot F_2 \cdots F_x$. Let n_x be the node accessed by F_x we denote S' as the abstract state before the invocation of F_x. Because of the inverse interpretation of logical status we can assert $n_x.key \in S' \implies n_x.key \in S \wedge n_x.key \notin S' \implies n_x.key \notin S$. Thus, we have $S' = S$ and we can remove F_x from h' without altering the abstract state. Doing this for $F_i, 1 \leq i \leq x$, we obtain $h = h' = h'' = F_1' \cdots F_2' \cdots F_y' \cdots$, hence T_1 is removed from the history. □

4.3 Progress Guarantees

Lock-free transaction transform provides lock-free progress because it guarantees that for every possible execution scenario, at least one thread makes progress in finite steps by either committing or aborting a transaction. We reason about this property by examining unbounded loops in all possible executions paths, which can delay the termination of the operations. For a system with i threads, the upper bound of the number of active transactions is i. Consider the while loop that executes the operations on line 5.13. This loop is bounded by the maximum number of operations in a transaction, denoted as j, but threads may set out to help each other during the execution of each of the operations. The number of recursive helping invocations is bound by the number of active transactions. In the worst case where only 1 thread remains live and $i - 1$ threads have failed, the system guarantees a transaction will commit in at most $i * j$ steps. In the presence of cyclic dependencies among transactions, the system guarantees that a duplicate transaction descriptor will be detected within $i * j$ steps.

5. PERFORMANCE EVALUATION

We compare the overhead and scalability of our lock-free transactional list and skiplist against the implementations based on transaction boosting, NOrec STM from Rochester Software Transactional Memory package [29] and Fraser's lock-free object-based STM [10]. RSTM is the best available comprehensive suite of prevailing STM implementations. In our test, TML [5] and its extension NOrec [5] are among the fastest on our platform. They have extreme low overhead and good scalability due to elimination of ownership records. We choose NOrec as the representative implementation because its value-based validation allows for more concurrency for readers with no actual conflict.

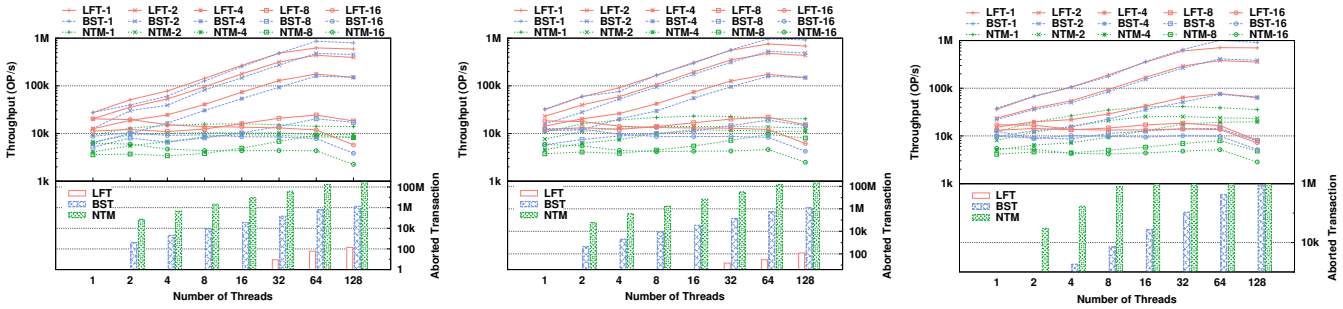
For transaction boosting, we implement the lookup of abstract lock using Intel TBB's concurrent hash map. Although the transaction boosting is designed to be used in tandem with STMs for replaying undo logs, it is not necessary in our test case as the data structures are tested in isolation. To reduce the runtime overhead, we scrap the STM environment and implement a lightweight per-thread undo log for the boosted data structures. We employ a micro-benchmark to evaluate performance in three types of workloads: write dominated, read dominated, and mixed. This canonical evaluation method [5, 18] consists of a tight loop that randomly chooses to perform a fixed size transaction with a mixture of INSERT, DELETE and FIND operations according to the workload type. We also vary the transaction size (i.e., the number of operations in a transaction) from 1 to 16 to measure the performance impact of rollbacks. The tests are conducted on a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1 GHz). Both the micro-benchmark and the data structure implementations are compiled with GCC 4.7 with C++11 features and `O3` optimizations. [7]

5.1 Transactional List

We show the throughput and the number of spurious aborts in Figure 2 for all three types of lists. The throughput is measured in terms of number of completed operations per second, which is the product of the number of committed transactions and transaction size. The number of spurious aborts takes into account the number of aborted transactions except self-aborted ones (i.e., those that abort due to failed operations). This is an indicator for the effectiveness of the contention management strategy. Each thread performs 10^5 transactions and the key range is set up to 10^4. Our lock-free transactional list is denoted by LFT, the boosted list by BST, and the NOrec STM list by NTM. The underlying list implementations for both LFT and BST were based on Harris' lock-free design [18]. The linked list for NTM is taken directly from the benchmark implementation in RSTM suite. Since the lock-free list and the RSTM list use different memory management scheme, we disable node reclamation for fair comparison of the synchronization protocols. For each list legend annotation, we append a numeric postfix to denote the transaction size (e.g., BST-4 means the boosted list tested with 4 operations in one transaction).

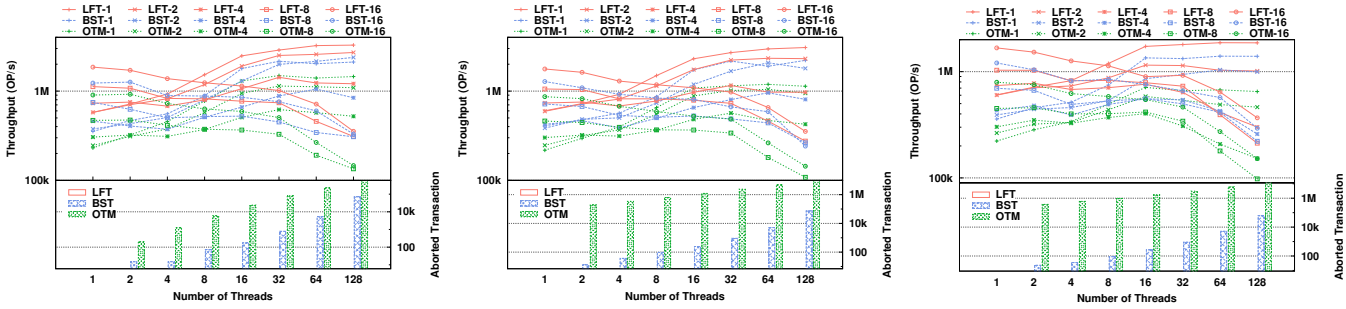
In Figure 2a, threads perform solely write operations. The upper half of the graph shows the throughput with both y- and x-axes in logarithm scale. Starting with transaction of size 1, the throughput curve of BST-1 and LFT-1 essentially expose the overhead difference between the two transaction synchronization protocols. Because each transaction contains only one operation, the code paths for transaction rollback in BST and transaction helping in LFT will not be taken. For each node operation, BST-1 needs to acquire and release a mutex lock, while LFT-1 needs to allocate a transaction descriptor. For executions within one CPU chip (no more than 16 threads), LFT-1 maintains a moderate performance advantage to BST-1, averaging more than 15% speedup. As the execution spawns across multiple chips, LFT-1's performance is setback by the use of descriptor, which incurs more remote memory accesses. This trend can be observed for all scenarios with different transaction sizes. Another noticeable trend is that LFT lists gain better performance as the transaction size grows. For example, on 64 threads the throughput of LFT-2 slightly falls short

[7] All source code can be downloaded from http://cse.eecs.ucf.edu/gitlab/deli/transactional-linked-data-structure



(a) 50% Insert, 50% Delete, 0% Find (b) 33% Insert, 33% Delete, 34% Find (c) 15% Insert, 5% Delete, 80% Find

Figure 2: Throughput and Spurious Abort Counts for Transaction Lists (10K Key Range)



(a) 50% Insert, 50% Delete, 0% Find (b) 33% Insert, 33% Delete, 34% Find (c) 15% Insert, 5% Delete, 80% Find

Figure 3: Throughput and Spurious Abort Counts for Transaction Skiplists (1M Key Range)

behind that of BST-2, then the performance of LFT-4 is on par with BST-4, and finally LFT-8 and LFT-16 outperforms their BST counterpart by as much as 50%. Two factors contribute to the great scalability of LFT lists in handling large transactions: 1) its helping mechanism manages conflict and greatly reduces spurious aborts whereas in BST such aborts cause a significant amount of rollbacks; 2) the number of allocated transaction descriptors decreases as the transaction size grows whereas in BST the number of required lock acquisitions stays the same.

Generally, we observe that for small transaction sizes (no more than 4 operations), BST and LFT lists explore fine-grained parallelism and exhibit similar scalability trends. The throughput increases linearly until 16 threads, and continues to increase at a slower pace until 64 threads. Because executions beyond 16 threads span across multiple chips, the performance growth is slightly reduced due to the cost of remote memory accesses. The executions are no longer fully concurrent beyond 64 threads, thus the overall throughput is capped and may even reduce due to context switching overhead. LFT lists obtain an average of 25% speedup over BST lists. For large transactions, the throughputs of both LFT and BST list do not scale well. This could be attributed to the semantic and the randomness of the benchmark. As the transaction size grows, the probability of a randomly generated sequence of operations will all succeed is inherently smaller. Most of the transactions were self-aborted due to some failed attempts to locate the target element. LFT lists outperforms BST lists by an average of 40% in these scenarios. On the other hand, the throughput of all NTM lists stagnates as the number of threads increases. Since NTM uses a single writer lock, concurrency is precluded for this write-dominated test case. On 64 threads, both BST and LFT lists are able to achieve as much as 10 times better performance than NTM lists.

On the bottom half of Figure 2a, we illustrate the histogram of spurious aborts across all transaction sizes and cluster them by thread counts. The y-axis is in logarithmic scale. For BST lists and NTM lists the number of spurious aborts grows linearly with the increase of threads. BST lists have about 100 times less aborts than NTM lists, which matches our intuition that semantic conflict detection can remarkably reduce the number of false conflicts. Also as expected no approach incurs spurious aborts in single thread scenario. Remarkably, LFT lists do not introduce spurious aborts until 32 threads, and the number of aborts is 4 orders of magnitude smaller than that of BST lists. The helping mechanism of LFT is able to resolve majority of the conflicts in a cooperative manner, and aborts only when cyclic dependencies exist among transactions.

We show the results from mixed and read-dominated workloads in Figure 2b and 2c. The throughputs follow the same pattern as in Figure 2a with LFT lists' performance advantage slightly diminishes in read-dominated workload. This is because the Find operations in LFT lists also update descriptors in nodes, which requires extra cycles compared with read-only BST Find operations. Nevertheless, LFT lists still maintain an average of 20% speedup over BST lists in these scenarios and achieves as much as 40% throughput gain for large transactions. Because of allowing reader concurrency, NTM lists also exhibit some degree of scalability in read-dominated scenarios.

5.2 Transactional Skiplist

In Figure 3, we show the throughput and the number of spurious aborts for three types of transactional skiplists. All of them are based on Fraser's open source lock-free skiplist [10], and the epoch-based garbage collection is enabled in all three. The naming convention for BST and LFT skiplists remain the same as in Figure 2. We denote the STM based skiplist as OTM because it uses Fraser's object-based STM. Compared with word-based STMs, an object-based STM groups memory into blocks thus reducing metadata overhead. Since skiplists have logarithmic search time, we are able to stress the algorithms with heavier workloads: each threads now performs 1 million transactions and the key range is also boosted to 1 million.

Overall, with a peak throughput of more than 3 million (OP/s), transaction execution on skiplists is considerably more efficient than on linked lists. Also both OTM and BST skiplists generates 2 orders or magnitude less spurious aborts than their list counterparts. Because skiplist algorithms traverse exponentially less nodes than list algorithms, a single operation can finish much sooner, which greatly reduces the probability of memory access conflicts in STM and lock acquisition time out in transaction boosting. Another noteworthy difference is that the divergent scalability trends of large and small transactions. As we can see in Figure 3a, large transaction such as LFT-8 and LFT-16 achieves maximum throughputs on a single thread, then their throughputs steadily fall as the number of threads increases. On the contrary, the throughputs of small transactions such as LFT-2 and LFT-4 start low, but gain momentum as more threads are added. Large transactions have lower synchronization overhead but are vulnerable to conflict. As the number of threads increases, a failed large transaction could easily forfeit a considerable amount of operations. On the flip side, small transactions incur greater synchronization overhead, but are less likely to encounter conflicts when more threads contend with each other.

Despite the differences, we still observe performance results generally similar to that of transaction lists. LFT and BST skiplists outperform OTM skiplists by as much as 3 times across all scenarios, while LFT skiplists maintain an average of 60% speedup over BST for large transactions. For example, in Figure 3a on 32 threads LFT-8 outperforms BST-8 by 125%. Even for small transactions, LFT skiplists begin to set the throughput apart from BST skiplists further than what is in Figure 2. For example, in Figure 3b on 32 threads LFT-2 achieves an 30% speedup over BST-2.

6. RELATED WORK

Non-blocking linked data structures have been extensively studied because their distributed memory layout provides data access parallelism and scalability under high levels of contention [18, 27, 31, 39]. To the best of our knowledge, there is no existing linked data structure that provides native support for transactions. A transactional execution of data structure operations can be seen as a restricted form of *software transactions* [17], in which the memory layout and the semantics of the operations are well defined according to the specification of the data structure. Straightforward generic constructions can be implemented by executing all shared memory accesses in coarse-grained *atomic sections*, which can employ either optimistic (e.g., STM) or pessimistic (e.g., lock inference) concurrency control. More sophisticated approaches [2, 12, 21] exploit semantic conflict detection for transaction-level synchronization to reduce benign conflicts. We draw inspirations from previous semantic based conflict detection approaches. However, with the specific knowledge on linked data structure, we further optimize the transaction execution by performing in-place conflict detection and contention management on existing nodes.

6.1 Transactional Memory

Initially proposed as a set of hardware extensions by Herlihy and Moss [23], transactional memory was intended to facilitate the development of lock-free data structures. However, due to current HTM's cache-coherency based conflict detection scheme, transactions are subject to spurious failures during page faults and context switches [6]. This makes HTM less desirable for data structure implementations. Considerable amount of work and ingenuity has instead gone into designing lock-free data structures using low-level synchronization primitives such as CompareAndSwap, which empowers researchers to devise algorithm-specific fine-grained concurrency control protocol.

The first software transactional memory was proposed by Shavit and Touitou [37], which is lock-free but only supports a static set of data items. Herlihy, et al., later presented DSTM [22] that supports dynamic data sets on the condition that the progress guarantee is relaxed to obstruction-freedom. Over the years, a large number of STM implementations have been proposed [5, 7, 10, 29, 35]. We omit complete reviews because it is out of the scope of this paper. As more design choices were explored [28], we have seen emerging discussions on the issues of STM regarding usability [34], performance [3], and expressiveness [15]. There is also an increasing realization that the read/write conflicts inherently provide insufficient support for concurrency when shared objects are subject to contention [26]. It has been suggested that "STM may not deliver the promised efficiency and simplicity for *all* scenario, and multitude of approaches should be explored catering to different needs" [1].

6.2 Lock Inference

STM implementations are typically *optimistic*, which means they execute under the assumption that interferences are unlikely to occur. They maintain computationally expensive redo or undo logs to allow replay or rollback in case a transaction experiences interference. In light of this shortcoming, pessimistic alternatives based on lock inference have been proposed [30]. These algorithms synthesize enough locks through static analysis to prevent data races in atomic sections. The choice of locking granularity has an impact on the trade-off between concurrency and overhead. Some approaches require programmers' annotation [11] to specify the granularity, others automatically infer locks at a fixed granularity [9] or even multiple granularities [4]. Nevertheless, most approaches associate locks with memory locations, which may lead to reduced parallelism due to false conflicts as seen in STM. Applying these approaches to real-world programs also faces scalability changes in the presence of large libraries [14] because of the high complexity involved in the static analysis process. Moreover, the use of locks degrades any non-blocking progress guarantee one might expect from using a non-blocking library.

6.3 Semantic Conflict Detection

Considering the imprecise nature of data-based conflict

detection, semantic-based approaches have been proposed to identify conflicts at a high-level (e.g., two commutative operations would not raise conflict even though they may access and modify the same memory data) which enables greater parallelism. Because semantically independent transactions may have low-level memory access conflicts, some other concurrency control protocol must be used to protect accesses to the underlying data structure. This results in a two-layer concurrency control. Transactional boosting proposed by Herlihy [21] is the first dedicated treatment on building highly concurrent transactional data structures using a semantic layer of abstract locks. The idea behind boosting is intuitive: if two operations commute they are allowed to proceed without interference (i.e., thread-level synchronization happens within the operations); otherwise they need to be synchronized at the transaction level. It treats the base data structure as a black box and uses *abstract locking* to ensure that non-commutative method calls do not occur concurrently. For each operation in a transaction, the boosted data structure calls the corresponding method of the underlying linearizable data structure after acquiring the abstract lock associated with that call. A transaction aborts when it fails to acquire an abstract lock, and it recovers from failure by invoking the inverses of already executed calls. Its semantic-based conflict detection approach eliminates excessive false conflicts associated with STM-based transactional data structures, but it still suffers from performance penalties due to the rollbacks of partially executed transactions. Moreover, when applied to non-blocking data structures, the progress guarantee of the boosted data structure is degraded because of the locks used for transactional-level synchronization. Transactional boosting is pessimistic in that it acquires locks eagerly before the method calls, but it still requires operation rollback because not all locks are acquired at once. Koskinen et al. [26] later generalized this work and introduce a formal framework called coarse-grained transactions. Bronson et al. proposed transaction prediction, which maps the abstract state of a set into memory blocks called predicate and relies on STM to synchronize transactional accesses [2]. Hassan et al. [20] proposed an optimistic version of boosting, which employs a white box design and provides throughput and usability benefits over the original boosting approach. Other STM variants, such open nested transactions [32] support a more relaxed transaction model that leverages some semantic knowledge based on programmers' input. The relaxation of STM systems and its implication on composability have been studied by Gramoli et al. [13]. The work by Golan-Gueta et al. [12] applies commutativity specification obtained from programmers' input to inferring locks for abstract data types.

7. CONCLUSION

We introduced a methodology for transforming lock-free linked data structures into high-performance lock-free transactional data structures. Our approach embeds the transaction metadata in each node, which enables resolving transaction conflicts cooperatively through thread-level synchronization. No undo logs nor rollbacks are needed because operations can correctly interpret the logical status of nodes left over by aborted transactions. Data structures that guarantees lock-free or weaker progress will be able to maintain their progress properties during the transformation. We demonstrated the application of our lock-free transactional transformation on two fundamental data structures: a linked list and a skiplist. The performance evaluation results show that our transaction synchronization protocol has low overhead and high scalability–providing an average of 40% speed-up over our good-faith transaction boosting implementations across all scenarios and as much as 125% more throughput for large transactions. The performance gain over STMs are even more substantial: more than 10 times over the alternative word-based STM and 3 times over the object-based STM. Besides the performance advantages, our approach decreases spurious aborts to a minimum, which is desirable because transaction success rate is a decisive factor for a majority of the applications.

We defer describing the methodology to encompass dictionary API to future work. The key change is to add two value fields (old, and current) in the NodeInfo structure so that the old version can be recovered if update fails. We also plan to further evaluate our approach on SMP systems to verify potential performance gains under low memory latency.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] H. Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 1–5. ACM, 2010.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 6–15. ACM, 2010.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40, 2008.

[4] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *ACM SIGPLAN Notices*, volume 43, pages 304–315. ACM, 2008.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[6] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. 2009.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[8] 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*, pages 131–140. ACM, 2010.

[9] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *ACM SIGPLAN Notices*, volume 42, pages 291–296. ACM, 2007.

[10] K. Fraser. *Practical lock-freedom*. PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[11] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. In *ACM SIGPLAN Notices*, volume 48, pages 263–274. ACM, 2013.

[12] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–41. ACM, 2015.

[13] V. Gramoli, R. Guerraoui, and M. Letia. Composing relaxed transactions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1171–1182. IEEE, 2013.

[14] K. Gudka, T. Harris, and S. Eisenbach. Lock inference in the presence of large libraries. In *ECOOP 2012–Object-Oriented Programming*, pages 308–332. Springer, 2012.

[15] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313. ACM, 2008.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[17] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[18] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[19] A. Hassan, R. Palmieri, and B. Ravindran. Integrating transactionally boosted data structures with stm frameworks: A case study on set. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[20] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *Principles of Distributed Systems*, pages 437–452. Springer, 2014.

[21] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

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

[25] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[26] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. *ACM Sigplan Notices*, 45(1):19–30, 2010.

[27] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*, pages 206–220. Springer, 2013.

[28] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7. ACM, 2004.

[29] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[30] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.

[31] 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*, pages 73–82. ACM, 2002.

[32] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

[33] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[34] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[35] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. 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*, pages 187–197. ACM, 2006.

[36] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. *ACM SIGPLAN Notices*, 46(10):51–64, 2011.

[37] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[38] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM, 1999.

[39] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE*

Transactions on Parallel and Distributed Systems,
PP(99):1–1, 2015.