Lock-Free Transactional Transformation for Linked Data Structures

DELI ZHANG, Microsoft
PIERRE LABORDE, LANCE LEBANOFF, and DAMIAN DECHEV,
University of Central Florida

Nonblocking data structures allow scalable and thread-safe access 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 separately from the underlying data structure's thread synchronization. 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 a transaction fails, we recover the correct abstract state by reversely interpreting the logical status of a node. We also present an obstruction-free version of our algorithm that can be applied to dynamic execution scenarios and an example of our approach applied to a hash map. In our experimental evaluation using transactions with randomly generated operations, our lock-free transactional data structures outperform the transactional boosted ones by 70% on average. They also outperform the alternative STM-based approaches by a factor of 2 to 13 across all scenarios. More importantly, we achieve 4,700 to 915,000 times fewer spurious aborts than the alternatives.

CCS Concepts: • **Computing methodologies → Concurrent algorithms**;

Additional Key Words and Phrases: Transactional data structure, lock-free, transactional memory, transactional boosting

ACM Reference format:
Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-Free Transactional Transformation for Linked Data Structures. *ACM Trans. Parallel Comput.* 5, 1, Article 6 (June 2018), 37 pages.
https://doi.org/10.1145/3209690

6

This work is supported by the National Science Foundation under the following grant numbers: NSF OAC 1440530 and NSF OAC 1740095.
Authors' addresses: D. Zhang, 1 Microsoft Way, Redmond, WA 98052; email: deli.zhang@outlook.com; P. Laborde, L. Lebanoff, and D. Dechev, 211 Harris Center (Bldg. 116), Dept. of Computer Science, University of Central Florida, 4000 Central Florida Blvd., Orlando, FL 32816 USA; emails: {pierrelaborde, lancelebanoff}@knights.ucf.edu, dechev@cs.ucf.edu.

https://doi.org/10.1145/3209690

1 INTRODUCTION

With the growing prevalence of multicore systems, numerous highly concurrent nonblocking data structures have emerged [8, 30, 34, 44]. 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 that 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 nonatomic composition of operations [39]. 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 nonblocking progress guarantees provided by some concurrent containers. Specifically, lock-based designs are vulnerable to deadlock, livelock, and thread starvation. In contrast, lock-free data structures are free from deadlock and livelock by definition of the lock-free progress guarantee—system-wide progress is guaranteed to occur in a finite number of steps in any possible execution.

The problem of implementing high-performance transactional data structures[4] is important and has recently gained much attention [2, 11–13, 19, 22, 27]. 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 (i) atomically (i.e., if one operation fails, the entire transaction should abort) and (ii) in isolation (i.e., concurrent executions of transactions appear to take effect in some sequential order).

Software Transactional Memory (STM) [23, 40] 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] [45]. 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

[1] http://libcds.sourceforge.net/.
[2] http://ucf-cs.github.io/Tervel/.
[3] https://www.threadingbuildingblocks.org/.
[4] Also referred to as *atomic composite operations* [11].

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 [22]. 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 article, 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—all sets whose contents are 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 [41]. 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 (LFTT) treats the base data structure as a *white box* and introduces a new code path for transaction-level synchronization using only the single-word CompareAndSwap (CAS) synchronization primitive. The two key challenges for high-performance data structure transaction executions are (i) to efficiently buffer write operations so that their modifications are invisible to operations outside the transaction scope and (ii) 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. Note that read-only transactions will cause conflicts, including with other read-only transactions, because our transaction synchronization acts at the node level. Otherwise, the operation helps execute the active transaction according to the instructions in the descriptor. As an alternative, we also present an aggressive contention management strategy in which an operation forcibly aborts competing transactions. We compare this obstruction-free version to our original lock-free version.

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 used *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 transactional transformation on a lock-free linked list, a lock-free skiplist, and a wait-free hash map 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 70% speedup over transactional boosting-based approaches, an average of 13 times speedup over the word-based STM, and an average of 2 times over the object-based STM. Moreover, the number of aborts generated by our approach are 4,700 times fewer than transactional boosting and 915,000 times fewer than STMs. Our transactional hash map has no spurious aborts for any tested scenario. We applied the obstruction-free version of our transactional transformation approach to the linked list and skiplist. The performance results were similar to those for the lock-free version of our approach.

This article 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.
- 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 article 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 3.1 and the template for building transactional hash maps in Section 3.2. 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 for constructing transactional data structures. We conclude the article 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, 22], 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: (i) node-based semantic conflict detection, (ii) interpretation-based logical rollback, and (iii) 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.

```
enum TxStatus{
  Active,
  Committed,
  Aborted
};
enum OpType{
  Insert,
  Delete,
  Find
};
struct Operation{
  OpType type;
  int key;
};
struct Desc{
  int size;
  TxStatus status;
  Operation ops[];
};
struct NodeInfo{
  Desc* desc;
  int opid;
};
struct Node{
  NodeInfo* info;
  int key;
  ...
};
```

Fig. 1. Type definitions.

```
int Mark = 0x1;
define SetMark(p) (p | Mark)
define ClearMark(p) (p & ~Mark)
define IsMarked(p) (p & Mark)
```

Fig. 2. Pointer marking.

For clarity, we list the constants and data type definitions in Figure 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* [25] 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: It stores all the necessary context for helping finish a delayed transaction, and 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 Figure 2. The *Mark* flag is co-located with the *info* pointers.

The NodeInfo and transaction descriptors necessarily increase the amount of memory that our program requires. For example, consider a linked list, which is known to have a worst-case space complexity of $O(n)$. When we add the NodeInfo descriptors to each node, we add a constant amount of space per node; this constant is subsumed by the definition of the notation. Therefore, the space complexity remains $O(n)$. Note that the NodeInfo descriptors can be recycled as soon as they are replaced with new ones by the compare-and-swap operation on Line 15 of Algorithm 2.

We are using one transaction descriptor per transaction executed by the application. These transaction descriptors are discarded after transaction execution. The amount of space used by each transaction descriptor is a constant amount for the *size* and *status*, plus a constant amount of space for each operation in the transaction. The size and number of transaction descriptors present

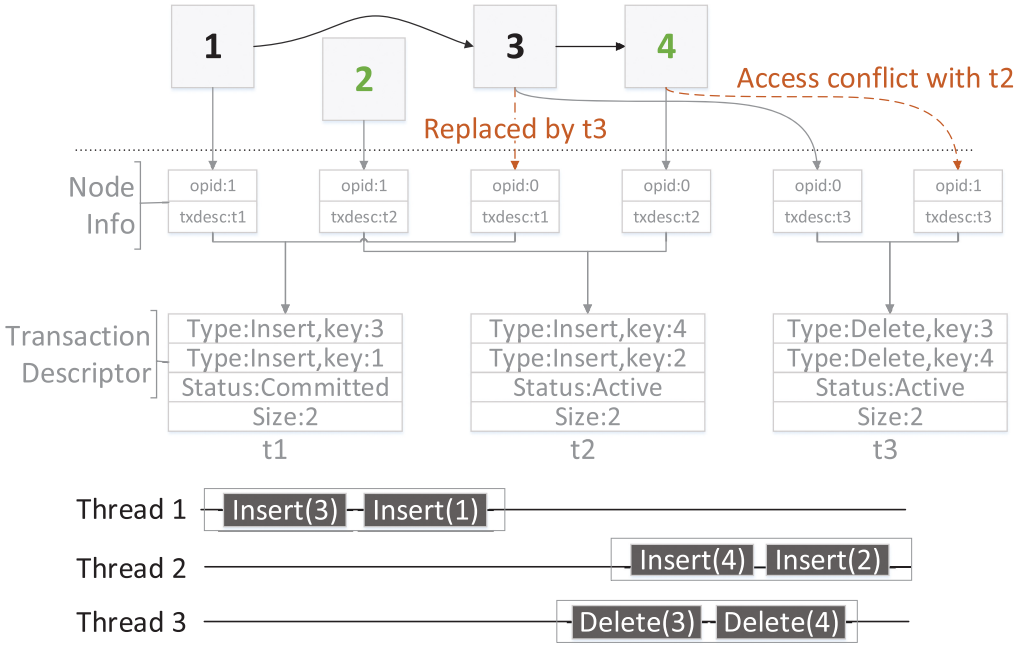


Fig. 3. Transaction execution and conflict.

in the data structure are limited by the square of the number of nodes in the data structure. In the worst case, every node would correspond to a thread that executes a transaction that performs an operation on every node: There would be n transactions of size n. We add the space complexity of these transaction descriptors to the space complexity that we found for the data structure with NodeInfo descriptors. This yields a worst-case space complexity for our transactional linked list of $O(n^2)$. In practice, a transaction size greater than 4 would be considered large, and the amount of memory used by descriptors would not be prohibitive in most cases.

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 [22]), 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.

We illustrate an example of node access conflict in Figure 3. At the beginning, Thread 1 committed a transaction t, 1 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 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 toward 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; 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 [22]. 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 1, 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. We need to have a way to determine whether or not a node with a certain key is reachable in the data structure, what we call a *physical node*, or a node that is physically present. This is separate from the idea of a node that is logically present because the physical node could be declared logically present or

ALGORITHM 1: Logical Status

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

ALGORITHM 2: 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.desc, oldinfo.opid + 1 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
```

not depending on the outcome of logical status interpretation. Function IsNodePresent tests for physical presence in the container and is used as a precursor to logical status interpretation. For example, if we are executing an Insert operation, then we need to know whether or not the key we are attempting to Insert already logically exists. However, before we are able to determine logical presence, we need to know if the key is physically present so that we can perform logical status interpretation on that physical node. The process of determining a key's logical presence is described later.

Given a node's physical 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 [22]. Since the Find operation is read-only, no rollback is needed. Therefore, a node which has most recently had a Find operation performed on it is always present regardless of 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. Note that IsNodePresent is called by Algorithm 2 on Line 10.

2.3 Logical Status Update

As mentioned earlier, 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 2. Given a node n, the function UpdateInfo reads its

ALGORITHM 3: Transaction Execution

```
1  thread_localStack helpstack
2  Function ExecuteTransaction(Desc* desc)
3      helpstack.Init()
4      ExecuteOps(desc, 0)
5      return desc.status = Committed
6
7  Function ExecuteOps(Desc* desc, int opid)
8      bool ret ← true
9      set delnodes
10     if helpstack.Contains(desc) then
11         CAS(&desc.flag, Active, Aborted)
12         return
13     helpstack.Push(desc)
14     while desc.status = Active and ret and opid < desc.size do
15         Operation* op ← desc.ops[opid]
16         if op.type = Find then
17             ret ← Find(op.key, desc, opid)
18         else if op.type = Insert then
19             ret ← Insert(op.key, desc, opid)
20         else if op.type = Delete then
21             Node* del
22             ret ← Delete(op.key, desc, opid, del)
23             delnodes.Insert(del)
24         opid ← opid + 1
25     helpstack.Pop()
26     if ret = true then
27         if CAS(&desc.flag, Active, Committed) then
28             MarkDelete(delnodes, desc)
29     else
30         CAS(&desc.flag, Active, Aborted)
```

current *info* field (Line 2.2[5]), verifies its sanity, and attempts to update $n.info$ through the use of CAS (Line 2.15). 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 2.7). 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 2.6. We also skip the update and report success if the operation has already been performed by other threads (Line 2.8). Due to the use of the helping mechanism, the same operation may be executed multiple times by different threads. The condition check on Line 2.8 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 2.10 and test if the key's presence (as required by deletions and finds) or lack of presence (as required by insertions) is desired on Line 2.11. The boolean flag

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

wantkey is passed on by the caller function to indicate if the presence of the key is desired. The operation reports failure when *wantkey* contradicts *haskey*. Finally, we validate that the transaction is still active (Line 2.13) 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 3 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 3.14), 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 that 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 3.27). 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 3.28), 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 that 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 3.10). We recover by aborting one of the transactions, as shown on Line 3.11.

2.5 Obstruction-Free Approach

Our cooperative contention management strategy requires that the users specify all operations in a transaction beforehand. However, some applications would benefit from dynamic transaction execution, in which the operations in a transaction are not predetermined prior to the execution of the transaction. In these applications, after a transaction begins, the operation types or operands of future method calls may depend on user input or other external events. Our helping scheme could possibly support dynamic transaction execution, but it would break the data structure's lock-free progress guarantee. For example, consider one transaction that begins by modifying node *n* and then waits on user input for its next operation. Other concurrent transactions that access *n* need to help that transaction before performing their own operations on *n*. This causes the concurrent transactions to be blocked (waiting on the user input) even though their own operations do not require user input.

To allow dynamic transaction execution, we introduce an alternative approach called *obstruction-free transactional transformation*, which eliminates the helping mechanism of the original lock-free approach. It instead adopts an aggressive contention management strategy. When two transactions conflict, the later transaction forcibly aborts the earlier one. In the previous example, the transaction that is waiting on user input may be aborted by concurrent transactions, causing it to restart. This allows the concurrent transactions to continue with their own operations

ALGORITHM 4: Obstruction-Free 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          CAS(&oldInfo.desc, Active, Aborted)
8      else if oldinfo.desc, oldinfo.opid + 1 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
```

without needlessly waiting on user input. In this case, the progress guarantee provided by the system degrades to obstruction-free. Obstruction-freedom guarantees that if a method executes without interruption from other threads, then it will finish in a finite number of steps.

Other contention management strategies are possible, such as having the thread able to also abort itself or aborting the transaction that has fewer completed operations. We can apply a large number of contention management scheme that are currently adopted by STMs. We chose aggressive contention management in our case as it is able to demonstrate the idea of obstruction-free execution without the clutter of complex contention management algorithms.

We modify the lock-free UpdateInfo function shown in Algorithm 2 to produce the obstruction-free UpdateInfo function listed in Algorithm 4. In both versions, a transaction examines the target node n to check if a conflicting transaction is also accessing n (Line 2.6 and Line 4.6). If so, a contention management strategy must be applied to ensure serializability. The lock-free version helps the conflicting transaction by executing that transaction's operations (Line 2.7). The obstruction-free version instead aborts the conflicting transaction if it is still active (Line 4.7).

We modify the methods dealing with transaction execution shown in Algorithm 3 to produce the obstruction-free methods listed in Algorithm 5. Because the helping mechanism can cause cyclic dependencies between transactions, the lock-free version must create and maintain a thread-local *help stack*, which contains the descriptors of the transactions that the thread is currently helping. Cyclic dependencies are detected by checking for duplicate transaction descriptors in the help stack (Line 3.10). The obstruction-free approach does not allow such cyclic dependencies to occur, so we remove the help stack.

3 APPLICATION EXAMPLES

In this section, we present the application of our methodology to linked lists, skiplists, and hash maps.

ALGORITHM 5: Obstruction-Free Transaction Execution

```
1  Function ExecuteTransaction(Desc* desc)
2      ExecuteOps(desc, 0)
3      return desc.status = Committed
4
5  Function ExecuteOps(Desc* desc, int opid)
6      bool ret ← true
7      set delnodes
8      while desc.status = Active and ret and opid < desc.size do
9          Operation* op ← desc.ops[opid]
10         if op.type = Find then
11             ret ← Find(op.key, desc, opid)
12         else if op.type = Insert then
13             ret ← Insert(op.key, desc, opid)
14         else if op.type = Delete then
15             Node* del
16             ret ← Delete(op.key, desc, opid, del)
17             delnodes.Insert(del)
18         opid ← opid + 1
19     if ret = true then
20         if CAS(&desc.flag, Active, Committed) then
21             MarkDelete(delnodes, desc)
22     else
23         CAS(&desc.flag, Active, Aborted)
```

3.1 Linked List- and Skiplist-Based Sets

In this section, we demonstrate the application of our lock-free transactional transformation on linked list- and skiplist-based sets. The process involves two steps: (i) identify and encapsulate the base data structure's methods for locating, inserting, and deleting nodes; and (ii) integrate the UpdateInfo function (Algorithm 2) 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, 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 2.4). 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.

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
```

The DELETE operation listed in Algorithm 7 is identical to INSERT except that it terminates with failure when the target node does not exist (Line 7.13). We also adopt a two-phase process for unlinking nodes from the data structure: Deleted nodes will first be buffered in a local set in EXECUTEOPS, and, when the transaction commits, the *info* field of buffered nodes will be marked (Line 7.24) and consequently unlinked from the data structure by invoking the base data structures' DO_DELETE unless the node has already been deleted (Line 7.19) or if the node has become part of another transaction (Line 7.22). The node could become part of another transaction because its key could be reinserted by another thread after the original transaction's status is compare-and-swapped to *committed* but before MARKDELETE is called.

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 of deleted nodes is important for linked lists because the number of "zombie" nodes has linear impact on sequential search time. Leaving deleted 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 ($O(logn)$).

The FIND operation listed in Algorithm 8 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 3) with it as an argument. The allocation of the

ALGORITHM 7: Template for Transformed Delete Function

```
1  Function Delete(int key, Desc* desc, int opid, 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
17 Function MarkDelete(set delnodes, Desc* desc)
18     for del ∈ delnodes do
19         if del = NIL then
20             continue
21         NodeInfo* info ← del.info
22         if info.desc ≠ desc then
23             continue
24         if CAS(del.info, info, SetMark(info)) then
25             Do_Delete(del)
```

ALGORITHM 8: 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
```

transaction descriptor contributes to most of the transaction execution overhead. We discuss this in more detail in Section 5.

3.2 Hash Maps

In this section, we demonstrate the application of our lock-free transactional transformation on hash maps. Map data structures store keys and their associated values. Maps also provide an update operation to change the value associated with a particular key, in addition to the insert, find, and remove operations that are present in set data structures. To support map data structures, we add a value field to the Operation and Node structs present in Figure 1 and an Update to the OpType enumeration.

The reason we make these changes to Figure 1 is to provide two different places to save a node's value: one in the node itself and one in the node's descriptor. We use these two locations to preserve the current value of a node and buffer pending updates in the node's descriptor. This allows Find operations from the same transaction to return the correct value, held in the descriptor, if the transaction commits without erroneously overwriting the value stored in the node by the most recently committed transaction. If we use the Find operation from Algorithm 8, we will erroneously overwrite pending updates as the Find operation will place its descriptor at a node, overwriting the node descriptor of the active Update operation. To solve this problem, we note that the node descriptor of a Find operation only needs to store the key that it is searching for and its operation type. In this case, the pending update can be preserved by copying its value from the old node descriptor of the Update operation to the node descriptor of the Find operation, which is now placed at the node. In this way, the pending writes to a node's value can be preserved without overwriting the node's current value, which would prevent inverse interpretation on transaction abort. This is an extension of logical status interpretation as we choose a different value depending on whether or not the transaction is aborted, active, or committed.

Our addition of value fields to the Operation and Node structs in Figure 1 allows us to perform logical status interpretation on key-value pairs as we will be able to recover the previous value associated with a key if an Update operation is aborted. We propagate this change to IsKeyPresent by treating the Update operation the same way we treat a Find as neither operation changes the presence of a key.

To perform logical status interpretation of a key-value pair, we implement an IsValuePresent function as the Update operation buffers writes to a node in the node's descriptor until it commits. The pseudocode for this function is presented in Algorithm 9. This function returns whether or not the value present in the Node should be treated as present; if not, the value in the NodeInfo descriptor is used because there is a pending update from the same transaction descriptor whose buffered write is logically interpreted as the node's value.

In the IsValuePresent function, INVALID represents a sentinel value that indicates a value has not been set for a Find operation. The semantics that we adhere to for a map data structure do not allow the user to search for a specific key-value pair; instead, the user searches for a key and the matching value is returned. We use the value field of a Find operation to hold pending updates buffered in a node's descriptor which would otherwise be erroneously overwritten by Find placing its own NodeInfo descriptor at that node.

We copy the pending updates, which are buffered in a node's descriptor, to the node in a lazy fashion. Once the transaction with the pending updates is committed, the next Find or Update operation that attempts to update the NodeInfo descriptor of that node will examine that node descriptor in order to determine whether or not the last operation that was performed was a committed Find or Update. If it sees a Find operation's descriptor that holds a different valid value from the node's current value, then the MapUpdateInfo algorithm will update the node's value.

If the operation sees an UPDATE descriptor at the node with any value other than that currently stored in the node, then the operation copies the value stored in the descriptor to the node. Once the operation completes the copy, it can perform its operation as usual. This copy preserves correct semantics for all operations as the old VALUE will be ignored by an INSERT or DELETE operation, and a FIND or UPDATE operation uses the new value in the node's VALUE field. The pseudocode for the MAPUPDATEINFO algorithm is displayed in Algorithm 10. The lazy update of the node's value occurs in the if-then statement on Line 10.17.

ALGORITHM 9: Logical Status for Maps

```
1  Function IsNodePresent(Node* n, int key)
2      return n.key = key
3
4  Function IsKeyPresent(NodeInfo* info, Desc* desc)
5      OpType op ← info.desc.ops[info.opid]
6      TxStatus status ← info.desc.status
7      switch status do
8          case Active do
9              if info.desc = desc then
10                 return op = Update or op = Find or op = Insert
11             else
12                 return op = Update or op = Find or op = Delete
13         case Committed do
14             return op = Update or op = Find or op = Insert
15         case Aborted do
16             return op = Update or op = Find or op = Find
17
18 Function IsValuePresent(NodeInfo* info)
19     Operation op ← info.desc.ops[info.opid]
20     if op.type == Update || op.type == Find && op.value! = INVALID then
21         return false
22     return true
```

As with the transactional linked list and skiplist, we encapsulate the base data structure's methods for locating, inserting, and deleting nodes. We describe the templates for each of the four canonical map operations INSERT, DELETE, FIND, and UPDATE. The only change we make to the templates for the INSERT, DELETE, and FIND operations, as shown in Algorithms 6, 7, and 8, is that we must set the VALUE field of the new node in an INSERT to the value associated with the key. No other changes are necessary as these operations do not examine the value associated with a key. Additionally, the underlying DO_LOCATEPRED algorithm must search for a node using the hashed key instead of the key itself. The template for the UPDATE function is similar to that of the INSERT function (except we call MAPUPDATEINFO with TRUE as the final argument) since the MAPUPDATEINFO function lazily updates the VALUE of a node.

4 CORRECTNESS

We base our correctness discussion on the main theorem of Herlihy and Koskinen's work [21], which states that the history of committed transactions is *strictly serializable* for any transactional

ALGORITHM 10: Update NodeInfo for Maps

```
1  Function MapUpdateInfo(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.desc, oldinfo.opid + 1 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     Operationop ← info.desc.ops[info.opid]
16     Operation oldOp ← oldinfo.desc.ops[oldinfo.opid]
17     if op.type == Update || op.type == Find then
18         if oldOp.value ! = node.value && oldinfo.desc.status ==
           Committed && IsValuePresent(oldinfo) then
19             n.value ← oldOp.value
20     if CAS(&n.info, oldinfo, info) then
21         if op.type == Find then
22             if oldOp.type == Update || (oldOp.type == Find && oldOp.value ! = INVALID) then
23                 n.info.value ← oldOp.value
24                 return n.info.value
25             else
26                 return n.value
27         return success
28     else
29         return retry
```

data structure that obeys the rules of linearizability, commutativity isolation, compensating actions, and disposable methods. We first provide definitions of these rules, then we prove that lock-free transactional transformation follows these rules and therefore guarantees strictly serializability.

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 [36], which is the analogue of linearizability [26] 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], some of which we have extended or modified. A *history* of computation is a sequence of instantaneous events. Every invocation I and response R is associated with a transaction T. Every *transaction* is a sequential execution of the method calls that represent the operations that the user requested be performed atomically. A single thread running in isolation, where each invocation is followed by the corresponding method call's response, defines a *sequential history*. A *sequential specification* for a data structure defines a set of *legal histories* for that data structure. A *concurrent history* is one in which events of different threads are interleaved.

Definition 4.1 (Strict Serializability). 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 4.2 (Commutativity). 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, as can be seen in Definition 4.2. The commutativity specification for set operations is listed in Equation (1):

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

Definition 4.3 (Inverse Operations). For a history h and any given invocation I and response R, let I^{-1} and R^{-1} be the inverse invocation and response. That is, the invocation and response such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reached after history h.

Definition 4.4 (Disposable Method Calls). For a history h, let G be the set of histories g such that $h \cdot g$ is legal. A method call denoted $I \cdot R$ is disposable if $\forall g \in G$; if $h \cdot I \cdot R$ and $g \cdot I \cdot R$ are legal, then $h \cdot I \cdot R \cdot g$ and $h \cdot g \cdot I \cdot R$ are legal and both define the same state.

The method call $I \cdot R$ is disposable if it can be postponed arbitrarily long without anyone being able to tell that $I \cdot R$ did not occur.

4.2 Rules

We now define the rules presented by Herlihy and Koskinen [21], which—if a transactional data structure follows them—guarantee that the data structure is stricly serializable.

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 noncommutative 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.*

Rule 3 (Compensating Actions). *For any history h which contains the abort of transaction T, then it must be the case that T executed the following operations: $I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1}$ where i indexes the last successfully completed method call.*

Rule 4 (Disposable Methods). *For any history h and transaction T, any method call invoked by T that occurs after T commits or aborts must be disposable.*

4.3 Serializability and Recoverability

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

Linearizability requires that concurrent operations appear as if they took place instantaneously at some point 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 set, which determines the outcome of the decision point, is read.

Lemma 4.5. *The set operations Insert, Delete, and Find are linearizable, satisfying Rule 1.*

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 either Line 2.12 or Line 2.14, no write operation will be performed to change the logical status of the node. The state-read point for the case where the operation reports failure on Line 2.12 occurs when the previous transaction status is read from *oldinfo.desc.status* on Line 1.6, which is called on Line 2.10. The state-read point for the case where the operation reports failure on Line 2.14 occurs when the current transaction status is read from *info.desc.status* on Line 2.13. 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 2.15 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after the CAS is $i = d \Rightarrow S'_i = S_i \cup n.key$. For all other transactions $i \neq d \Rightarrow S'_i = S_i$. 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 6.9) 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 and analagous underlying linearizable operations with Insert. □

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

Lemma 4.6. *Conflict detection in lock-free transactional transformation satisfies commutativity isolation as defined in Rule 2.*

Proof. As identified in Equation (1), two set operations commute if they access different keys. Because of the one-to-one mapping from node to keys, we have $\forall n_x, n_y \in N, x \neq y \Rightarrow n_x \neq n_y \Rightarrow n_x.key \neq n_y.key$. This means that two set operations commute if they access two different nodes. Let T_1 denote a transaction that currently accesses node n_1; that is, $n_1.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 were to access n_1, it must perform ExecuteOps for T_1 on Line 2.7 because ExecuteOps always updates the transaction status when it returns on Line 3.27 or 3.30 (note that failed CAS also means the transaction status has been set, by

another thread). We thus ensure that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. □

Lemma 4.7. *When a transaction aborts, the logical rollback mechanism of lock-free transactional transformation is equivalent to performing the inverses of completed operations, satisfying Rule 3.*

Proof. Let T denote a transaction that executes the operations $I_0 \cdot R_0 \cdots I_i \cdot R_i$ on nodes $n_0 \cdots n_i$ and then aborts. Let S_0 denote the abstract state of the data structure immediately before I_0, and S_i denote the abstract state immediately after R_i. Rule 3 requires that T executes the inverses of the successful method calls: $I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1}$ after those method calls have succeeded. This is equivalent to requiring that the abstract state be restored to its original state S_0.

In a data structure generated by lock-free transactional transformation, when T aborts, T's transaction descriptor status is changed from *Active* to *Aborted*. The IsKeyPresent function ensures that for each node n_x in $n_0 \cdots n_i$, the next operation that accesses n_x will interpret the current abstract state S_y to be equal to S_0. We now show that this is true for every possible operation $I_x \cdot R_x$ on a case-by-case basis for Insert, Delete, and Find.

Insert method. There are two cases for an Insert($n_x.key$) call. In the first case, $n_x.key \notin S_0$. The Insert method call places a new node n_x into the data structure (Line 6.9), which has a transaction descriptor pointing at T, or Insert changes the existing node's transaction descriptor field to point to T (Line 6.7). Then assume that the transaction, T, aborts some time after R_x, so T's transaction descriptor status is set to *Aborted*. The next operation that accesses n_x will follow the transaction descriptor field of n_x, observe T's descriptor status as *Aborted*, and logically interpret that $n_x.key \notin S_y$ (Line 1.16). Therefore, $S_y = S_0$.

In the second case, $n_x.key \in S_0$. The Insert method call does not perform any writes to the data structure, so it does not change the abstract state. Then assume that T aborts some time after R_x, so T's transaction descriptor status is set to *Aborted*. However, this action does not affect the abstract state regarding $n_x.key$, so $S_y = S_0$.

Delete method. There are two cases for a Delete($n_x.key$) call. In the first case, $n_x.key \in S_0$. The Delete method call changes the transaction descriptor field of the existing node n_x to point to T (Line 7.7). Then assume that T aborts some time after R_x, so T's transaction descriptor status is set to *Aborted*. The next operation that accesses n_x will follow the transaction descriptor field of n_x, observe T's descriptor status as *Aborted*, and logically interpret that $n_x.key \in S_y$ (Line 1.16). Therefore, $S_y = S_0$.

In the second case, $n_x.key \notin S_0$. This case is similar to the Insert case in which $n_x.key \in S_0$ because the Delete method does not perform any writes to the data structure in this case.

Find method. There are two cases for a Find($n_x.key$) call. In the first case, $n_x.key \in S_0$. The Find method call changes the transaction descriptor field of the existing node n_x to point to T (Line 7.7). Whether or not T aborts, the next operation that accesses n_x will always logically interpret that $n_x.key \in S_y$ (Line 1.16). Therefore, $S_y = S_0$.

In the second case, $n_x.key \notin S_0$. This case is similar to the Delete case in which $n_x.key \notin S_0$ because the Find method does not perform any writes to the data structure in this case. □

Lemma 4.8. *The MarkDelete method is disposable, so lock-free transactional transformation satisfies Rule 4.*

Proof. For a set generated by lock-free transactional transformation, MarkDelete is the only method that can be invoked by a transaction T after T has committed or aborted. We base our proof on the fact that the MarkDelete method does not change the abstract state of the data structure, so it can be postponed arbitrarily without anyone being able to tell that it did not occur, which implies that it is disposable.

We now prove that the MarkDelete method does not change the abstract state of the data structure. For MarkDelete to be called on a node n, it must have been invoked by a transaction T that called Delete(n) and then committed. Immediately after the Delete(n) and commit, $n.key \notin S$, where S is the abstract state of the data structure. Although n is still linked in the data structure, its key is not present in the abstract state. After MarkDelete(n) returns, n is no longer linked in the data structure. This step is handled by the underlying Do_Delete method of the base data structure. Because n is no longer linked in the data structure, $n.key \notin S$. Therefore, MarkDelete does not change the abstract state and so it is disposable. □

Theorem 4.9. *For a data structure generated by lock-free transactional transformation, the history of committed transactions is strictly serializable.*

Proof. Follow Lemmas 4.5, 4.6, 4.7, 4.8, and the main theorem in Herlihy and Koskinen's work [21], the theorem holds. □

Theorem 4.10. *For a data structure generated by lock-free transactional transformation, the complete history of transactions (including committed and aborted transactions) is strictly serializable.*

Proof. The Theorem of Aborted Transactions in Herlihy and Koskinen's work states that for any system that obeys all of the rules shown, any history defines the same abstract state as a history with aborted transactions removed. Following Theorem 4.9, and the Theorem of Aborted Transactions, then, for a data structure generated by lock-free transactional transformation, any history defines the same abstract state as a strictly serializable history. Therefore, any such history is strictly serializable. □

4.4 Progress Guarantees

Lock-free transactional transformation 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 3.14. 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 [32], 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 extremely 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 locks 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 light-weight 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 @2.1GHz). Both the micro-benchmark and the data structure implementations are compiled with GCC 4.7 with C++11 features and 03 optimizations.[6]

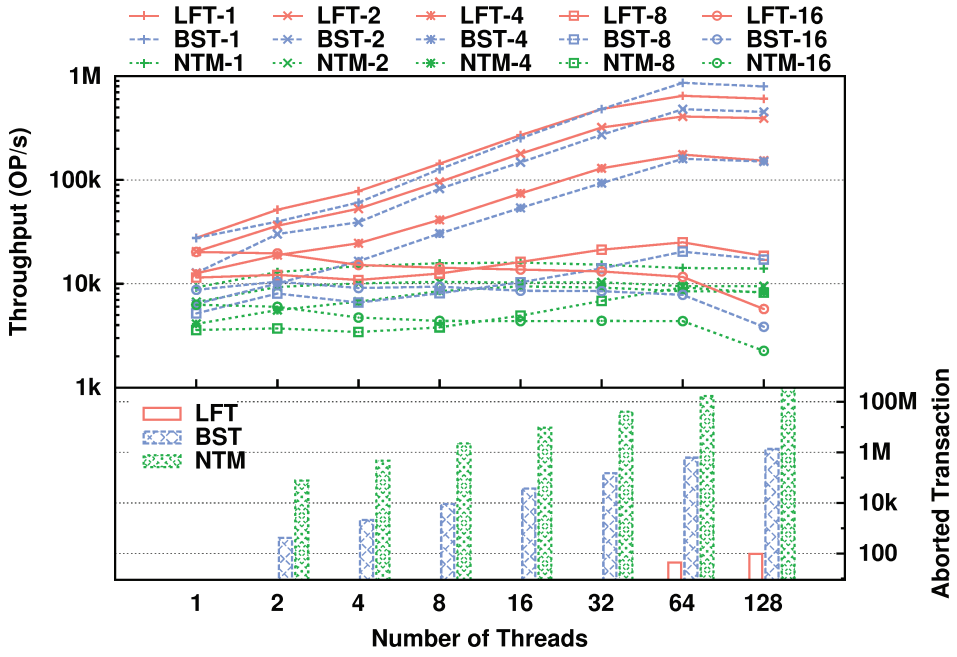
5.1 Transactional List

We show the throughput and the number of spurious aborts in Figure 4 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's lock-free design [18]. The linked list for NTM is taken directly from the benchmark implementation in the 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).

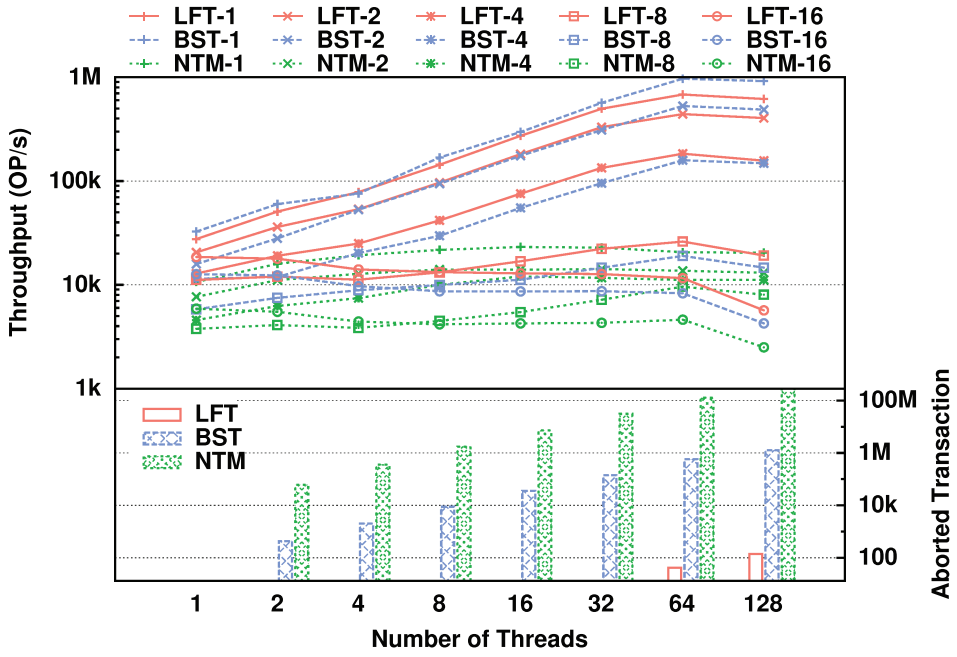
In Figure 4(a), threads perform solely write operations. The upper half of the graph shows the throughput with both y- and x-axes in logarithmic scale. Starting with a transaction of size 1, the throughput curve of BST-1 and LFT-1 essentially expose the overhead difference between the two transaction 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 over BST-1, averaging more than 13% speedup. As the execution spawns across multiple chips, LFT-1's performance is set back by the use of descriptors, which incur more remote memory accesses. 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 behind that of BST-2, then the performance of LFT-4 is on par with BST-4, and finally LFT-8 and LFT-16 outperform their BST counterpart by as much as 50%. Two factors contribute to the great scalability of LFT lists in handling large transactions: (i) Its helping mechanism manages conflict and greatly reduces spurious aborts whereas in BST such aborts cause a significant amount of rollbacks; and (ii) 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 four operations), BST and LFT lists explore fine-grained parallelism and exhibit similar scalability trends. The throughput

[6] All source code can be downloaded from https://github.com/ucf-cs/tlds.

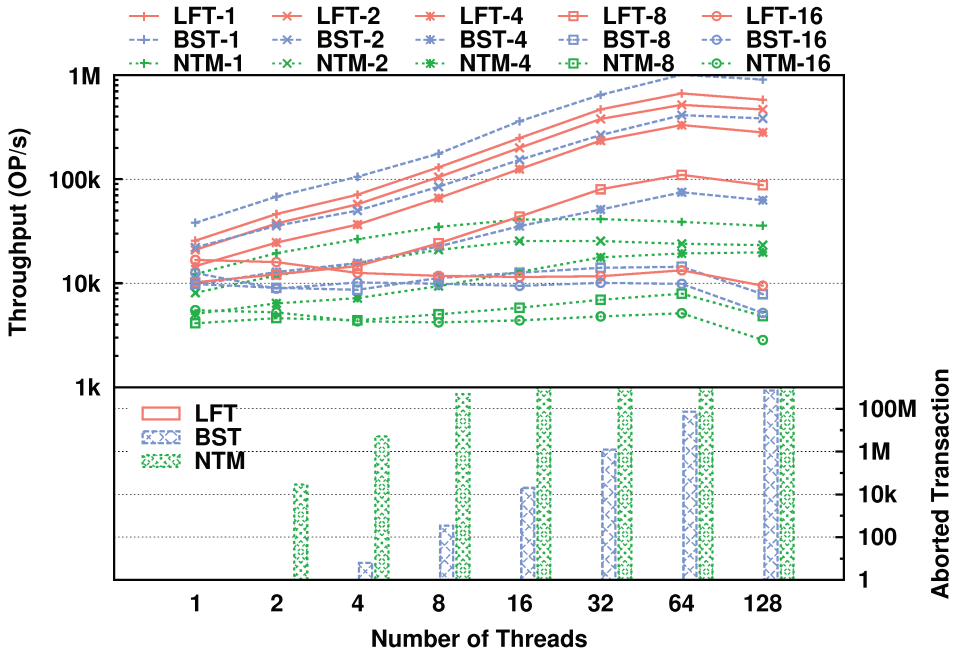


(a) Throughput and Spurious Abort Counts for Lists (10K Key Range): 50% Insert, 50% Delete, 0% Find



(b) Throughput and Spurious Abort Counts for Lists (10K Key Range): 33% Insert, 33% Delete, 34% Find

Fig. 4. Continued.



(c) Throughput and Spurious Abort Counts for Lists (10K Key Range): 15% INSERT, 5% DELETE, 80% FIND

Fig. 4. Throughput and spurious abort counts for lists (10K key range).

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. For large transactions, the throughputs of both LFT and BST lists do not scale well. This could be attributed to the semantics and the randomness of the benchmark. As the transaction size grows, the probability that all of a randomly generated sequence of operations will succeed is inherently smaller. Most of the transactions were self-aborted due to some failed attempts to locate the target element. LFT lists outperform BST lists by an average of 2 times 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 33 times better performance than NTM lists when averaged across all scenarios.

On the bottom half of Figure 4(a), 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 fewer 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 4,200 times smaller than that of the BST list. The helping mechanism of LFT is able to resolve a 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 4(b) and 4(c). The throughputs follow the same pattern as in Figure 4(a), with LFT lists' performance advantage slightly diminished in a 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. The LFT list achieves as much as 2.4 times throughput gain for large transactions over BST lists in these scenarios. Because of allowing reader concurrency, NTM lists also exhibit some degree of scalability in read-dominated scenarios.

5.2 Transactional Skiplist

In Figure 5, 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 4. 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.

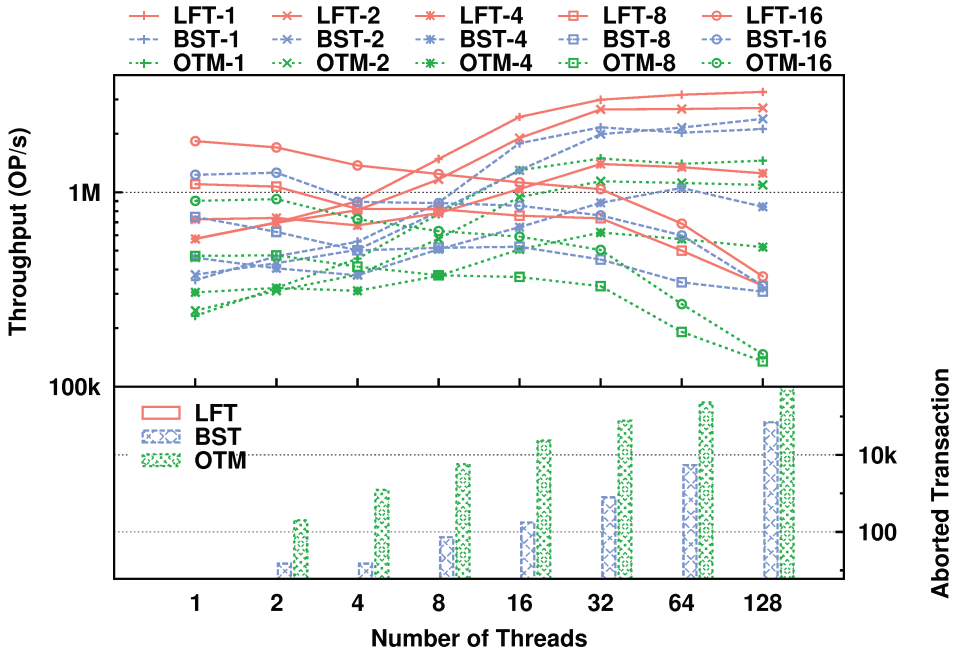
Overall, with a peak throughput of more than 3 million (OP/s), transaction execution on skiplists is considerably more efficient than on linked lists. OTM and BST skiplists generate 790 and 25 times fewer spurious aborts than their list counterparts, respectively. Because skiplist algorithms traverse exponentially fewer 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 the divergent scalability trends of large and small transactions. As we can see in Figure 5(a), large transactions such as LFT-8 and LFT-16 achieve maximum throughput 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 outperforms the OTM skiplist by as much as 2.7 times across all scenarios, and BST outperforms the OTM skiplist by as much as 2.3 times. For example, in Figure 5(a) on 32 threads LFT-8 outperforms BST-8 by 63%. Even for small transactions, LFT skiplists begin to set the throughput apart from BST skiplists further than what is in Figure 4. For example, in Figure 5(b) on 32 threads LFT-2 achieves a 62% speedup over BST-2.

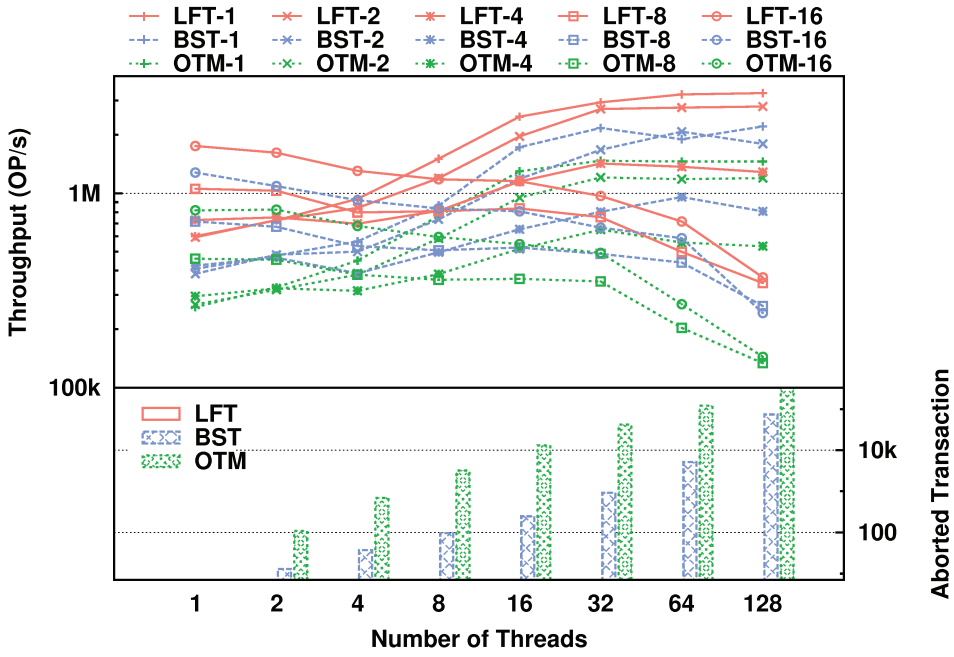
5.3 Transactional Map

In Figure 8, we show the throughput for the lock-free transactional hash map. The LFTT map, and the transaction boosting version that we compare to, are based on the wait-free hash map in La Borde et al. [28].

To test the LFTT map with a large workload, we apply the same evaluation procedure as in Section 5.2, giving each thread a workload of 1 million transactions and setting the key range to 1 million. As we can see in Figure 8(a), large transactions such as LFT-8 and LFT-16 achieve maximum throughput on a single thread, then their throughput steadily falls as the number of threads increases. This is the same behavior observed in Figure 5(a). The trend is weaker for the graph shown in Figure 8(c) because the 75% FIND operations lead to a greater number of operations

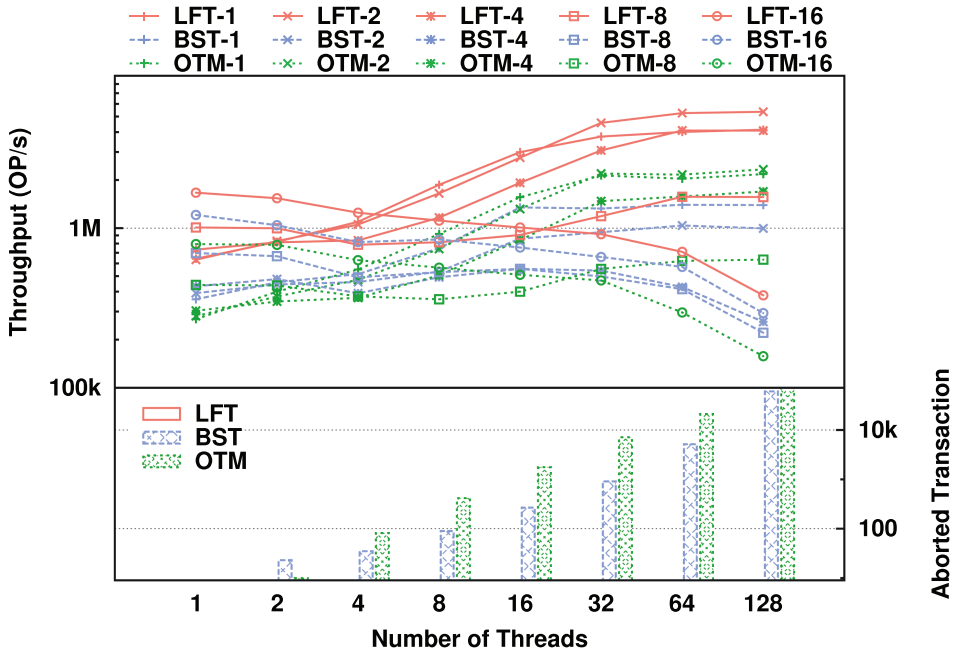


(a) Throughput and Spurious Abort Counts for Skiplists (1M Key Range): 50% Insert, 50% Delete, 0% Find



(b) Throughput and Spurious Abort Counts for Skiplists (1M Key Range): 33% Insert, 33% Delete, 34% Find

Fig. 5. Continued.



(c) Throughput and Spurious Abort Counts for Skiplists (1M Key Range): 15% INSERT, 5% DELETE, 80% FIND

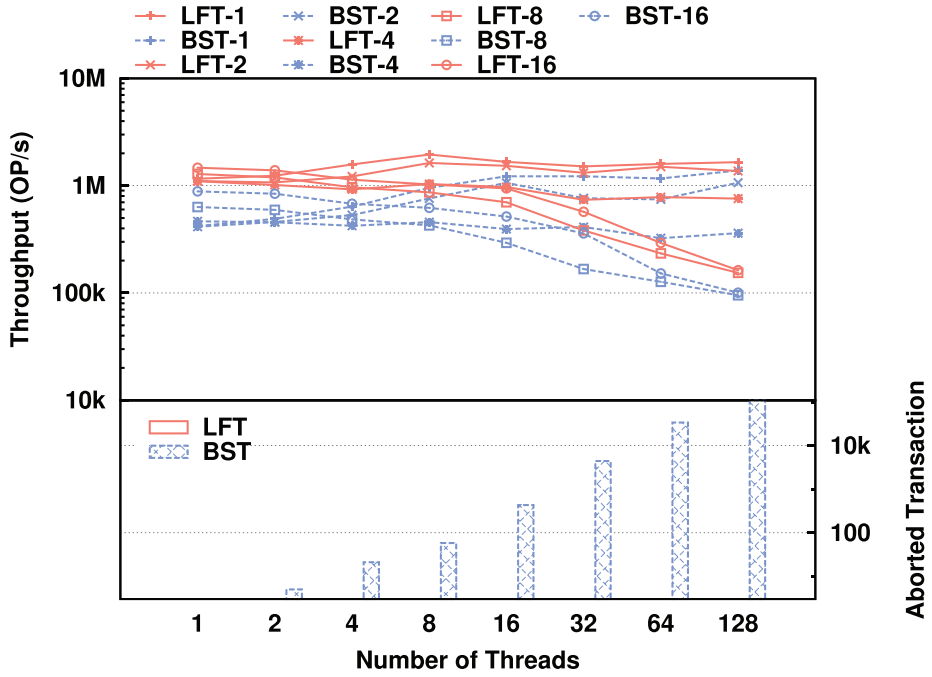
Fig. 5. Throughput and spurious abort counts for skiplists (1M key range).

in the larger transaction sizes committing without failing. An example of this is shown for the transaction size of 8 in Figure 8(c), which seems to level out compared to the other two operation distributions with the same transaction size. The transactional boosting version of the hash map follows the same trends in these cases but has lower performance due to executing transactions which must be rolled back when they abort. In comparison, our approach does not suffer from any spurious aborts in any of the tested scenarios. Overall, with a peak throughput of more than 2.6 million (OP/s), transaction execution on our hash map is considerably more efficient than on linked lists, and comparable to skiplists despite the extra overhead on the UPDATEINFO function due to the UPDATE operation. On average, the LFTT hash map is 74% faster than the transactional boosting version.

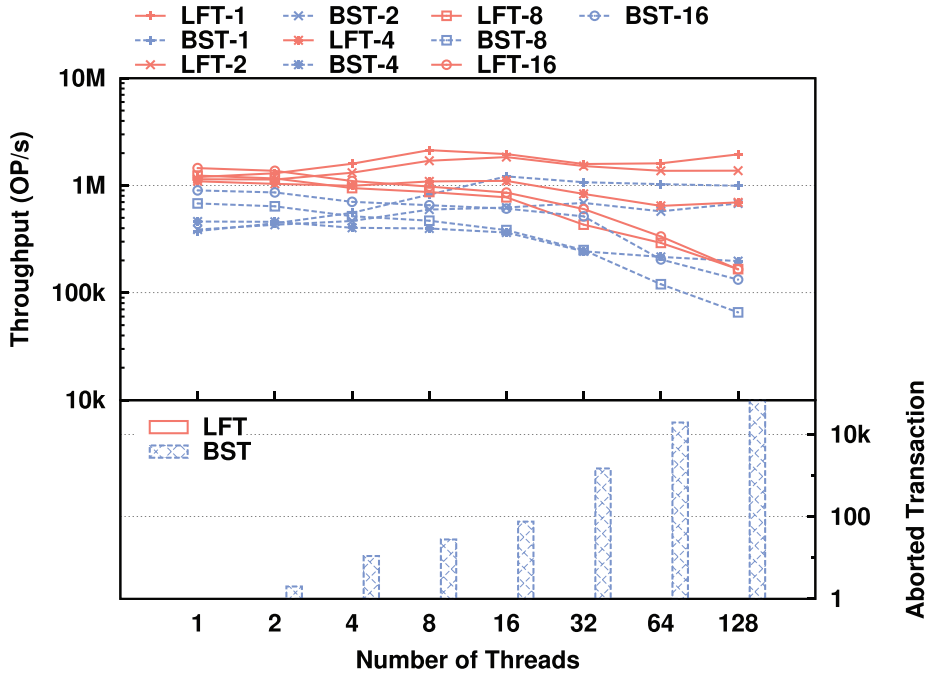
5.4 Obstruction-Free Performance

We compare the overhead and scalability of our obstruction-free version of transactional transformation against our lock-free version. The lock-free version is denoted by LFT, and the obstruction-free version is denoted OFT.

In Figure 6, we show the throughput and the number of spurious aborts for the LFT list and the OFT list. We apply the same evaluation procedure as in Section 5.1, giving each thread a workload of 10^5 transactions and setting the key range to 10^4. The OFT list performs similarly to the LFT list, with LFT outperforming OFT by merely 1.4% on average. However, with greater numbers of threads and larger transactions, OFT shows a performance disadvantage under higher contention. For 64 and 128 threads, LFT-8 and LFT-16 achieve an average of 2.5% speedup over OFT-8 and OFT-16. These scenarios increase the chance that two transactions will concurrently access the

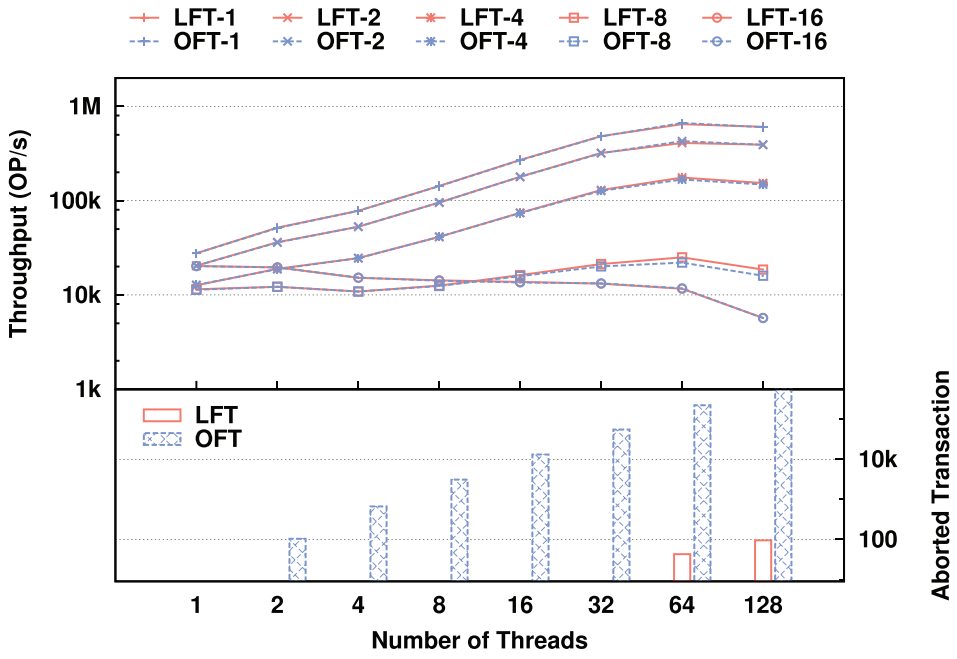


(a) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 50% Insert, 50% Delete, 0% Update, 0% Find

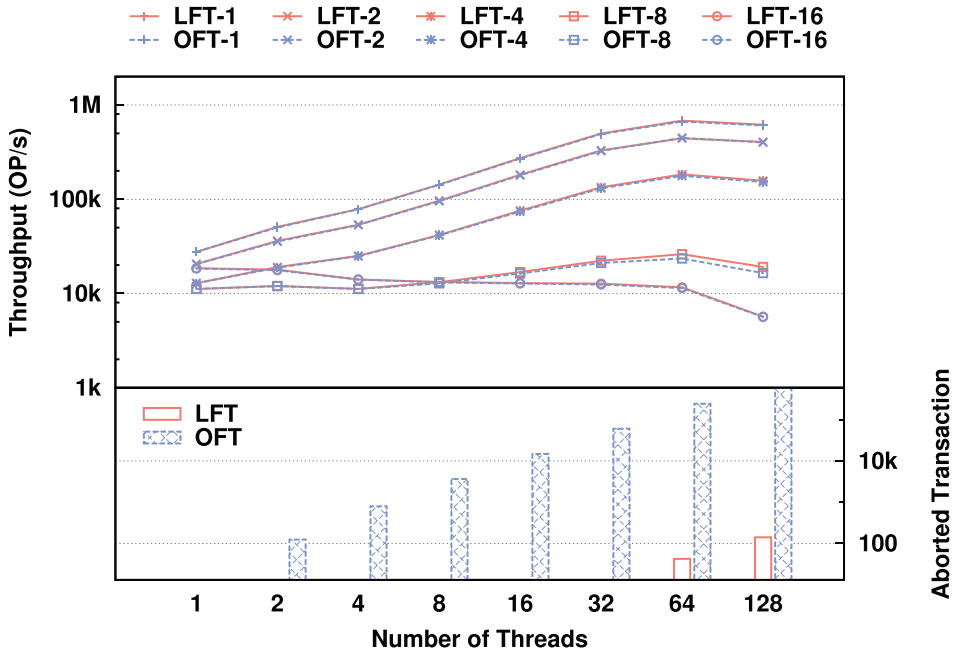


(b) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 25% Insert, 25% Delete, 25% Update, 25% Find

Fig. 8. Continued.

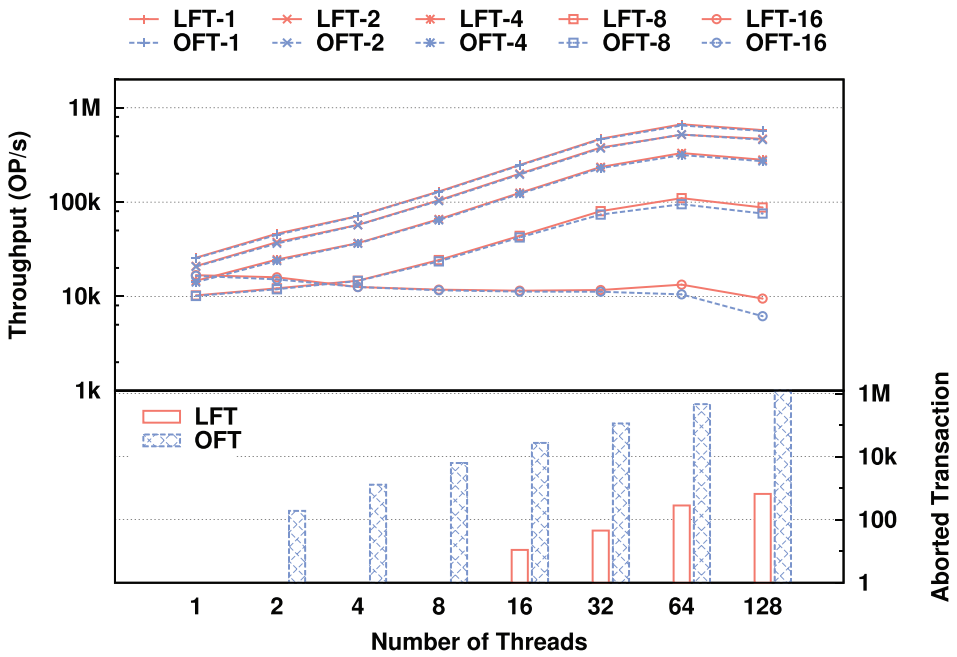


(a) Throughput and Spurious Aborts for LFT vs OFT Lists (10K Key Range): 50% Insert, 50% Delete, 0% Find



(b) Throughput and Spurious Aborts for LFT vs OFT Lists (10K Key Range): 33% Insert, 33% Delete, 34% Find

Fig. 6. Continued.



(c) Throughput and Spurious Aborts for LFT vs OFT Lists (10K Key Range): 15% Insert, 5% Delete, 80% Find

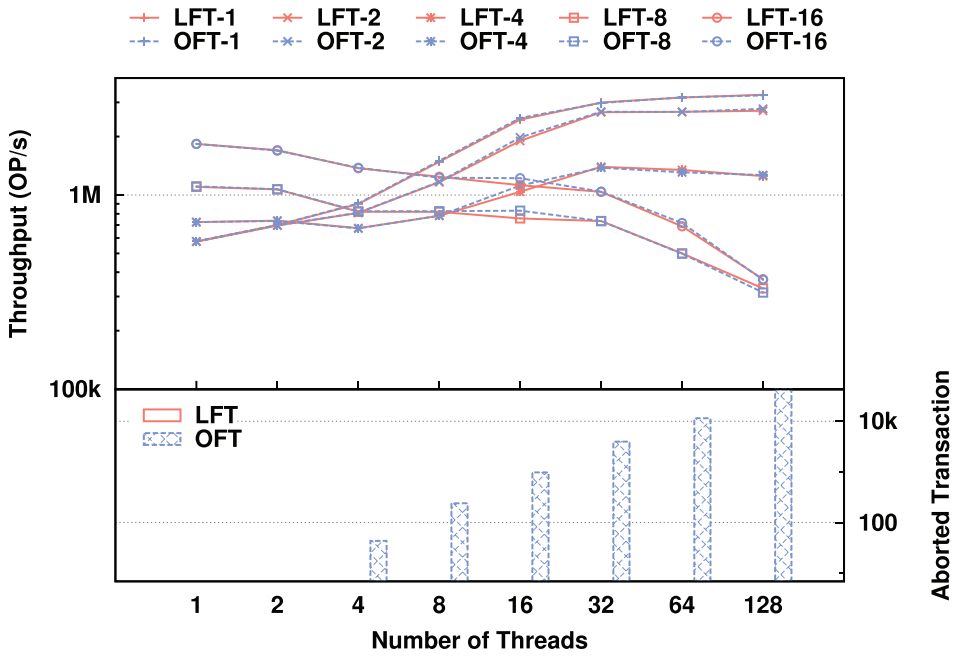
Fig. 6. Throughput and spurious abort counts for lock-free versus obstruction-free transactional lists (10K key range).

same node. In OFT, this situation will cause one of the transactions to spuriously abort, effectively canceling the progress that the aborted transaction made. In contrast, the helping mechanism of LFT manages this conflict in a cooperative manner, allowing every transaction to make progress.

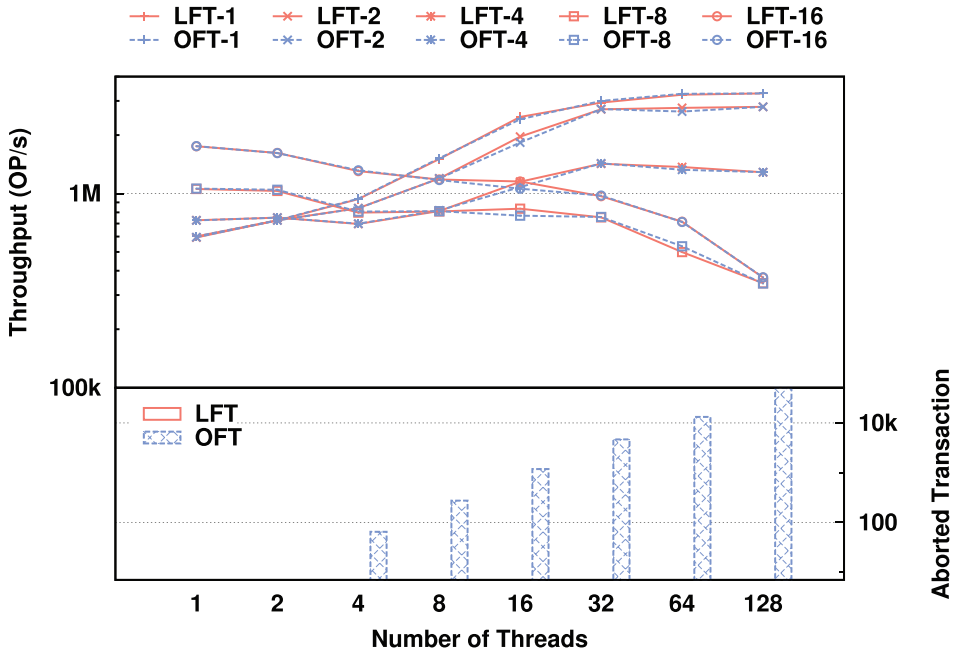
In Figure 7, we compare the performance of the OFT skiplist to the LFT skiplist. As in Section 5.2, we boost the workload to 1 million transactions per thread and increase the key range to 1 million. Because of the larger key range, the number of spurious aborts is greatly reduced. As a result, the OFT skiplist does not show the same performance disadvantage as the OFT list. In this low-contention environment, the LFT and OFT skiplists perform very similarly, only showing slight variations in throughput.

6 RELATED WORK

Nonblocking linked data structures have been extensively studied because their distributed memory layout provides data access parallelism and scalability under high levels of contention [18, 30, 34, 44]. 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, 22] exploit semantic conflict detection for transaction-level synchronization to reduce benign conflicts. We draw

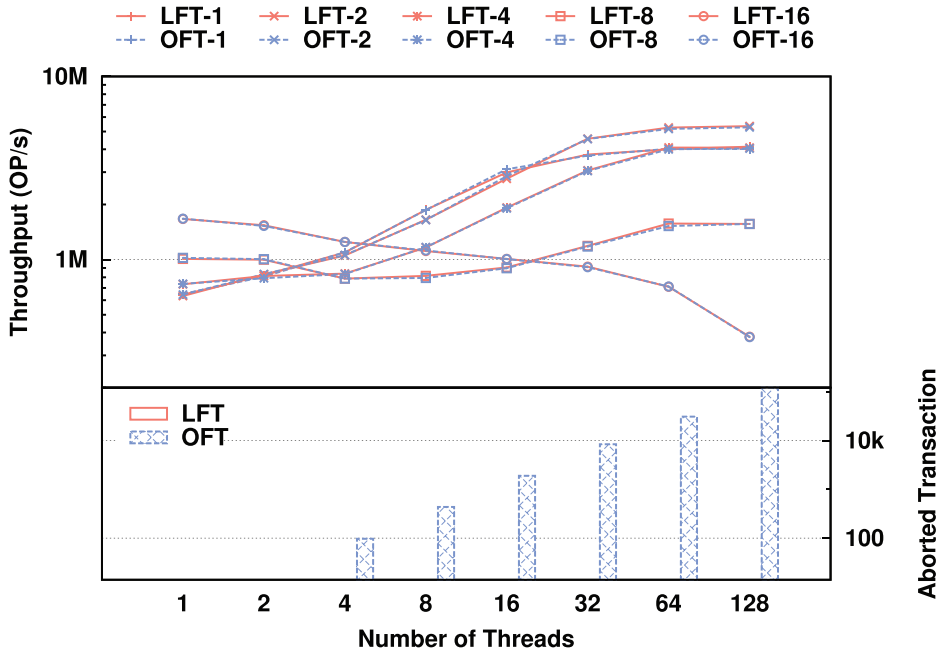


(a) Throughput and Spurious Aborts for LFT vs OFT Skiplists (1M Key Range): 50% Insert, 50% Delete, 0% Find



(b) Throughput and Spurious Aborts for LFT vs OFT Skiplists (1M Key Range): 33% Insert, 33% Delete, 34% Find

Fig. 7. Continued.



(c) Throughput and Spurious Aborts for LFT vs OFT Skiplists (1M Key Range): 15% Insert, 5% Delete, 80% Find

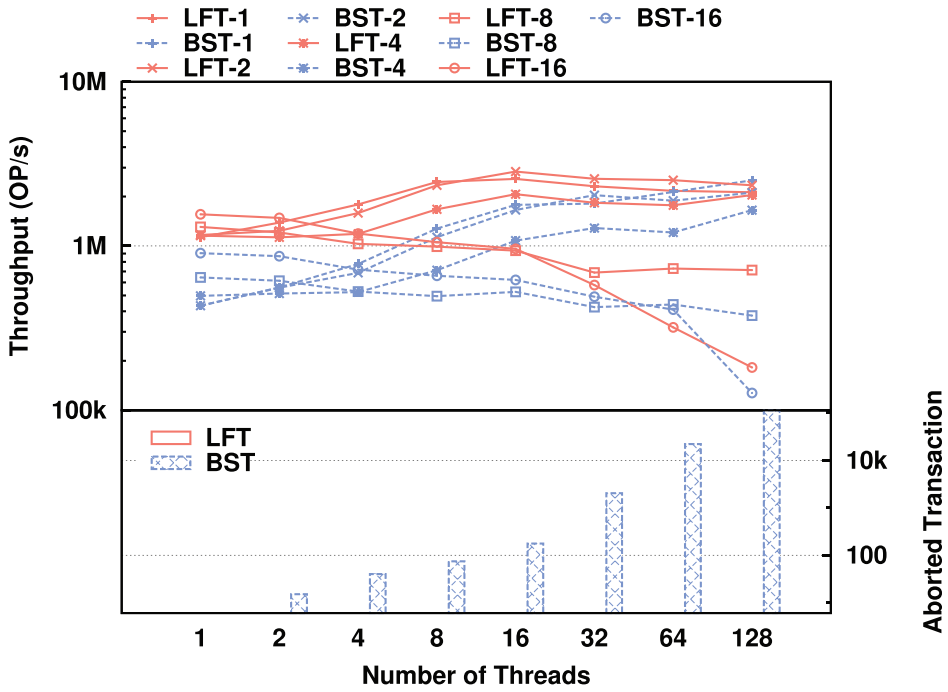
Fig. 7. Throughput and spurious abort counts for lock-free versus obstruction-free transactional skiplists (1M key range).

inspirations from previous semantics-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 [24], 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. A 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 [40], which is lock-free but only supports a static set of data items. Herlihy et al. later presented DSTM [23] that supports dynamic datasets 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, 32, 38]. We omit complete reviews because it is beyond the scope of this article. As more design choices were explored [31], we have seen emerging discussions on the issues of STM regarding usability [37], performance [3], and expressiveness [15]. There is also an increasing realization that the read/write conflicts inherently provide insufficient support for concurrency when shared



(c) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 15% Insert, 5% Delete, 5% Update, 75% Find

Fig. 8. Throughput for the lock-free transactional hash map (1M key range).

objects are subject to contention [27]. It has been suggested that "STM may not deliver the promised efficiency and simplicity for *all* scenarios, and a 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 [33]. 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 tradeoff 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 nonblocking progress guarantee one might expect from using a nonblocking library.

6.3 Semantic Conflict Detection

Considering the imprecise nature of data-based conflict detection, semantics-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 access to the underlying data structure. This results in a two-layer concurrency control. Transactional boosting proposed by Herlihy [22] 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 noncommutative 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 semantics-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 nonblocking 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. [27] later generalized this work and introduced 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 a *predicate* and relies on STM to synchronize transactional accesses [2]. Hassan et al. [20] proposed an optimistic version of boosting that employs a white box design and provides throughput and usability benefits over the original boosting approach. Other STM variants, such open nested transactions [35], 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 specifications obtained from programmers' input to inferring locks for abstract data types.

Spiegelman et al. [43] propose an approach that collects a read-set and write-set for a transaction and performs validation on the nodes before executing physical modifications. When a transaction begins, it is assigned a version number. The transaction then collects a local read-set of nodes that it reads and a write-set of nodes that it will modify. At commit-time, the transaction locks the nodes in the write-set and then it validates that all nodes in the read-set are unchanged by checking their version numbers. If the transaction fails to acquire a lock or the validation fails, that means another transaction made noncommutative method calls, and the transaction aborts. If successful, the transaction makes physical modifications to the nodes in the write-set, updates the nodes' version numbers, and then releases the locks. This approach benefits from the absence of rollbacks, but, when applied to nonblocking data structures, it diminishes the progress guarantee of the data structure by using locks. This problem could be remedied by replacing locks with transaction descriptors, and we could ensure opacity by applying a helping mechanism similar to the one used in our approach. However, to prevent the livelock scenario mentioned in Section 2.4, we must detect cyclic dependencies among transactions and abort one of the transactions when a cyclic dependency occurs. The aborted transaction would then need to perform a rollback of the changes it made, which would incur a performance penalty. We do not compare our approach to theirs as their source code is proprietary [42].

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 guarantee 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 70% speedup over our good-faith transaction boosting implementations across all scenarios. The performance gains over STMs are even more substantial: more than 13 times over the alternative word-based STM and 2 times over the object-based STM. In addition to 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 also presented an obstruction-free version of our algorithm which can be applied to dynamic execution scenarios and an example of our approach using map data structures.

Future work on this approach could include a semi-automatic transformation of a modified data structure containing methods such as Do_LocatePred and Do_Insert into a transactional data structure using the source-to-source transformation capabilities of the ROSE compiler [29]. ROSE would be necessary to generate methods (e.g., Insert from Do_Insert by inserting function calls to UpdateInfo, Do_Insert, and Do_LocatePred). ROSE would also be needed to add an ExecuteOps function to the transformed data structure.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their detailed and helpful suggestions.

REFERENCES

[1] Hagit Attiya. 2010. 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*. ACM, 1–5.

[2] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 6–15.

[3] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Queue* 6, 5 (2008), 40.

[4] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. 2008. Inferring locks for atomic sections. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 304–315.

[5] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *ACM SIGPLAN Notices*, Vol. 45. ACM, 67–78.

[6] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, 157–168.

[7] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. Springer-Verlag, 194–208.

[8] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 131–140.

[9] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. 2007. Lock allocation. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 291–296.

[10] Keir Fraser. 2004. *Practical Lock-freedom*. Ph.D. Dissertation. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[11] Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 2013. Concurrent libraries with foresight. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 263–274.
[12] Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 2015. Automatic scalable atomicity via semantic locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 31–41.
[13] Vincent Gramoli, Rachid Guerraoui, and Mihai Letia. 2013. Composing relaxed transactions. In *Proceedings of 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1171–1182.
[14] Khilan Gudka, Tim Harris, and Susan Eisenbach. 2012. Lock inference in the presence of large libraries. In *ECOOP 2012–Object-Oriented Programming*. Springer, 308–332.
[15] Rachid Guerraoui and Michal Kapalka. 2008. On obstruction-free transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 304–313.
[16] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 175–184.
[17] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–263.
[18] Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*. Springer-Verlag, 300–314.
[19] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Integrating transactionally boosted data structures with stm frameworks: A case study on set. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*.
[20] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. On developing optimistic transactional lazy set. In *Principles of Distributed Systems*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer International Publishing, 437–452.
[21] Maurice Herlihy and Eric Koskinen. 2007. *Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects*. Technical Report CS-07-08. Brown University.
[22] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 207–216.
[23] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. ACM, 92–101.
[24] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, New York, 289–300. http://dx.doi.org/10.1145/165123.165164
[25] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier.
[26] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
[27] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. 2010. Coarse-grained transactions. *ACM SIGPLAN Notices* 45, 1 (2010), 19–30.
[28] Pierre LaBorde, Steven Feldman, and Damian Dechev. 2015. A wait-free hash map. *International Journal of Parallel Programming* (2015), 1–28. http://dx.doi.org/10.1007/s10766-015-0376-3
[29] Lawrence Livermore National Laboratory. 2018. ROSE Compiler Project. (2018). Retrieved March 19, 2018, from http://rosecompiler.org/.
[30] Jonatan Lindén and Bengt Jonsson. 2013. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*. Springer, 206–220.
[31] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. 2004. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*. ACM, 1–7.
[32] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*.
[33] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: Synchronization inference for atomic sections. *ACM SIGPLAN Notices* 41, 1 (2006), 346–358.
[34] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 73–82.
[35] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 68–78.

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

[37] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, 47–56. http://dx.doi.org/10.1145/1693453.1693462

[38] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 187–197.

[39] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. *ACM SIGPLAN Notices* 46, 10 (2011), 51–64.

[40] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[41] Nir Shavit and Asaph Zemach. 1999. Scalable concurrent priority queue algorithms. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 113–122.

[42] Alexander Spiegelman. 2016. *Personal Communication*. (October 2016).

[43] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 682–696.

[44] D. Zhang and D. Dechev. 2015. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 99 (2015), 1–1. http://dx.doi.org/10.1109/TPDS.2015.2419651

[45] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. 2015. Low-overhead software transactional memory with progress guarantees and strong semantics. *SIGPLAN Notices* 50, 8 (Jan. 2015), 97–108. http://dx.doi.org/10.1145/2858788.2688510

Received November 2016; revised December 2017; accepted March 2018