LOFT: Lock-Free Transactional Data Structures

Avner Elizarov

Technion - Computer Science Department - M.Sc. Thesis MSC-2019-01 - 2019

LOFT: Lock-Free Transactional Data Structures

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Avner Elizarov

Submitted to the Senate of
the Technion — Israel Institute of Technology
Kislev 5779 Haifa November 2018

The research thesis was done under the supervision of Prof. Erez Petrank in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Concurrent data structures are widely used in modern multi-core architectures, providing atomicity (linearizability) for each concurrent operation. However, it is often desirable to execute several operations on multiple data structures atomically. This requirement cannot be easily supported by concurrent data structures, and requires the use of a locking mechanism or some sort of transactional framework. We present a design of such a transactional framework supporting linearizable transactions of multiple operations on multiple data structures in a lock-free manner. Our design is comprised of concurrent lock-free data structures supporting a transactional API, and a transaction engine responsible for executing operations as an atomic transaction. We employ a helping mechanism to obtain lock-freedom, and an advanced lock-free contention management mechanism to mitigate the effects of aborting transactions. When cyclic helping conflicts are detected, the contention manager reorders the conflicting transactions execution allowing all transactions to complete with minimal delay. Our design supports operation dependencies, i.e., results of operations can affect the inputs of subsequent operations or their control flow. To exemplify this framework we implement a transactional set using a skip-list, a transactional queue, and a transactional register. We present an evaluation of the system showing that we outperform general software transactional memory, and are competitive with lock-based transactional data structures, while providing an additional (lock-free) progress guarantee.

Abbreviations and Notations

$LOFT$	—	Lock-Free Transactional Data Structures
$TDSL$	—	Transactional Data Structures Libraries
$TL2$	—	Transactional Locking 2
STM	—	Software Transactional Memory
HTM	—	Hardware Transactional Memory
API	—	Application Programming Interface
TX	—	Transaction
CAS	—	Compare and Swap
JIT	—	Just in Time Compiler

Chapter 1

Introduction

Concurrent computing architectures have become widespread, raising the need for efficient and scalable concurrent algorithms and data structures. Concurrent data structures are designed to utilize all available cores, achieving good performance as well as consistent behavior in the form of linearizability of operations [15]. Many implementations of concurrent data structures were proposed in recent years (e.g., [21, 18, 9, 2]), providing an abstraction of a sequential data structure that can be accessed concurrently. However, it is often desirable to have several operations, operating on multiple data structures appear to take effect simultaneously and atomically. Linearizability of a single operation does not always suffice [1, 7, 13]. For example, moving an item from one queue to another while maintaining the invariant that other threads always see it in exactly one queue cannot be supported by a regular concurrent queue without costly synchronization.

To ensure the atomicity of such operations one can use a global lock to synchronize all accesses to the data structures, but this approach limits concurrency significantly, hindering scalability. Furthermore, the use of locks is susceptible to deadlocks, live locks, priority inversions, etc. A different approach for obtaining atomicity is *Transactional Memory* [14]. Transactional memories allow the programmer to specify a sequence of instructions that take effect atomically or not at all. Transactional Memories can be implemented using specialized hardware (HTM) or using software (STM)[20, 13]. In these implementations all reads and writes of a transaction logically appear to occur at a single instant of time (or not at all), and intermediate states are not visible to other threads. This approach is programmer-friendly, simplifying programming in a concurrent envirnoment. However, HTMs are limited in transaction size and STMs carry a performance cost [3]. In both implementations, conflicting accesses to data cause transactions to abort, and re-execute. The conflict detection mechanism and the need to re-execute transactions create an overhead that reduces performance and foils progress guarantees.

Recently, transactional data structure libraries [1, 11, 16] were proposed to deal with some of the above disadvantages. Transactional data structures limit transactions to only execute operations

on data structures, but they provide transaction semantics for concurrent data structures, and support atomic transactions containing sequences of data structure operations. Transactional data structures use mechanisms that build on the specific implementations of these data structures to reduce both the overhead and the abort rate.

In this work, we propose a framework for linearizable execution of transactions on data structures called LOFT that supports full lock-freedom. In order to be used with LOFT, a lock-free data structure has to be extended to support an adequate API that we define in this work. The proposed LOFT mechanism executes LOFT transactions, which consist of LOFT data structures operations. Transactions are always executed atomically. The LOFT engine extends and adapts a standard helping mechanism to help concurrently executing transactions to complete. In addition, the LOFT engine employs advanced contention management for handling cyclic conflicts. Cyclic conflicts are detected dynamically and transactions are executed in an adequate order that avoids the conflict. Next, we present an optimization for a common special case, where transactions contain a predetermined control flow, and they contain operations whose operands are known upfront. We formalize this special case in Section 2.5 and use reordering to fully avoid transaction aborts. Measurements show that this optimization (when applicable) benefits performance significantly, due to avoided aborts.

We exemplify LOFT for transactions on sets, queues, and register objects. We implemented a transactional abstract set, by extending a lock-free linked-list with the required LOFT API. We then add a skip-list to allow fast indexing into the list elements. The obtained transactional set is efficient and allows a transaction with various operations on multiple sets to be executed atomically. Next, we implemented a transactional queue, and finally we added a transactional register. We implemented and measured the LOFT sets against software transactional memory and the transactional data structures of [7]. Results show that the LOFT mechanism performs better than STMs and transactional data structures (for most scenarios). In contrast to lock-based transactional data structures and lock-based STMs, LOFT provides a lock-free progress guarantee and an advanced contention management mechanism for cyclic helping conflicts.

Chapter 2

The LOFT Framework

Our model for concurrent multi-threaded computation follows the linearizability model of [15]. In particular, we assume an asynchronous shared memory system where a finite set of deterministic threads communicate by executing atomic operations on shared variables.

We consider a collection of data structures $\langle d_1, d_2, \ldots \rangle$ on which simultaneous operations will be executed as a transaction. The collection is dynamic, i.e., data structures can be added or deleted from the collection during the execution. All operations in a transaction t are to be executed at a single instant in time, with no intermediate state of t visible to other threads. For each operation a result is returned according to the data structure's semantics. For example, consider two sets set_1, set_2 which are meant to be identical, and contain exactly the same items. When performing an operation on set_1, we would like it to be performed on set_2 as well s.t. no other thread can see the results of the operation performed only on one of the sets. Such behavior can be achieved in a lock-free manner using the LOFT framework.

In this Section we describe the LOFT engine and the underlying structures used by its data structures (Section 2.1). We then present the API to be implemented by LOFT data structures (Section 2.2). We continue with the engine implementation in Section 2.3. Finally, we expand on our support of operation dependencies (Section 2.4), and how we can improve our algorithm when operation dependencies do not exist (Section 2.5).

2.1 LOFT Engine

We propose the *LOFT Engine* which receives a list of data structure operations, executes them atomically and returns a list indicating the result of each operation according to the data structure semantics. Different transactions can overlap and perform changes to the data structures simultaneously. Thus, when executing a transaction we must provide a guarantee of atomicity. Simply

executing and committing the operations in a transaction one after the other is not possible since a concurrent transaction may affect later operations resulting in an inconsistent state of the data structures. Hence, a more complex mechanism should be implemented.

The basic idea of the proposed algorithm is to add a transaction label (descriptor), for each operation in a transaction t, that is visible to conflicting operations. When another transaction tries to label a conflicting operation, it detects t's label, and helps finish t's operations before proceeding with its own. This helps to not let transactions manipulate the memory addresses relevant to conflicting operations concurrently, and therefore avoids the need for validation prior to committing. Thus, we avoid unwanted aborts or an inconsistent state of the data structures. We rely on the conflict definition specified in [11] (two operations conflict if they do not commute). Namely, a label of an operation must be visible to operations that do not commute so that they do not run concurrently. Of course, labels may be further visible also to operations that commute with it and then excessive conflict resolution may harm performance, but not foil correctness.

Intuitively, the presented design resembles a lock-free analogue of two-phase locking. The first stage labels all of t's operations, and the second stage completes the operations and removes the labels. Contention is solved using a helping mechanism, and dead-locks (cycles) are solved using contention management. Let us start by defining several structures used by the engine and the data structures:

TNode: A TNode (Transaction Node) structure holds a pointer to its transaction t, the information needed to complete or un-label a single Transactional operation $op \in t$, op's result and op's index in the transaction. A LOFT data structure is responsible for creating a TNode for every operation performed on it.

DSop: A DSop (Data Structure Operation) structure encapsulates the parameters of a single operation op in a transaction t. This structure contains the data structure ds to perform the operation on, the operation $opName$ to perform on ds (e.g. *Add, Remove* for a set) and the parameters for that operation (e.g. *key* for a *set* operation). Additionally, it specifies op's dependencies on other operations as explained in Section 2.4. This structure is given as input to the transaction engine, as opposed to the TNode which is generated by the corresponding data structure.

Transaction Object: This is a descriptor object [10] that specifies a transaction t. It contains the *operations* to execute (given by the user as DSops), a *nodes* array containing pointers to t's TNodes (all initialized to null), a *status* field, a synchronized *index* and an *id*. The *status* field represents t's current status. The status is `Pending` while t's operations are being labeled. Upon successful labeling of t's operations its status is modified to `Done`, and it is exactly at this point that t's operations are considered logically committed and become visible to other threads. The status can also be set to `Replaced` when a cyclic helping conflict is detected as explained in Section 3. The *index* field represents the index of the next operation to label (initialized to 0), and the *id* field is a unique identifier representing t's age (smaller is older).

2.2 LOFT Data Structure API

Let us describe the methods that need to be implemented by LOFT data structures. We later present a specific implementation of a LOFT Set in Section 5, a LOFT Register in Section 6, and a LOFT Queue in Section 7. The API consists of three methods:

labelOp: This method receives a transaction object t with status `Pending`, the index i of an operation *op* in t's list of operations, and a thread-local *help stack* containing pointers to transactions this thread is currently helping. The *labelOp* method is responsible for ensuring the data structure state relevant to *op* does not change until *op* is completed. It does that by adding a label to the data structure that is visible to all other threads executing operations conflicting with *op* (e.g. changing the value of a linked list node's field). This label contains a pointer to *op*'s TNode and it allows transactions executing conflicting operations to help complete *t* before proceeding.

The *labelOp* method is also responsible for returning a TNode, as specified above. A *null* value is returned in case the labeling was unsuccessful and needs to be retried (for example, after helping a conflicting transaction). If during the execution of this method a label of a conflicting operation op' belonging to a different transaction t' is visible then a call to the *help* method to complete t' must be placed. If op' belongs to the same transaction t, then the label should be updated to indicate that it belongs to several operations.

completeOp: This method receives a transaction object t with status `Done`, the index i of an operation $op \in t$, and *op*'s TNode n (the output of *labelOp*). When a transactions t is marked as `Done` it takes effect in the sense that other operations view all of t's operations as already logically executed. The **completeOp** method performs the "physical" execution, completing operation execution on the relevant data structure, using the information in n. It also updates *op*'s result in t's transaction object, and removes the label added for *op* so conflicting operations can proceed. We require that *completeOp* be idempotent, i.e., that concurrent or consecutive invocations of *completeOp* with the same parameters have the same effect on the data structure as a single invocation. This enables different threads to help complete the transaction concurrently.

unlabelOp: This method receives a transaction object t with status `Replaced`, index i of an operation $op \in t$, and a TNode n belonging to *op* (the output of *labelOp*). A transaction is marked as `Replaced` when it is rescheduled to a later execution due to conflicts, and its operation labels should be removed. The *unlabelOp* method is responsible to remove the label added for *op* and revert any changes *op* performed on its data structure using the information in n. We require *unlabelOp* to be idempotent as well.

```
1:  execute(operations):
2:      t ← new Transaction(operations)
3:      t.status ← Pending
4:      executeTX(t, new Stack())
5:      return t.getResults()

6:  executeTX(t, helpStack)
7:      helpStack.push(t)
8:      while (i ← t.index) < t.size do
9:          if t.status = Pending
10:             op ← t.getOp(i)
11:             n ← op.ds.labelOp(t, i, helpStack)
12:             if n ≠ null
13:                 t.nodes[i].cas(null, n)
14:                 t.index.cas(i, i + 1)
15:         else
16:             break                ▷ t is Done or Replaced
17:     if t.status == Replaced
18:         replace(t)
19:     else
20:         t.status ← Done
21:         completeTransaction(t)
22:     helpStack.pop()

23: completeTransaction(t) :
24:     for i ← 0; i < t.size; i++ do
25:         n ← t.getNode(i)
26:         op ← t.getOp(i)
27:         op.ds.completeOp(t, i, n)

28: help(t, n, helpStack) :
29:     if n ∉ t.nodes              ▷ n is a duplicate TNode
30:         op ← t.getOp(n.opIndex)
31:         op.ds.unlabelOp(t, n.opIndex, n)
32:         return
33:     if t ∈ helpStack
34:         resolveCycle(t, helpStack)
35:         return
36:     executeTX(t, helpStack)
```

Figure 2.1: Transaction Engine Implementation

2.3 Engine Implementation

The LOFT engine implements the **execute** method which executes operations as an atomic transaction. It is external to the data structure implementation, and assumes the above API implemented by the data structures on which the operations should be performed. The **execute** method receives a list of data structure operations parameters $\langle DSop_1, \ldots, DSop_m \rangle$ to execute atomically, and it returns a list containing the result of each operation. The operations are encapsulated in a transaction object *t* that will be used for labeling. *t*'s *status* is initialized to `Pending`, indicating it is still in progress. Next, *executeTX* is called with t and an empty thread-local *helpStack*, to execute *t* from its first operation. The *helpStack* records all the transactions this thread is currently helping, and is used to identify cyclic helping conflicts as explained in Section 3. When executeTX is complete, *t*'s operations have completed and we return the results to the user.

The **executeTX** method is used to execute a transaction t, one operation after another starting from t's $index$ field. This method receives a transaction t to execute and a thread-local $helpStack$. The method adds t to its help stack and iterates over t's remaining operations (assuming it is still `Pending`). For each operation *op*, *executeTX* calls its data structure's *labelOp* method. *labelOp* is responsible for making *op* visible to other threads with conflicting operations and returns a TNode

n. We save n in t's nodes array (using a cas) and try to increase t's index. If null is returned (instead of a TNode), then the operation should be retried, and we do not increase t's index.

When t is no longer `Pending` or all of its operation were labeled we check its status. If it is `Replaced`, we call the *replace* method as described in Section 3. Otherwise, we set its status to `Done` and call *completeTransaction* which iterates over all of t's operations and calls *completeOp* on each operation's data structure to complete the operation and remove its label. Finally, we pop t from the help stack as it was completed (Line 22).

The **help** method is called by a Thread T executing transaction t' when it detects a label of a conflicting transaction t and tries to complete it. *help*'s parameters are t's transaction object, the TNode n belonging to the conflicting operation and T's $helpStack$ containing pointers to the transaction it is currently helping. This method verifies that n is indeed saved in t's *nodes* array. If not, it was mistakenly labeled due to a benign race and should be unlabeled, so we call *unlabelOp* of the relevant data structure and return. This scenario is described in Section 5.1. We also verify that a helping cycle is not created. If one is created, we handle it as explained in 3. If all checks passed we call $executeTX$.

2.4 Programming Interface and Operation Dependencies

Our current work does not focus on an adequate programming interface for specifying operations in transactions. We focus on the engine that executes transactions in a lock-free manner and the contention management mechanism. In our implementation we use a very simple solution in which the input list of operations is written in the format of a list of DSop structures. We leave more sophisticated software engineering constructions to future work.

Specifying operations with dependencies is even more challenging. For simplicity of presentation, the pseudo code described in this work does not handle data dependencies between different operations inside a transaction. However, our implementation does include support for such dependencies, including conflicting operations inside the same transaction, by allowing operation parameters to be determined by the results of previous operations. For example, moving *data* from one queue to another can be defined as adding a dependency between a *dequeue* operation on one queue to an *enqueue* operation executed on another queue. The *data* of the *enqueue* operation is unknown when the transaction starts executing. It is revealed during the transaction execution.

To obtain dependent data for an operation in our implementation, we let the user specify a function f which receives as input the *results* of all previous operations and outputs the *parameters* of the subsequent operation. The required update to the proposed engine design includes calling the function f before labeling an operation *op* to update its parameters.

The above can be extended to support control flow dependencies, which we have not implemented. For example, deciding the number of operations to perform in the transaction based on

the result of a previous operation cannot be specified using a list of DSop structures. However, it is possible to support control flow dependencies by changing the input to the transaction engine. We can replace the list of operations with a representation of the required logic as an object which supports: (1) Retrieving the next operation to label based on previous results, (2) updating the result of an operation after it was labeled (and thus advancing to the next operation). This object should support concurrent accesses to it, and its internal logic must be deterministic. One can think of this object as having an instruction pointer to the next operation, which is advanced when it receives the operation's result. When using such an object, control flow dependencies can be supported in its internal logic. The extension to support control flow dependence and an adequate programming interface is outside the scope of this work.

2.5 Abort Freedom

When the control flow of all transactions is known upfront (i.e., does not depend on results of transaction operations), and when there are no data dependencies in the sense that all parameters are also known in advance, it is possible to perform an optimization which sorts operations inside a transaction. This sorted execution reordering guarantees that no cyclic helping conflicts occur and full abort-freedom can be obtained. Measurements show that this optimization is highly beneficial (when applicable). Let us formalize the requirements of LOFT transactions and LOFT data structures that enable the sorting optimization and abort-freedom.

We call a set of transactions *data-independent* if the transactions control flow and operations parameters do not depend on result of transaction operations. A set T of *data-independent* transactions on a given set of LOFT data structures *enables the sorting optimization* if there exists a total order r between transactional operations and a labeling mechanism l which satisfy the following conditions:

1. Conflicting operations reside in the same equivalence class of r. Specifically, for operations op_1, op_2 if there exists a state of the data structures for which op_1 conflicts with op_2, then we call op_1 and op_2 *potentially conflicting*, and require $op_1 =_r op_2$. For example, two inserts of the same key to a set are (always) conflicting. On the other hand, a dequeue operation and an enqueue operation to the same queue are *potentially conflicting* since they conflict only when the queue is empty.

2. l's labeling of an operation $op_1 \in t_1$ blocks the labeling of any **potentially conflicting** operation $op_2 \in t_2$. For the LOFT engine, this means that t_2 is forced to *help* t_1 complete before executing.

3. Executing a transaction $t \in T$ (in its original operations order) is equivalent to executing t when its operations are re-ordered according to the relation r (while maintaining original order for

operations that are in the same equivalence class). Meaning, the result for each operation is the same, and the state of the data structure after executing the transaction is the same.

Let us exemplify the above on the set and the queue. Consider any given strict total order r_1 on the data structures, and a total order r_2 over operations on the same data structure, which places *potentially conflicting* operations in the same equivalence class. We define the total order r in our implementation as follows: $op \leq_r op' \iff$ either $op <_{r_1} op'$ or $op =_{r_1} op'$ and $op \leq_{r_2} op'$. We next describe r_2 for the LOFT set and LOFT queue.

For the LOFT set described in Section 5, we define r_2 as follows: $op \leq_{r_2} op' \iff op.key \leq op'.key$. When r is used with the labeling of operations described for the LOFT set, all above conditions are met:

1. Two set operations *potentially conflict* if they operate on the same key in the same set, and indeed such operations are placed in the same equivalence class of r.

2. The *labelOp* implementation presented in Section 5.1 does not allow transactions to concurrently label a node with the same key.

3. Sorting a transaction by the above order on sets does not change the result of each operation and the state of the set at the end of the transaction execution, because operations affect only a single key in the set. Operations performed on the same key remain sorted in the original order, hence their result remains unchanged.

For the LOFT queue we define all operations on the same queue to be in the same equivalence class of r_2. The labeling described in Section 7 does not meet the above requirement, as it allows for enqueues of transactions t and dequeues of transaction t' to execute concurrently, even though they *potentially conflict* (as described above). Therefore, the sorting optimization cannot be applied. However, it is not very difficult to design a labeling mechanism which labels both the *head* and the *tail* of the queue, forbidding concurrency of enqueues and dequeues. Such a labeling mechanism satisfies the above criteria and can enable the sorting optimization.

We next claim that transactions cannot create a helping cycle when their operations are sorted. Let r be the total order and l the labeling mechanism which satisfy the above requirements. We execute the operations in each transaction according to their order in r. Assume, in a way of contradiction, that there exists a helping cycle $t_1, t_2, \ldots, t_n, t_{n+1} = t_1$, s.t. each transaction t_i helps the next transaction in the cycle t_{i+1} due to a conflict when trying to label an operation $op_i \in t_i$. t_i helps t_{i+1} only if t_{i+1} earlier labeled an operation op'_i which conflicts with op_i. This means that op'_i was labeled prior to op_i. Since operations are labeled one after the other, then op'_i must have been labeled prior to op_{i+1} because op_{i+1} cannot be labeled due to a conflict with an operation $op'_{i+1} \in t_{i+2}$. The operations inside a transaction are sorted, which means that $op'_i \leq_r op_{i+1}$. If $op'_i =_r op_{i+1}$,

then they conflict, which suggests that op'_i also conflicts with op'_{i+1} (transitivity), however both op'_i and op'_{i+1} are labeled, meaning they did not conflict. Thus, $op'_i <_r op_{i+1}$. Since op_i and op'_i are conflicting they must be in the same equivalence class, hence $op_i =_r op'_i <_r op_{i+1}$ for every i. From transitivity, we get that $op_1.key < op_n.key$, but $op_n.key < op_{n+1}.key = op_1.key$. Thus, our assumption that a helping cycle exists is contradicted.

Chapter 3

Resolving Cyclic Helping conflicts

Existing transactional memory systems abort transactions whenever a conflict is detected, thus having to re-execute the transaction or require the programmer to specify how to proceed following an abort. Much work has been done to reduce the number of aborts [17, 1, 11, 22]. In all previous works, as well as in this work, a transaction must be aborted when a dependency cycle occurs. In this work we present a contention manager that dynamically identifies cyclic conflicts and resolves such cycles by rescheduling one transaction, for each cyclic dependency in a way that guarantees progress.

```
37: resolveCycle(t, helpStack) :
38:     cycle ← transactions above t in the stack including t
39:     if a transaction in the cycle is ¬ Pending
40:         return
41:     nextT ← oldest transaction in cycle
42:     oldT ← transaction above nextT in helpStack
43:     newT ← new Transaction(oldT.ops)
44:     oldT.nextT.cas(null, nextT)
45:     oldT.newT.cas(null, newT)
46:     oldT.status ← Replaced
47:     replace(oldT)
48: replace(t) :
49:     for i ← 0; i < t.size; i++ do
50:         n ← t.getNode(i)
51:         op ← t.getOp(i)
52:         op.ds.unlabelOp](t, i, n)
53:     executeTX(t.nextT, new Stack())
54:     executeTX(t.newT, new Stack())
```

Figure 3.1: Cycle Resolving Mechanism

We proceed with the details. In our framework concurrent transactions may create a helping cycle. For example, consider two transactions t_1, t_2, each executing the same two operations op_1, op_2 in an opposite order. This can lead to a scenario in which t_1 and t_2 add a label for op_1 and op_2 respectively. Then, when t_1 tries to execute op_2 it notices t_2's label and must help t_2 which requires executing op_1 and forms an endless helping loop. This happens since accesses to the Data Structures and within a data structure are not sorted in advance. Sorting accesses is a standard mechanism for avoiding deadlock or other cyclic conflicts. As explained in Section 2.5 we cannot always sort

operations according to accesses because operations in transactions may depend on one another and when they do, operands are not available upfront for sorting. Therefore, a mechanism for resolving helping cycles is employed.

Our mechanism finds such cycles using a help stack containing pointers to the transaction objects of the transactions a thread is currently helping. When a call to the *help* method with a transaction object t is made, we validate that t does not exist in our help stack (line 33). If it does, then a helping cycle exists and we must reschedule the execution of one of the transactions in the cycle to let other transactions complete. The helping cycle consists of all transactions in the *helpStack* starting from t. We first verify that all of the transactions in the cycle are still `Pending`, as some other thread might have already resolved the cycle. If the cycle still exists, we find the oldest transaction in the cycle (smallest id) denoted $nextT$, and attempt to help it complete first. We reschedule the execution of the transaction $oldT$ that $nextT$ tried to help (the transaction above it in the stack), and un-label $oldT$'s operations. Thus, we remove the conflict which caused $nextT$ to help $oldT$, and now $nextT$ can continue to execute. Before un-labeling $oldT$'s operations, we replace its current transaction object with a new transaction object $newT$ with $oldT$'s operations. We save a pointer to $newT$ and $nextT$ in $oldT$. We then set $oldT$'s status to `Replaced`, thus informing all other threads helping $oldT$ to call the *replace* operation which performs the following protocol:

1. Remove the labels installed by $oldT$ using the data structure's *unlabelOp* method.
2. Call *executeTX* with $nextT$ to complete the transaction $nextT$ before re-labeling $oldT$'s operations, hence the conflict causing $nextT$ to help $oldT$ will not occur again.
3. Call *executeTX* with $newT$ to re-label $oldT$'s operations with a new transaction object.

By enforcing all threads helping $nextT$ to perform the above protocol we ensure that $nextT$ or some older transaction (in a future helping cycle with $nextT$) completes and transactions don't remain stuck. The reason for replacing $oldT$'s transaction object is to enable all helping threads to distinguish between operations labeled prior to its re-labeling and operations labeled in the rescheduled execution.

Chapter 4

Related Work

Much work has been done in the field of software transactions (e.g. [4, 1, 13, 6]). These came to replace (or integrate with) hardware transactional memory, and provide an interface for composing several operations atomically. A typical implementation of software transactional memory maintains a *read-set* (addresses accessed during the transaction and their values) and a *write-set* (addresses to be updated and their new values) for all operations performed during the transaction. A locking mechanism is employed to lock all addresses in these sets and the content of the *read-set* is validated to not have changed. If this validation succeeds, then the transaction commits and performs the writes in the *write-set*. Otherwise, the transaction is aborted and has to be retried. Under contention, this basic approach induces aborts and may damage performance, sometimes drastically. The focus of this work is to improve performance, eliminate aborts, and provide a progress guarantee to transactions. This is done at the cost of limiting transaction's capabilities to only execute data structure operations.

There exist previous attempts to implement transactional data structures libraries. Transactional Boosting [11] proposed a methodology for transforming highly concurrent linearizable objects into transactional objects. This transformation enables concurrent commutative method invocations on transactional objects to execute without aborting the transaction. Other solutions [1, 7] incorporate the construction of a concurrent data structure and an interface providing the ability to execute an atomic sequence of operations (*add*, *remove* or *contains*) on them. This requires some changes to the underlying data structure code. These implementations are not lock-free and transactions can be aborted.

A recent paper [22] also proposes lock-free support for transactions performed on linked data structures in an attempt to eliminate transaction aborts. However, their mechanism is not abort-free. Transactions abort when a cyclic helping conflict is detected. Also, in their implementation each transaction must complete all *insert* and *remove* operations of the set successfully according to the set semantics. Otherwise, the transaction aborts. This *always-succeeding* transaction framework

can be limiting when a programmer does not want to fail the entire transaction when one operation is unsuccessful. For example, having an add operation fail means that an item with the same key is already present, and in many cases this outcome is sufficient for the programmer. Our implementation can easily support this *always-succeeding* semantics by adding a $Failed$ option for a transaction's status, which will act similarly to $Replaced$, only without the need to re-label operations. In Addition, the mechanism of [22] is only implemented on lists, it does not support singleton operations and operation dependencies.

Finally, there is a subtle bug in the algorithm proposed in [22], which goes as follows. Consider the scenario in which a thread $T1$ helps perform an add operation belonging to a transaction executed by another thread $T2$. Thread $T1$ becomes idle right before it adds (CAS) the new node n between its $pred$ and $succ$, and $T2$ in that time performs the add operation itself (by adding his own new node n') and then completes the transaction. Immediately thereafter, another transaction kicks in and performs a $remove$ of n'. Finally, $T1$ awakens and performs the intended CAS successfully, as both $pred$ and $succ$ remain in the data structure and $pred$'s next pointer is again set to $succ$. It then completes the transaction and n is considered logically in the list. This means that the add operation was performed twice for the same transaction, thus violating the correctness of the data structure.

Chapter 5

A LOFT Set

In this section we present an implementation of a LOFT set. Each item *i* in a set has a unique key k_i and some *data*. There exists a total order $<$ on all possible keys such that for any two keys $k_i \neq k_i'$ either $k_i < k_i'$ or $k_i' < k_i$. We also assume that there exist two special keys $-\infty$ and ∞ which are respectively smaller (larger) than all possible keys. The LOFT set supports the following operations: (1) *add* for adding a new item, (2) *remove* for removing an item, (3) *contains* for checking existence of an item, and (4) *get* for retrieving an item's *data*.

We implement the LOFT set as a simple lock-free concurrent (ordered) linked-list, based on Harris's non-blocking linked list implementation [9]. We design this list carefully to maintain correctness and Linearizability, which ensures all operations in a transaction take effect at once in the presence of concurrently executing transactions and singleton operations (such as *add*, *remove*, etc). In addition, we add a skip-list index on top of each linked list for faster lookup time. The skip-list is implemented in a more relaxed manner that does not always hold an updated view of all items in the set. This inaccuracy may increase the search time but will never lead to an incorrect result of an operation. We start by describing the linked-list implementation and further elaborate on the indexing skip-list and why it is safe to relax its coordination with the linked-list in Section 5.3.

Similar to Harris [9], a node in the list contains an item's *data*, *key*, and a *next* pointer to the following node in the list. To support the *remove* operation, the *next* pointer also contains a *marked* boolean flag. The pointer and boolean flag can be updated together or apart from each other, atomically. A node is considered to be (logically) removed from the set if its *marked* flag is turned on. The list contains two special unlabeled dummy nodes - *head* and *tail*, with keys $-\infty$ and ∞ respectively. The list starts with the *head* node and ends with the *tail* node. These nodes are not considered data structure items, and therefore cannot be removed from the list. We next extend his list to support LOFT.

We view two set operations as conflicting when they operate on the same key. This expands the non-commutative definition of [11] which only considers two set operations as conflicting when at

least one is a write operation (e.g. *add*, *remove*). In our implementation two read operations on the same key (e.g. *contains*, *get*) are also treated as conflicting. [1] Thus, the labeling of an operation op in a transaction t is implemented by adding a special transaction descriptor to a node containing op's key. If no such node exists, e.g., during a successful *add* operation or an unsuccessful *remove* operation, then a new node with op's key is allocated and added in its correct location in the list. The descriptor contains a pointer to t's transaction object, op's result and op's index. For brevity, the pseudo code presented in this section does not handle multiple operations on the same key in the same transaction. However, our implementation does support this. These operations need to label the same node, so we add a flag to the transaction descriptor stating that several operations are performed on this node. When completing or un-labeling the operations we calculate the new state of the node based on all operations performed on it in their original order.

Recall that a TNode structure holds all the information required for completing or un-labeling an operation, op's result, and op's index. For the list, we are able to save space by keeping all this information on the actual list node. This is an **optimization** performed to avoid allocating a TNode for each operation. We store op's index and result in the list node's transaction descriptor. As for the information needed for completing or un-labeling op, we require only the list node's address and its $data$ field. Thus, a pointer to a list node serves as pointer to the TNode.

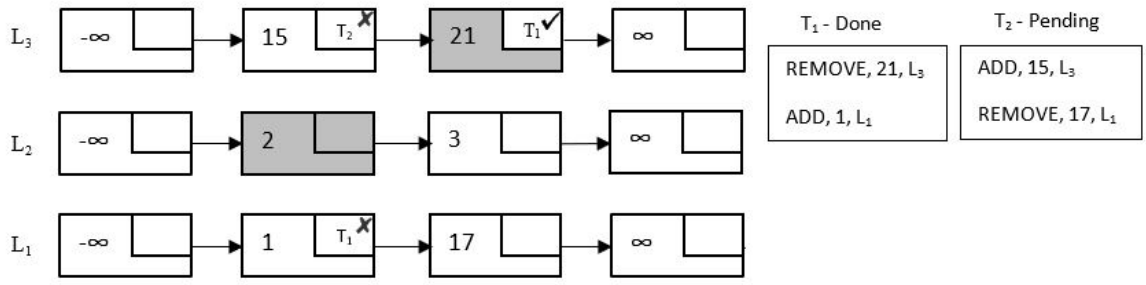

Figure 5.1: Collection of three LOFT sets, each containing the *head* and *tail* nodes, and other items, some currently labeled by a transaction. Grayed out nodes are items marked for deletion (logically removed). The transactions appear next to the lists, with their status and operations.

In Figure 5.1 we present an example of three LOFT sets. The arrows represent the next references. Each item has a transaction descriptor, possibly empty, which contains a pointer to a transaction and the *result* of the transaction's operation on the node (✓ for a successful operation and X for a failed one). Transaction T_1 is *Done*, and T_2 is still `Pending`. T_1 successfully removed the node with key 21 in L_3 and failed to add an item with key 1 to L_1, since an item with that key already exists. T_1's remove operation was both labeled and completed. However, T_1's add operation was

[1] These additional conflict resolutions reduce performance, but we did not see a simple correct, and efficient way to avoid them for the LOFT set design.

only labeled and not completed. This completion can be accomplished by any helping thread. T_2 labeled its first operation, an unsuccessful *add* of key 15 to L_3, but did not label its second operation yet.

In what follows we describe the LOFT set algorithm. We present the implementation of the required API in Section 5.1 and the auxiliary methods used to implement it (Section 5.2). We then introduce a skip-list index used for faster lookup time (Section 5.3), and present further optimizations in Section 5.4.

5.1 LOFT API Implementation

```
55: labelOp(t, i, helpStack) :
56:     key, opName, data ← t.getOp(i)
57:     shouldKeyExist ← (opName ≠ Add)
58:     pred, n ← find(key)
59:     if n.key = key          ▷ Item with key possibly exists
60:         return labelNode(t, n, i,
61:                          shouldKeyExist, helpStack)
62:     else                    ▷ No item with key exists
63:         res ← ¬shouldKeyExist      ▷ Update result
64:         desc ← new Descriptor(t, i, res)
65:         newNode ← new Node(key, data, desc)
66:         newNode.next ← n
67:         if pred.next.cas(n, newNode, false, false)
68:             return newNode          ▷ newNode added
69:         return null

70: unlabelOp(t, i, n) :
71:     if n = null
72:         return
73:     DSop ← t.getOp(i)
74:     res ← n.res
75:     desc ← n.transaction
76:     if desc = null ∨ desc.t ≠ t
77:         return                  ▷ Already Unlabeled
78:     existed ← (DSop.opName = Add ∧ ¬res)
79:               ∨(DSop.opName ≠ Add ∧ res)
80:     if existed                  ▷ n already existed
81:         n.transaction.cas(desc, null)      ▷ Un-label
82:     else                        ▷ n was added by t
83:         mark n.next
84:     return

85: completeOp(t, i, n) :
86:     opName ← t.getOp(i).name
87:     res ← n.res
88:     data ← n.data
89:     desc ← n.transaction
90:     if desc = null ∨ desc.t ≠ t
91:         return                  ▷ Already Completed
92:     mark ← (opName = Remove) ∨
93:            (opName ∈ {Contains, Get} ∧ ¬ res )
94:     if mark
95:         mark n.next                ▷ remove n
96:     else                        ▷ Clear label
97:         n.transaction.cas(desc, null)
98:     if opName = Get
99:         t.setResult(i, data)
100:    else
101:        t.setResult(i, res)
```

Figure 5.2: The LOFT set API implementation

LabelOp: The **labelOp** method receives a transaction t, index i of an operation op in t, and a $helpStack$. It then extracts op's *key*, *data* and the name of the operation to perform on the item (e.g. add, remove). This method tries to label an existing node containing key with a transaction descriptor pointing to t or adds such a node if a node with key does not exist. A variable $shouldKeyExist$ is set to true if op is not an *add* operation, meaning op expects that a node with $op.key$ exists in the list. A call to *find* is executed in order to get the predecessor $pred$ of the node containing *key* and $pred$'s successor n s.t. $n.key \geq key$. If $n.key = key$, then it is possible that a node with key is already present. We call the *labelNode* method to label n and return *labelNode*'s result (n if n was successfully labeled or null to retry).

If $n.key > key$ (Line 62), then no node containing *key* exists, so we add a new node $newNode$ between $pred$ and n. This node holds op's *data*, and is labeled with a new Transaction Descriptor pointing to t and result set to be the opposite of *shouldKeyExist* (since a node containing *key* does not exist). We set $newNode$'s next pointer to n, and try to insert it using a *cas*. If the *cas* is successful, we return $newNode$. Otherwise, the data structure state changed, and we return null to retry the operation.

Duplicate Labels It may happen that more than one node gets labeled for the same operation op in a transaction t. This is possible when different threads try to label op concurrently on different nodes, e.g. both allocate a new node for insertion. However, in our algorithm only one is actually used for labeling op. This is the first node added to t's nodes array using a *cas* instruction (Line 13). All other nodes are not added to the nodes array and do not become part of the transaction. Hence, we remove their label (Line 31), ensuring that op will only be executed once. An example of such a scenario is when two threads T_1 and T_2 execute the same *Add* operation op of a transaction t and try to perform the cas operation in Line 67. Suppose T_1 succeeds and T_2 is suspended by the scheduler before completing the cas operation. Meanwhile, the transaction t is completed and a subsequent transaction executes a remove operation on the same node. T_2 is now scheduled to continue and successfully completes the cas operation between $pred$ and n. Thus, T_2's inserted node is labeled by op of transaction t, that has already completed. This scenario is then identified when another thread notices this label and starts the helping procedure. It discovers that this node is not referenced by the nodes array and then it reverts the operation op on this node (Line 31).

CompleteOp: The **completeOp** method is responsible for completing a transaction t's operation op on a single node, and is called only after t's state is `Done`. **completeOp**'s parameters are a node n, a transaction t, and op's index i. We first verify that n is still labeled by t. If not, we return as op was already completed. To complete op, we either need to remove n from the set (marking its *next* pointer) or clear its label (its *transaction descriptor*). We remove n in the following two cases: (1) op is *remove*, so we either complete a successful *remove* operation or remove a dummy node in case of an unsuccessful *remove* operation; (2) res is false and $opName \in \{$*contains, get*$\}$, so we

remove a dummy node in an unsuccessful *contains* or *get* operation. We do not un-label a marked node as it will be physically removed by a later *find* operation, making it inaccessible to other transactions. In all other cases we un-label n by clearing its transaction descriptor, thus allowing it to be manipulated by other transactions.

UnlabelOp: The **unlabelOp** method is responsible for removing the label of a transaction's operation *op*, and is called only after t's state is `Replaced`. **unlabelOp**'s parameters are a node n, a transaction t, and *op*'s index i. As in *completeOp* we verify that n is still labeled with t. If not, we return as n was already unlabeled. To un-label *op* we either need to remove n's label if n was already in the list prior to t or otherwise remove the node n. n was already in the list prior to t in the following two cases: (1) *op*'s result was *false* and its operation was *Add* (an add operation fails only if a node with the same key already exists). (2) *op*'s result was *true* and its operation was not *Add* (i.e. *remove*, *cotains* or *get*). These operations succeed only if a node with the same key exists in the set. In all other cases, n was inserted by t and it should be removed by marking n's *next* pointer.

5.2 Auxiliary methods

```
102: find(key) :
103: retry:                                  ▷ Starting a new search
104:     pred ← head
105:     n ← pred.next
106:     while true do                       ▷ While item not yet found
107:         succ, mark ← n.next
108:         if mark
109:             pred.next.cas(n, succ, false, false)
110:             goto retry                  ▷ List changed so retry
111:         if n.key ≥ key
112:             return pred, n
113:         pred ← n
114:         n ← succ
```

```
115: labelNode(t, n, i, shouldKeyExist, helpStack):
116:     desc ← n.transaction
117:     if desc ≠ null                      ▷ Another transaction
118:         if desc.t = removeT
119:             mark n.next                 ▷ Complete remove
120:             return null                 ▷ n removed so retry
121:         if desc.t = t
122:             return n                    ▷ Op already executed
123:         help(desc.t, n, helpStack)
124:         return null                     ▷ Retry
125:     if ¬ shouldKeyExist ∧(¬n.next.isMarked())
126:                                         ▷ Item exists
127:         desc ← new Descriptor(t, i, false)
128:         if n.transaction.cas(null, desc)
129:             return n                    ▷ Successfully labeled n
130:     desc ← new Descriptor(t, i, true)
131:     if shouldKeyExist ∧ n.transaction.cas(null,desc)
132:         return n                        ▷ Successfully labeled n
133:     return null                         ▷ Retry
```

Figure 5.3: Auxiliary functions

Find: The **find** method receives a key *key* and returns two items, $pred$ and n, such that (1) $pred.key < key$, (2) $n.key \geq key$ and (3) n is $pred$'s successor in the data structure. The method starts its traversal from the *head* node. The $pred$ and n references are advanced throughout the

data structure, until the relevant two items are found and returned (Line 112). In addition to finding the requested item, the *find* method also helps physically remove items that are logically removed ($next$ pointer is marked). If n is unmarked, the traversal continues. Otherwise, we try to physically remove n by performing a cas of (Line 109) $pred$'s successor to $succ$ (n's successor). We then start the traversal again from *head* as the state of the data structure has changed.

LabelNode: The **labelNode** method receives a transaction t, a list node n, a *helpStack*, an operation's index i and a boolean $shouldKeyExist$. It returns a node labeled with transaction t. If n's state in the list is inconclusive it returns null to retry. We check that n is not labeled with another transaction by checking its transaction descriptor $desc$. If it is non-null and belongs to the singleton remove operation (explained in Section 5.5), then n is in the process of being removed. We mark n so it will be physically unlinked from the data structure and return null to retry. If n is already labeled with t, then another thread executed this operation on n, as part of t so we return n. If n is labeled by a different transaction $desc.t$, then we help $desc.t$ complete (Line 123), and return *null* to retry.

If n is unlabeled we try to label it. If n is not expected to exist in the set ($\neg shouldKeyExist$) and its $next$ pointer is not marked then the operation is unsuccessful. We allocate a transaction descriptor with a *false* result and try to label n (Line 128), Upon success we return n. If n is marked, then we return $null$ in Line 133, assuming it will be unlinked by a consequent *find* method. If n is expected to exist in the set, and we successfully label it with a descriptor $desc$ pointing to t (Line 131), then it is not marked (since every marked node remains labeled forever with the transaction that marked it), and we can return it. This means that the operation was successful so $desc$'s result is *true*. If the labeling fails, then n is already labeled, and we return $null$ to retry.

5.3 Adding Fast Skip-List Indexing

Each operation executed on the LOFT set generates a call to the *find* procedure. This procedure finds a predecessor of the node with the required key by traversing the list from its head, one node at a time. This traversal is extremely costly when the lists are long, and we would like to start the traversal closer to the required node. To do that, we add a skip-list index to the set similarly to [1, 12], mapping keys to list nodes. This mechanism will be used by the *find* method to retrieve some close preceding node to the required node much faster.

The indexing skip list does not necessarily represent the exact content of the set, as it is lazily updated by threads after completing a transaction. This does not hurt the algorithm as the *find* method only needs a valid predecessor of the required node. Upon completing a transaction, every successful *add* and *remove* operation in the transaction is performed on the index with the relevant node and key. Successful singleton *add* and *remove* operations also propagate their operation to the index.

5.4 Additional Optimization

When implementing the transactional set we performed an additional optimization for improved performance. Similar to Doug Lea's *Concurrent Skip List Map* implementation in Java we replaced the mark bits with Marker Nodes, for marking logically deleted nodes. When deleting a node, instead of marking its *next* pointer, we splice in a Marker Node. Eliminating marker bit masking or the use of java's *Atomic Markable Reference* class achieves faster traversal time.

5.5 Singleton Methods

```
134: add(key, data) :
135:     while true do
136:         pred, n ← find(key)
137:         if n.key = key
138:             desc ← n.transaction
139:             if desc ≠ null          ▷ Another transaction
140:                 if desc.t = removeT
141:                     mark n.next          ▷ Complete singleton remove
142:                 else
143:                     help(desc.t, n, new Stack())
144:             else if ¬n.next.isMarked()
145:                 return false      ▷ Item already exists
146:         else
147:             newNode ← new Node(key, data, null)
148:             newNode.next ← n
149:             if pred.next.cas(n, newNode, false, false)
150:                 return true ▷ Successfully added item

151: remove(key) :
152:     while true do
153:         pred, n ← find(key, this)
154:         if n.key = key
155:             res ← labelNode(removeT, key, n, true)
156:             if res = n        ▷ Labeled n with removeT
157:                 mark n;           ▷ Logically remove n
158:                 return true
159:         else                          ▷ key doesn't exist
160:             return false              ▷ Failed to remove

161: contains(key) :
162:     return get(key) ≠ null

163: get(key) :
164:     while true do
165:         pred, n ← find(key)
166:         if n.key ≠ key          ▷ key doesn't exist
167:             return null
168:         data ← n.data
169:         desc ← n.transaction
170:         if desc = null                ▷ n is unlabeled
171:             return data
172:         t ← desc.t
173:         if t = removeT
174:             return null
175:         if t.status = Pending
176:             op ← t.getOp(n.opIndex)
177:             exist ← (op = Add ∧ ¬n.res)
178:                 ∨(op ≠ Add ∧ n.res)
179:             return exist ? data : null
180:         else
181:             help(n, t, new Stack())
182:             deleted ← n.next.isMarked()
183:             return deleted ? null : data
```

Figure 5.4: Singleton methods

In addition to supporting transactions, our implementation enables the use of singleton methods (e.g. **add, remove**) on the data structure. These methods perform a single non-transactional oper-

ation and are transaction-aware, meaning they do not interrupt ongoing transactions. The singleton methods are lock-free, however the **get** and **contains** methods can be made wait-free if they do not help with the physical deletion of nodes (in the *find method*). These singleton methods do not incur the overhead of creating a transaction object, however when needed they do help transactions. To support the remove method we use a simplified shared transaction object *removeT* to notify transactions that this node is currently being removed by a singleton remove operation. Once a node is labeled by *removeT* it is logically deleted from the data structure, and will be marked by other operations and later physically removed.

Chapter 6

A LOFT Register

A transactional Register supports transactional *read* and *write* operations on a single memory location. Operations on the register are restricted to be inside a transaction. A Transactional Register has a pointer r to a Register. The Register itself also serves as a TNode structure, and it consists of the following fields: (1) a *transaction* pointer (the label), (2) the previous data stored in it $prev$, which is used to cancel an operation, (3) the current data stored in it $data$, (4) the index of the transactional operation performed on it i and (5) a boolean $isFirst$ stating if the register was created by the first operation in the transaction (information needed for un-labeling). We define any two operations on the register as conflicting, hence expanding the non-commutative definition of [11] which only requires that *read* and *write* operations conflict. [1] We implement the labeling of an operation in transaction t by pointing r to a new register with a transaction field set to t. We describe the register's implementation of the transactional API:

labelOp(t, i, helpStack): This method labels an operation op with index i in transaction t. We first check the transaction label of the register reg. If it belongs to a different transaction then $help$ is called for that transaction and we return null to retry. If it belongs to t then we verify that op wasn't already labeled by checking reg's index. If it is larger than i, then a later operation was already labeled, which means that op was labeled as well and we return null to continue. If $reg.index = i$ then op is currently labeled and we return reg as its TNode. If all previous validations passed, we create a new Register labeled with t. We set its $data$ field to the operation's new $data$ for a *write* operation or the data in the current register for a *read* operation. $prev$ is initialized to the data in the current register and i is set to the operation's index. Finally, we try to cas r to point to the new register we created and if successful return it.

unlabelOp(t, i, n): We validate that r points to a Register labeled with t. If so, and n belongs to the first operation in t performed on the register (which holds the data prior to t's labeling), then

[1] These additional conflict resolutions reduce performance, but we did not see a simple correct, and efficient way to avoid them for the LOFT register design.

```
184: labelOp(t, i, helpStack) :
185:     reg ← r
186:     opName, data ← t.getOp(i)
187:     if reg.t ≠ null ∧ reg.t ≠ t
188:         help(t, reg, , helpStack)
189:         return null
190:     if reg.t = t ∧ reg.index > i
191:         return null    ▷ later operation already labeled
192:     if reg.t = t ∧ reg.index = i
193:         return reg        ▷ operation is already labeled
194:     firstOp ← reg.t ≠ t     ▷ 1st operation in t on r
195:     prev ← reg.data
196:     if opName = Read
197:         data ← reg.data
198:     else                              ▷ Write operation
199:         data ← op.data
200:     newReg                                        ←
    new Register(data, prev, index, t, firstsOp)
201:     if r.cas(reg, newReg)
202:         return newReg
203:     return null

204: unlabelOp(t, i, n) :
205:     if n = null
206:         return
207:     reg ← r
208:     if reg.t ≠ t
209:         return
210:     if n.firstOp
211:         prev ← n.prev
212:         data ← n.prev
213:         unlabeledReg                              ←
    new Register(data, prev, −1, null, false)
214:         r.cas(reg, unlabeledReg)

215: completeOp(t, i, n) :
216:     reg ← r
217:     if reg.t ≠ t
218:         return
219:     t.setResult(i, n.data)
220:     if n == reg
221:         reg.t ← null                  ▷ remove label
```

Figure 6.1: LOFT register API implementation

we create a new unlabeled Register. We initialize its $data$ and $prev$ fields to $n.prev$, and try to cas r to this new Register.

completeOp(t, i, n): We validate that r points to n. If so, we create a new unlabeled Register and initialize its $data$ field to $n.data$ and $prev$ field to null. We then cas r to this new Register.

Chapter 7

A LOFT Queue

Our Transactional Queue is based on Michael-Scott's concurrent lock-free queue [19] implemented as a linked list and two pointers: *head* and *tail*. *tail* is a pointer to the last node in the queue and *head* is a pointer to a Head structure. This structure is composed of (1) a pointer to a node preceding the first node in the queue (initialized with a dummy node), and (2) a pointer to a transaction object of a dequeuing transaction (or null, if no such transaction exists). Each node in the list is implemented similarly to nodes in the LOFT Set implementation (excluding the mark bit on the next pointer). A node contains a transaction descriptor (label) and it also serves as a TNode structure. The queue supports the *enqueue* and *dequeue* operations. We define two operations as conflicting if they are both dequeues or both enqueues and define an enqueue operation and a dequeue operation to be conflicting when the queue is empty. This fits the non-commutative definition of [11] which determines conflicting operations to be operations that do not commute.

Enqueue operations in a transaction t add labeled nodes to the end of the list (successors of the node *tail* points to). This allows concurrent transactions to help t before performing their own *enqueue* operations. The *tail* pointer is updated only after t's status is set to `Done`. *Dequeue* operations in a transaction t verify that *head* is labeled with t (its transaction pointer points to t) and if so, they start traversing the list looking to label the first unlabeled node (was not dequeued by a previous operation). Concurrent transactions notice $head$'s label and help t finish before performing their own dequeues. An important invariant of the queue is that the node pointed to by $tail$ is a successor of the node pointed to by $head$. We now describe the queue's implementation of the LOFT API:

labelOp(t,i,helpStack): For an **enqueue** operation op, the *labelEnq* procedure tries to add the new item as the last node in the linked list. This is done by verifying that *tail.next* is null or is labeled by t. If so, *labelOp* traverses through the next pointers to the end of the list, adds a new node n containing the new item and returns n. n is labeled with t, and its result is set to $true$ since the enqueue was successful. It is possible that op was already labeled, which is detected by checking

```
222: labelOp(t, i, helpStack) :
223:     op ← t.getOp(i)
224:     if op.name = Enqueue
225:         return labelEnq(t, i, op, helpStack)
226:     else
227:         return labelDeq(t, i, op, helpStack)

228: unlabelOp(t, i, n) :
     op ← t.getOp(i)
229:     if n = null
230:         if op.name = Dequeue
231:             first ← head
232:             newHead ← first.node
233:             if first.t = t      ▷ Operation only labeled
     head
234:                 head.cas(first, new Head(newHead, null))
235:         return
236:     if op.name = Enqueue
237:         return unlabelEnq(t, i, op)
238:     else
239:         return unlabelDeq(t, i, op)

240: completeOp(t, i, n) :
241:     if op.name = Enqueue
242:         return completeEnq(t, i, n)
243:     else
244:         return completeDeq(t, i, n)
```

Figure 7.1: LOFT queue API implementation

the index field found in the transaction descriptor of every labeled node. If an index larger/equal to i is detected, then op was already labeled. If a different transaction is currently enqueuing, i.e. the $tail.next$ is labeled with a different transaction, we *help* it complete and return null to retry.

For a **dequeue** operation the *labelDeq* operation tries to label $head$ with a pointer to the transaction t. If it was already labeled by a different transaction t' we *help* t' complete and return null to retry. If we successfully label $head$ or it was already labeled with t we traverse the list searching for the first unlabeled node n. We label n with a transaction descriptor pointing to t and result set to $true$ (marking it as a successful dequeue) and return n. If all nodes in the queue are labeled, we find the first node enqueued by t and update its label with the index of the *dequeue* operation. If t did not enqueue nodes or all enqueued nodes were already dequeued by preceding *dequeues*, then this dequeue is a (failing) dequeue on an empty queue that should return null. To reflect this, we add a *dummy* node at the end of the list. The *dummy* node is labeled with t and its result is set to $false$ (no item to dequeue). We then return *dummy*. By inserting the dummy node to the list we prevent concurrent enqueues from executing so that we can guarantee the linearizability of this unsuccessful

dequeue. Otherwise, a concurrent transaction can enqueue items and complete before t completes, thus changing the result of t's dequeue. For the same reason, if during the traversal we encounter a concurrent enqueue by a different transaction we *help* it complete and return null to retry.

```
245: labelEnq(t, i, op, helpStack) :
246:     last ← tail
247:     next ← last.next
248:     if last ≠ tail ∨ t.status ≠ Pending
249:         return null
250:     if next = null          ▷ No transaction is enqueuing
251:         return addNode(t, i, op.data, last, true)
252:     nextT ← next.t
253:     if nextT = null
254:         tail.cas(last, next)                ▷ Fix tail
255:         return null
256:     if nextT.t ≠ t    ▷ Another transaction enqueuing
257:         help(nextT.t, next, helpStack)
258:         return null
259:     while next ≠ null do  ▷ Find last enqueued node
260:         last ← next
261:         next ← next.next
262:     lastDesc ← last.t
263:     if lastDesc = null ∨ lastDesc.opIndex > i
264:                             ▷ Operation already labeled
265:         return null
266:     if lastDesc.t ≠ t              ▷ Another transaction
267:         help(lastDesc.t, last, helpStack)
268:         return null
269:     if lastDesc.opIndex = i       ▷ Operation labeled
270:         return last
271:     return addNode(t, i, op.data, last, true) ▷ Add
        labeled node to end of queue
```

```
272: labelDeq(t, i, op, helpStack) :
273:     first ← head
274:     if t.status ≠ Pending
275:         return null
276:     if first.t = null
277:         newHead ← new Head(first.node, t)
278:         if ¬head.cas(first, newHead)     ▷ Label
279:             return null
280:         first ← newHead
281:     if first.t ≠ t                ▷ Another transaction
282:         help(first.t, null, helpStack)
283:         return null
284:     prev ← first.node
285:     next ← prev.next
286:     while next ≠ null do              ▷ Look for node
287:         desc ← next.t
288:         if desc = null      ▷ Found node to dequeue
289:             nextDesc ← new Desc(t, i, true)
290:             if next.t.cas(null, nextDesc)
291:                 return next           ▷ Labeled node
292:             return null                       ▷ Retry
293:         if desc.t ≠ t              ▷ Another transaction
294:             help(desc.t, next, helpStack)
295:             return null               ▷ Retry
296:         nextOp ← t.getOp(desc.i).name
297:         if nextOp = Dequeue
298:             if desc.i > i
299:                 return null
300:             if desc.i = i
301:                 return next
302:         if nextOp = Enqueue
303:                         ▷ Override existing descriptor
304:             desc.i ← i
305:             return next
306:         prev ← next                   ▷ Keep looking
307:         next ← prev.next
308:     return addNode(t, i, null, last, false)
309:             ▷ Empty queue - enqueue dummy node
```

Figure 7.2: LOFT queue label methods

completeOp(n,t,DSop): For an **enqueue** operation the *completeEnq* method fixes the tail to point to the node n given as parameter when n is the last node enqueued by t. It then removes the labels of all nodes enqueued by t. Thus, these nodes become visible to subsequent transactions. The

```
310: completeEnq(t, i, n) :
311:     desc ← n.t
312:     if desc = null ∨ desc.t ≠ t
313:         return
314:     if desc.i ≠ i ∨ n.data = null
315:             ▷ Overridden enqueue or dummy node
316:         fixTail(n)
317:         return
318:     last ← tail
319:     if isTailCorrect(last, t)          ▷ Tail updated
320:         n.t.cas(desc, null)            ▷ Clear n's label
321:         return
322:     if n.next = null     ▷ n is last enqueued node
323:         tail.cas(last, n)              ▷ Update tail to n
324:         next ← last.next
325:         while next ≠ null ∧ next ≠ n.next do
326:             ▷ Iterate from previous tail to new one
327:             desc ← next.t
328:             if desc ≠ null ∧ desc.t = t
329:                                ∧¬isDummy(next)
330:                 next.t.cas(desc, null)
331:                                        ▷ Clear label
332:             next ← next.next
```

```
333: completeDeq(t, i, n) :
334:     t.setResult(i, n.data)
335:     first ← head
336:     if isDummy(n)
337:         completeEnq(t, i, n)
338:     if first.t = null ∨ first.t ≠ t   ▷ Head unlabeled
339:         return
340:     if isLastDequeuedNode(n, t)
341:         newHead ← new Head(first.node, null)
342:         head.cas(first, newHead)      ▷ Update head
```

Figure 7.3: LOFT queue complete methods

tail pointer can also be updated If n is an enqueued dummy node or a node later dequeued. This is done to preserver the invariant that $tail$ is a successor of $head$. Otherwise, we might advance *head* to be a successor of *tail* (in *completeDeq*). When all *enqueue* operations of t are completed, then the $tail$ points to the last node enqueued by t as expected.

For a **dequeue** operation *completeOp* only needs to update *head* to point to the last node dequeued by t, which makes all dequeued nodes inaccessible from *head*. This update happens only when n is the last TNode dequeued by t (otherwise, *completeOp* does nothing). If so, we cas *head* to a new Head which points to n and has an empty transaction pointer (no label). We do not clear the labels (transaction descriptors) of dequeued nodes as they are no longer accessible from *head*. A special case where *completeOp* has to perform another action even when n is not necessarily the last node dequeued by t is when n is a dummy node. A dummy node is inserted at the end of the list in *labelOp* after the $tail$ node. Thus, when we later update head it will point to a node which is a successor of tail, hence the invariant that $tail$ is a predecessor of $head$ is not preserved. To preserve the invariant we fix the *tail* by calling *completeEnq* which points *tail* to n (Line 316).

unlabelOp(n,t,DSop): For an **enqueue** operation the *unlabelEnq* method needs to remove the nodes enqueued after the tail. It *cas*es *tail.next* to null (assuming $tail.next$ is labeled by t). Thus, all the nodes enqueued by the transaction are no longer accessible and the list returns to the same

```
343: unlabelEnq(t, i, n) :
344:     desc ← n.t
345:     if desc = null ∨ desc.t ≠ t
346:         return
347:     last ← tail
348:     next ← tail.next
349:     if tail ≠ last
350:         return
351:     nextDesc ← next.t
352:     if nextDesc = null
353:         return
354:     if nextDesc.t = t
355:         last.next.cas(next, null)
356:                 ▷ Remove all of t's enqueued nodes
```

```
357: unlabelDeq(t, i, n) :
358:     if n.data = null                    ▷ Dummy node
359:         unlabelEnq(t, i, n)
360:         desc ← n.t
361:     else if desc ≠ null ∧ desc.t = t
362:         n.t.cas(desc, null)             ▷ remove n 's label
363:     first ← head
364:     newHead ← first.node
365:     if first.t ≠ t
366:         return
367:     if isLastDequeuedNode(n, t)
368:         head.cas(first, new Head(newHead, null))
369:                                         ▷ clear head label
```

Figure 7.4: LOFT queue un-label methods

state as before the transaction.

To un-label t's **dequeue** operations we need to remove all labels added to the head node and other existing nodes by t. So, the *unlabelDeq* removes the label of the node n given as parameter. If n is also the last node labeled by a *dequeue* operation in t, we cas *head* to a new *Head* pointing to the same node with an empty transaction label (clearing its label). When n is the last node dequeued then it is guaranteed that all previous dequeues were already unlabeled and so clearing head's label allows for subsequent transactions to perform their dequeue operations. Once again a special consideration should be given to dummy nodes. When n is a dummy node we call *unlabelEnq* which physically removes it from the list (by *cas*ing *tail.next* to null) rather than un-labeling it, as it should not be in the queue.

```
370: isDummy(n) :
371:     return n.data = null

372: isTailCorrect(last, t) :
373:     next ← last.next
374:     if next = null
375:         return true
376:     desc ← next.desc
377:     if desc = null ∨ desc.t ≠ t
378:         return true
```

```
379: isLastDequeuedNode(n, t) :
380:     next ← n.next
381:     if next = null
382:         return true
383:     desc ← next.desc
384:     if desc = null ∨ desc.t ≠ t
385:         return true
386:     op ← t.getOp(desc.i)
387:     if op.name = Enqueue
388:         return true
389:     return false
```

Figure 7.5: LOFT queue aux methods

Chapter 8

Measurements

We compared the performance of our lock-free LOFT set (both with sorted operations and non-sorted operations, see Section 2.5) to the one described in TDSL [1]. TDSL employs a fine-grained lock-based approach for adding addresses to read and write sets, based on the underlying semantics of the data structure. We also evaluated a transaction-friendly skip list [5] running on top of a TL2 STM [6], implemented in the Synchrobench micro-benchmark suite [8]. This implementation reduces conflicts by deferring the work to a background thread that maintains the skip list, thus making it throughput-oriented. All implementations are in Java.

Each experiment consisted of two warm-up runs for JIT and 5 runs whose results were averaged. In Each run all threads executed operations/transactions for 10 seconds with keys uniformly selected from a predefined $range$. Each run was preceded with an initial insertion of $range/2$ randomly selected keys. We considered two different workloads: (1) a $read-oriented$ workload consisting of 90% *contains* operations, 5% *add* operations, and 5% *remove* operations; (2) a $write-oriented$ workload consisting of 10% *contains* operations, 45% *add* operations, and 45% *remove* operations. We ran our experiments on a machine with 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16-core processors (64 threads overall).

8.1 The LOFT Set Performance

Methodology We experimented with transactions comprised of various operations executed on a collection of four sets. Similar to the measurements made in [1], each transaction is assigned a random number of operations to perform from the range $[2, 7]$. Each operation is selected according to the designated workload, and the set on which it will be applied is randomly selected. We examine two different scenarios: one with a small key range (10^3) causing high contention and the other with a large key range (10^6) inducing low contention. For each scenario we ran the two workloads and

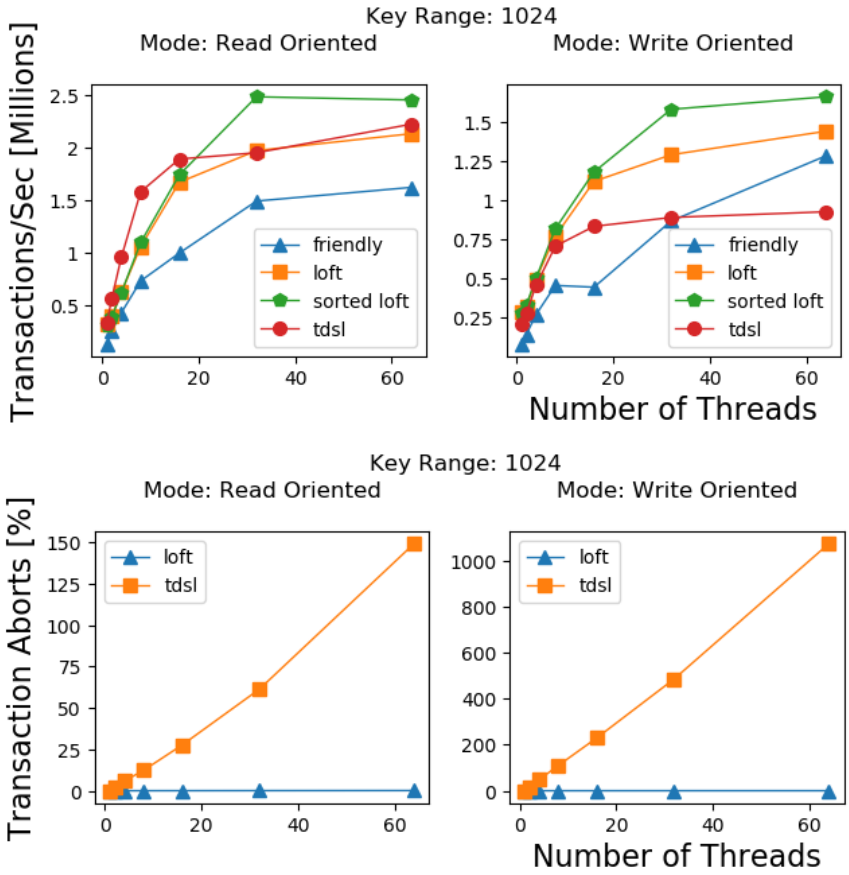

Figure 8.1: Throughput and aborts graphs of transactions over sets for high contention scenario.

present the results in Figures 8.1- 8.2.

Aborts The graphs in Figures 8.1- 8.2 present the abort rate with the workload distributions and the key range specified above. The abort rate of the Friendly Skip List over TL2 was omitted as it was much higher than that of TDSL. Sorted LOFT transactions were also omitted since their abort rate is 0, as there are no cyclic helping conflicts, hence no transaction gets rescheduled. We can see that for all scenarios and workloads the abort rate of LOFTtransactions is close to 0 (for each dependency cycle, a single transaction is aborted), while the abort rate of TDSL can reach 1000% when contention is high .

Throughput The graphs in Figures 8.1- 8.2 present the average throughput (Million transactions executed per second) with the workload distributions and the key range specified above. We

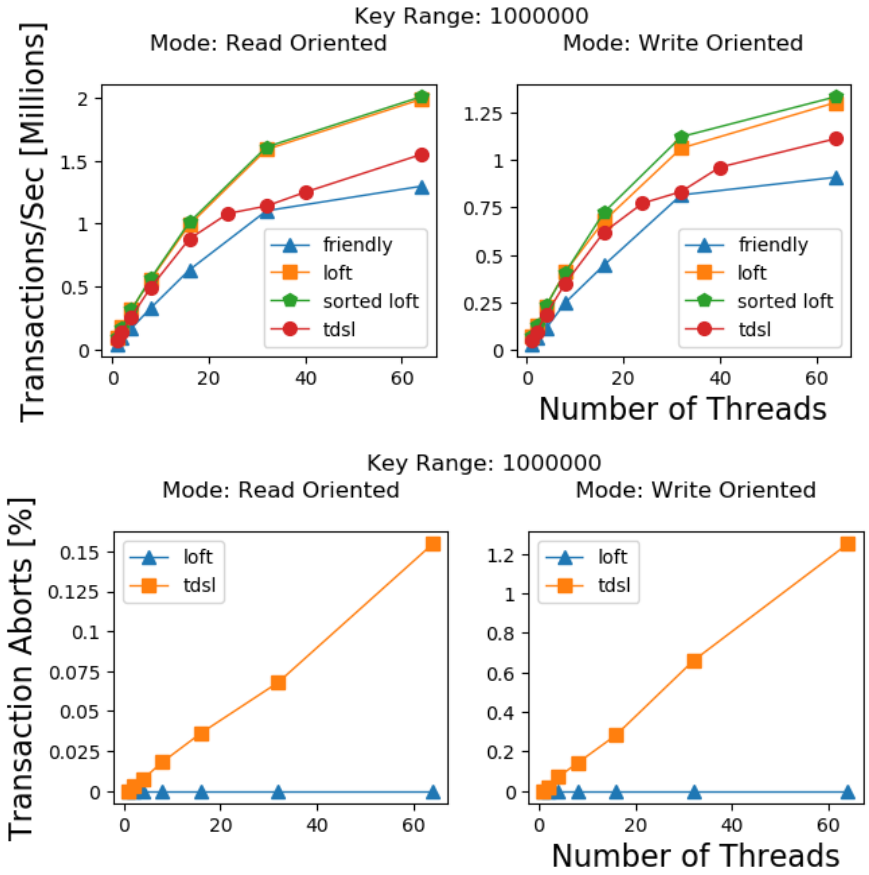

Figure 8.2: Throughput and aborts graphs of transactions over sets for low contention scenario.

can see that for a high contention scenario (shorter lists) our transactions are faster in write-oriented workloads, as the high contention induces many aborts in TDSL (up to 10 aborts per transaction) and the Friendly Skip List, while our solution hardly aborts. However, in the read-oriented workload our transactions performance is similar to that of TDSL since *contains* operations inside our transactions have to set a transaction label on the relevant node, or possibly create a node if it does not exist. This leads to more CAS operations performed on the data structures, compared to TDSL, offsetting the advantage of reduced aborts. We still outperform the friendly skip list significantly. In both workloads the sorted LOFT transactions outperform the non-sorted ones, and also TDSL (for many threads). The sorting of operations eliminates cyclic helping conflicts, thus there is no need for re-executions. For the low contention scenario, our algorithm outperforms the other algorithms for both the read and write oriented workloads. While all algorithms abort less, TDSL still aborts more frequently than LOFT which requires more traversals over the list and incurs smaller through-

put. We see that sorting the operations does not increase throughput, as there are fewer conflicts and fewer helping cycles to avoid using sorting.

8.2 The LOFT Set Singleton Performance

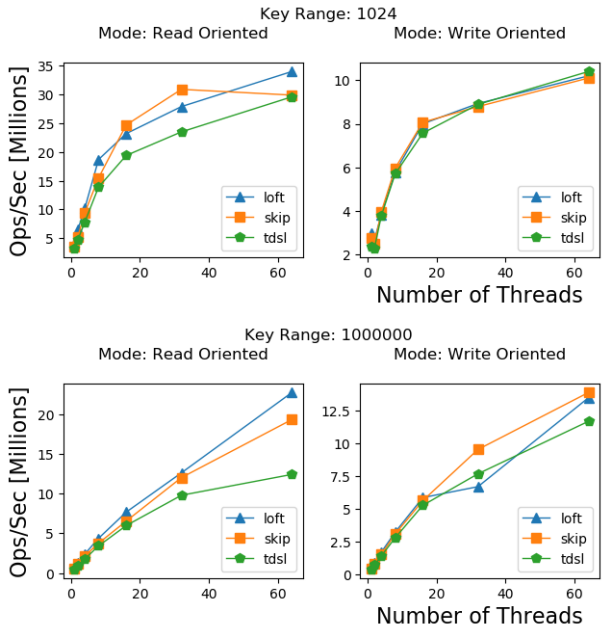

Figure 8.3: Throughput graphs of set singleton operations . Each row contains a different key range

Methodology As singletons (stand-alone operations) are the basis of a transactional data structure, we strove to have as little impact as possible to their performance. We therefore compared our singleton operations to those of TDSL and to the java implementation of a concurrent skip list map. We examined the same two contention scenarios, each with the two workloads and present the results in Figure 8.3.

Throughput We see in Figure 8.3 that in the high contention scenario all algorithms perform roughly the same and scale well, with our implementation out-performing the other algorithms in the write-oriented scenario. For the low contention scenario, our results are close to those of TDSL. However, the concurrent skip list achieves better results. These results show that supporting transactions has some overhead on singleton operations, however it is likely that improving the

skip-list index will result in better performance, which is comparable to that of the highly tailored concurrent algorithm.

8.3 The LOFT Queue Performance

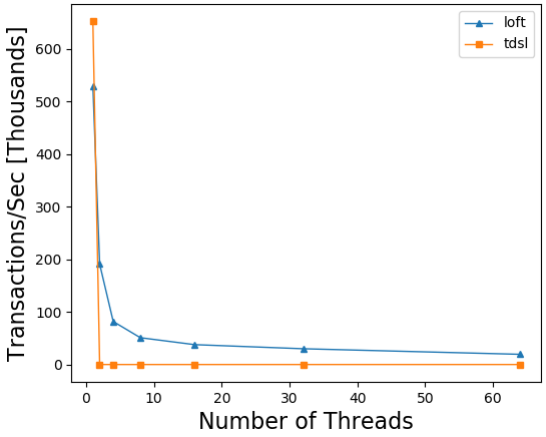

Figure 8.4: Throughput graphs of transactions over queues.

Methodology We experimented with transactions comprised of *enqueue* and *dequeue* operations executed on a collection of four queues. Each experiment consisted of two warm-up runs for JIT and 5 runs whose results were averaged. In Each run all threads executed transactions for 10 second. Each run was preceded with an initial insertion of 500000 elements to each queue. Similar to the measurements made in Section 8.1, each transaction is assigned a random number of operations to perform from the range $[2, 7]$. Each operation is selected uniformly, and the queue on which it will be applied is randomly selected. We present the results in Figure 8.4.

Throughput The graphs in Figure 8.4 present the average throughput (Thousand transactions executed per second). A queue is inherently not a very scalable data structure, as there is much contention over accessing its head and tail pointers. Additionally, since operations in a transaction are performed on multiple queues, there are many conflict cycles which trigger many aborts. Consider a scenario where two transactions *enqueue* to one queue and *dequeue* from another queue in the opposite order. In the TDSL implementation both threads abort their transaction, while our implementation aborts only one of the transactions, and schedules it for an upcoming execution. The results show that the TDSL queue implementation hardly succeeds to complete transactions

for multi-threaded executions. Performing operations of multiple threads to multiple queues induces aborts to almost all transactions, and only a handful of transactions successfully commit. In contrast, our implementation commits thousands of transactions even for 64 threads.

Chapter 9

Correctness

This section contains the Linearizability proof of the proposed LOFT framework, specifically using LOFT sets.

Linearization Points We define the Linearization points of a transaction and the transactional set's singleton methods:

Definition 1. *We define the Linearization point of a transaction t to be the update of its status to* `Done` *(Line 20)*

Definition 2. *We define the Linearization point of a contains and get singletons to be one of the following Lines:*

- *For a false result for contains or a null result for get: (1) saving $n.transaction$ in Line 169 and it points to removeT, (2) the find method returns a node n s.t. $n.key \neq key$ (Line 165), (3) checking $desc.t$'s status is* `Pending` *(Line 175) and the operation op added n, and (4) a true return value when checking if n is marked (Line 183).*
- *For a true result for contains or a non-null result for get: (1) a false return value when checking if n is marked (Line 183), (2) saving $n.transaction$ in Line 169 and it is null, and (3) checking $desc.t$'s status is* `Pending` *(Line 175) and the operation op did not add n.*

Definition 3. *We define the Linearization point of an add singleton to be one of the following Lines:*

- *An unsuccessful add: checking if n is marked returns false (Line 144).*
- *A successful add: the cas in Line 149 succeeds.*

Definition 4. *We define the Linearization point of a remove singleton to be one of the following Lines:*

- *A successful remove: the cas in Line 131 succeeds (labeling n).*
- *An unsuccessful remove: the call to find in Line 153 returns a node n s.t. $n.key \neq key$.*

Other Definitions

Definition 5. *we define op_i to be the operation with index i in some transaction t.*

Definition 6. *For two operations op_i and op_j in transaction t we say that op_i precedes op_j if $i < j$.*

Definition 7. *For two operations op_i and op_j in transaction t we say that op_j follows op_i if op_i precedes op_j.*

Definition 8. *An operation op_1 in transaction t_1 is conflicting with an operation op_2 in transaction t_2 if they do no commute [11].*

Definition 9. *For an operation op in a transaction t performed on a data structure ds we say that op is labeled if all threads executing conflicting operations belonging to different transactions must help t complete before preceding with their own execution.*

9.1 Linearizability Proof

We prove the Linearizability of the LOFT framework. We start by specifying the requirements from the LOFT data structures. Each LOFT data structure must satisfy the following claims:

Claim 10. *A labelOp method called with parameter op returns a TNode $n \neq null$ only if op is labeled.*

Claim 11. *The TNode n returned from a $labelOp$ method called with operation op contains op's result and the information needed to complete or un-label op.*

Claim 12. *After an operation op in a transaction t becomes labeled, op's result is unchanged until completeOp is called on it.*

Claim 13. *A labelOp method called with parameter op noticing a labeled conflicting operation $op' \in t'$ must call the transaction engine's help method with t' and retry its operation.*

Claim 14. *completeOp called with TNode n, transaction t and an index i of an operation $op \in t$ completes op according to its result, makes op's effect visible and removes op's labeling.*

Claim 15. *unlabelOp called with TNode n, transaction t and an index i of an operation $op \in t$ reverts any change performed by op to the data structure and removes op's labeling.*

Claim 16. *Concurrent and subsequent calls to completeOp with the same parameters leave the data structure in an equal state.*

Claim 17. *Concurrent and subsequent calls to unlabelOp with the same parameters leave the data structure in an equal state.*

Claim 18. *Concurrent and subsequent calls to labelOp with the same parameters do not result in an inconsistent state of the data structure.*

Claim 19. *The data structure does not change the transaction's status.*

We continue with the linearizability proof of the LOFT framework.

Claim 20. *For a* `Pending` *transaction t, the labelOp method is called with parameter op_j after operations preceding it were labeled.*

Proof. We prove by induction:

- Base: for $j = 0$ no operation preceded op_j and the Claim holds.
- Step: we assume the claim holds for $j-1$ and prove it for j. The *labelOp* method is called with op_j when $t.index = j$. $t.index$ is initialized to 0, and is incremented by 1 each time, meaning its value had to be $j-1$ before it was incremented to j. In order for $t.index$ to be incremented to j, a call to *labelOp* with parameter op_{j-1} must return a non-null node, which happens only if op_{j-1} is labeled (Claim 10). In addition, from the assumption *labelOp* with parameter op_{j-1} is called after all operation preceding it were labeled. Thus, all operations preceding op_j were labeled.

□

Claim 21. *A transaction t's status is changed to* `Done` *only after all operations in t are labeled.*

Proof. The update of t's status to `Done` (t's linearization point) by thread P happens only in the end of the *executeTX* method (Line 20) since a transaction's status is only changed in the transaction engine code and not in the data structures code (Claim 19). This code is reached when $t.index = t.size$ or if t's status was already changed to `Done` by another thread (Line 16). If $t.index = t.size$, then the last operation in t ($op_{t.size-1}$) was passed to $labelOp$ which returned a non-null TNode. Thus, t's last operation was labeled (Claim 10), and from Claim 20 this means that all operations preceding it were also labeled. Hence, all operations in t were labeled. Thus, our claim holds. □

Definition 22. *A helping cycle $C = t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$ consists of n transactions, each calling the help method with the next transaction as the parameter, to remove the label of a conflicting operation.*

Definition 23. *A helping cycle C is considered resolved if a conflicting operation op which caused some transaction t in C to call help is no longer labeled.*

Claim 24. *A helping cycle C is resolved only after one of the transactions in the cycle changes its status to* `Replaced`.

Proof. We assume that C is resolved and none of the transactions in C changes status to `Replaced`. For C to be resolved there exists a conflicting operation $op_i \in t_i$ which required transaction $t_{i-1} \in C$ to call *help* with transaction $t_i \in C$, and op_i is no longer labeled. An operation becomes unlabeled in *unlabelOp* or *completeOp*. *unlabelOp* can be called by: (1) *replace* which is

executed only for transactions with `Replaced` status, and from the assumption t_i's status is not `Replaced`; or (2) *help* called with a mislabeled node, which is not the case because then t_{i-1} would have been able to continue labeling its operations and not be part of C. This means that op_{i+1}'s label is removed by *completeOp*, however *completeOp* is only called for a transaction with status `Done`, which happens after all operations are labeled. However, t_{i+1} is in the help cycle C, which means that there exists a conflicting operation $op_{i+2} \in t_{i+2}$ preventing t_{i+1} from labeling one of its operations. Thus, op_{i+1}'s label is not removed, and C cannot be resolved.

We now prove that C is resolved when we change the status of a transaction $replaceT = t_i \in C$ to `Replaced`. t_i's status is changed to `Replaced` in *resolveCycle*. Before we change t_i's status we save its predecessor $nextT = t_{i-1} \in C$ in t_i's transaction object. After changing t_i's status to `Replaced` we call *replace* which removes the labels of t_i's operations and then execute t_{i-1}. Thus, the conflicting operation causing t_{i-1} to help t_i is no longer conflicting, and t_{i-1} can proceed with its next operations which makes C resolved. □

Claim 25. *Once a transaction object's status is set to* `Done` *it does not change.*

Proof. For a transaction object t, its status is initialized with `Pending` and is never set to `Pending` again. This means that after t's status is changed to `Done` it does not change back to `Pending`. This only leaves the `Replaced` status, which is set in *resolveCycle* if t is part of a helping cycle (Line 46). The *resolveCycle* method validates that all transactions in the cycle are still `Pending` (Line 39) so for the method to change t's status to `Replaced` t's status cannot be `Done` prior to this validation. Thus, we assume t's status is changed to `Done` by another thread $T1$ between Lines 39-46. t's status is changed to `Done` only in the *executeTx* method after all of its operations are labeled. For t's operations to successfully be labeled, it must finish helping all other transactions performing conflicting operations. However, t is part of a helping cycle which according to Claim 24 is resolved only after t's status changes to `Replaced` (Line 46), in contradiction to our assumption. Thus, our claim holds. □

Claim 26. *Once a transaction object's status is set to* `Replaced` *it does not change.*

Proof. For a transaction object t, its status is initialized with `Pending` and is never set to `Pending` again. This means that after t's status is changed to `Replaced` it does not change back to `Pending`. This only leaves the `Done` status, which is set in *executeTX* if t's status is not `Replaced` (Line 20). Thus, when a thread notices a status of `Replaced` it does not change it. We assume that a thread T changed a `Replaced` status to `Done`. This means that T read a status different from `Replaced` in Line 9 and the status was then changed to `Replaced` by another thread T_2 before T changed the status to `Done` in Line 20. From Claim 25 the status T read in Line 9 cannot be `Done` because this status can not change (and we know T_2 changed it to `Replaced`). Hence, the status T read was `Pending`. For T to reach Line 17 while t is still `Pending` it must finish the loop which labels all of t's operations (otherwise it would break in line 16). This means that all of t's operations are labeled. However, if t's status is later set to `Replaced` by T_2, it means that t is

part of a helping cycle, which is resolved only after changing t's status to `Replaced`. Thus, the labeling of one of t's operations could not be completed since there exists a conflicting operation of another transaction. Thus, T could not have finished the labeling loop and this contradicts our assumption and the claim holds.

□

Claim 27. *unlabelOp and completeOp will not be called with the same parameters.*

Proof. we will prove that *unlabelOp* cannot be called after *completeOp* with the same parameters and vice versa:

unlabelOp is called with a transaction object t and TNode n either by *replace* when t's status is `Replaced` or by *help* when n is a mislabeled node. If t's status is `Replaced` then from claim 26 this status will not change and so *completeOp* cannot be called with n and t as it is only called after setting t's status to `Done`. If n is a mislabeled node, then it is not part of t's nodes array and thus *completeOp* will not be called with n as parameter. Thus, *completeOp* cannot be called after *unlabelOp*.

Similarly, if *completeOp* is called with a transaction object t and TNode n, then t's status is set to `Done` and $n \in t.nodes$. From Claim 25 t's status will not change, hence *unlabelOp* won't be called with t and n by *replace* as t's status must be `Replaced` prior. *unlabelOp* will also not be called with t and n by *help* as n is not a mislabeled node (it is part of t's nodes array). Thus, *unlabelOp* cannot be called after *completeOp*. □

Observation 28. *completeTransaction is called for a transaction t only after setting t's status to* `Done`.

Claim 29. *For a transaction t comprised of operations $op_1, \ldots, op_n$ completeTransaction completes all of t's operations s.t. their effect is visible to other threads, and they are no longer labeled.*

Proof. *completeTransaction* is called after setting t's status to `Done` (Observation 28), and from Claim 21 this means that *completeTransaction* is called after $op_1, \ldots, op_n$ are labeled. *completeTransaction* then calls *completeOp* with each of the TNodes belonging to $op_1, \ldots, op_n$ and from claim 14 this means that $op_1, \ldots, op_n$ are completed according to their result, which does not change until *completeOp* is called (Claim 12). In addition, the effect of each operation is visible to other threads, and they are no longer labeled. We notice that *completeOp* can be called by other threads concurrently but from Claim 16 this leaves the data structures in an equal state. Thus, *executeTx* returns only when t's operations are completed, and our claim holds. □

Claim 30. *All operations of a transaction t seem to take effect on their data structure (as observed by other threads) only after t's linearization point.*

Proof. We assume in the form of a contradiction that there exists an operation $op \in t$ s.t. op's effect is visible to a different thread P prior to t's linearization point. For P to observe op's result, op's label must be removed. Otherwise, from Claim 13 P must call *help* with t and retry its operation.

op's label is removed in 2 cases: (1) *unlabelOp* was called with op as parameter and removed any changes *labelOp* performed to the data structure (Claim 15), thus op's effect on the data structure is not visible to P which contradicts our assumption. (2) *completeOp* was called with op as parameter and completed op's effect on the data structure. *completeOp* is called only in the *completeTransaction* which from Observation 28 is called only after t's status is set to `Done` (t's linearization point) which contradicts out assumption. Thus, our claim holds.

□

Claim 31. *A call to the executeTX method with transaction object t which status will not become* `Replaced` *returns after all of t's operations are labeled and completed.*

Proof. The *executeTX* method iterates over all unlabeled operations in t (Line 8) and calls *labelOp* for each operation. For the iteration to finish, either $t.index$ reaches $t.size$ or t's status is no longer `Pending`. For $t.index$ to reach $t.size$, a call to $labelOp$ for each unlabeled operation $op \in t$ must return a non-null node n which from Claim 10 means op is now labeled. It is possible that *labelOp* is called concurrently by other threads but from Claim 18 it does not cause an inconsistent state of the data structures. When the iteration finishes, we check the status of t. Since it is not `Replaced`, t's status is `Pending` or `Done`, and we call *completeTransaction* to complete all of its operations (Claim 29) and return. □

Claim 32. *A call to the executeTX method with a transaction object t returns after all of t's operations are completed and visible to other threads.*

Proof. We prove by induction on the number of *replace* invocations generated by *executeTX* on transaction objects containing the same operations as t.

Base: for 0 *replace* invocations we learn that t's status was not `Replaced`, otherwise the *replace* method would have been invoked. Thus, from Claim 31 the claim holds.

Step: We assume that for i *replace* invocations the $executeTX$ method with transaction object t returns after all of t's operations are completed and visible to other threads, and prove for $i+1$. Since there are $i+1$ invocations of *replace*, then the *executeTX* method must invoke the *replace* method. This method calls $unlabelOp$ for each of t's operations which from Claim 15 removes their labeling. Then, a call to *executeTX* is made with a new transaction object $newT$ containing t's operations and another transaction $nextT$. Since $nextT$ is a transaction with an operation conflicting with one of t's operations, it does not contain t's operations. Thus, all other i invocations of *replace* on transaction objects containing the same operations as t are generated from the call to *executeTX* with $newT$. From the assumption, the $executeTX$ method with transaction object $newT$ returns after all of $newT$'s operations are completed, hence t's operations are completed as t and $newT$ contain the same operations. Thus, the claim holds. □

Claim 33. *All operations of a transaction t seem to happen atomically to other threads, i.e. no thread sees the effect of some operations of transaction t and not the effect of other operations in t, and can proceed with its operations.*

Proof. We assume in the form of a contradiction that a thread P sees the effect of an operation $op_1 \in t$ and does not see the effect of an operation $op_2 \in t$. For P to see the effect of op_1, op_1 must be completed which happens only after calling *completeOp* with op_1 as parameter. *completeOp* is called in *completeTransaction* which from Observation 28 happens after t's status is set to `Done`. From claim 21 t's status is set to `Done` only after all operations in t are labeled. Thus, op_2 must be labeled or completed. Since P does not see the effect of op_2, then it is not completed, hence it is labeled. From Claim 13 P must call *help* with t as parameter, which calls *executeTX* and returns only after all of t's operations are completed (Claim 32). Thus, op_2 is completed and P sees its effect prior to proceeding with its own operations which contradicts our assumption, and so our claim holds. □

Claim 34. *The result of each operation in a transaction t matches the state of the data structures at t's linearization point.*

Proof. From Claim 12 an operation's result does not change until *completeOp* is called on it. *completeOp* is called in *completeTransaction* which from Observation 28 is called after t's status is set to Done (t's linearization point). In addition, t's status is set to `Done` after all operations in t are labeled (Claim 21. Thus, all operations are labeled and their results do not change, hence the state at t's linearization point corresponds to the results of t's operations. □

Theorem 35. *The transaction's engine execute method executes all operations given as input atomically, according to the data structures state at t's linearization point.*

Proof. The *execute* method creates a transaction object t containing the operations and calls *executeTX* with t as parameter and an empty help stack. This method completes all of t's operations (Claim 32) atomically (Claim 33), according to the data structures state at t's linearization point (Claim 34). □

9.2 The LOFT Set Linearizability Proof

We prove the claims required from the LOFT Set for the linearizability proof. We begin with some definitions:

Definition 36. *A node n is a successor of a node n' if there exists a path of zero or more $next$ pointers from n' to n. Similarly, a node n is a predecessor of a node n' if n' is a successor of n.*

Definition 37. *A node n is reachable if it is a successor of the $head$ node.*

Definition 38. *We define a node to be logically in the list if it is reachable, its transaction pointer is empty and its next pointer is unmarked.*

Definition 39. *We define a node to be labeled with transaction t if its transaction descriptor points to t.*

Claim 40. *An operation op in transaction t is labeled if there exists a node n with key $op.key$ which is labeled with t.*

Proof. From definition an operation op is labeled if all conflicting operations must help t before completing. For the set data structure two operations conflict if they are performed on the same key as they require manipulating the same node. Thus, when a node n is reachable and labeled with transaction t, all other transactions performing operation on the same key must help t as the *find* method will return n (Claim 44 and its transaction descriptor points to t, hence *help* will be called with t as parameter in *labelNode* (Line 123. Thus, op is labeled.

□

We recall that a node n in the list serves also as a TNode. All the information required from a TNode is present in the node's fields.

Claim 41. *The labelNode method returns a node $n \neq null$ only if n is labeled with transaction t.*

Proof. *labelNode* method returns a node $n \neq null$ either in Line 122 after checking that n is labeled with transaction t, or after successful *cas* operations of n's transaction descriptor to one pointing to t (Lines 131 and 128).

□

Claim 42. *A labelOp method called with parameter op returns a node $n \neq null$ only if op is labeled.*

Proof. The *labelOp* method returns $n \neq null$ in one of 2 cases. We show that for each case op is labeled:

1. *labelOp* calls *labelNode* which returns $n \neq null$ (Line 61). According to Claim 41 this happens only when n is labeled with transaction t. n has a key equal to op's key, and was returned from the *find* method so it is reachable from *head*. Thus, op is labeled.
2. We add $newNode$ to the list (Line 67 with its transaction descriptor pointing to t and key equal to op's key. $newNode$ is the successor of $pred$ (following the successful *cas*) and $pred$ was returned from *find* so it is reachable from *head* and hence $newNode$ is also reachable. $newNode$ is reachable and labeled with t, hence op is labeled (Claim 40).

□

Observation 43. *An operation op becomes labeled only during the labelOp method when labeling a node n containing op's key with transaction t. This happens either when adding a new node (Line 67), or in the labelNode method (Lines 128 and 131) called from labelOp.*

Claim 44. *The find method called with key returns two nodes $pred$ and n s.t. $pred.key < key$, $n.key \geq key$ and at some point between the invocation of find to its completion $pred.next = curr$*

Proof. We prove the Claim holds for each iteration of the while loop: In the first iteration of the while loop $pred = head$ and by definition $head.key = -\infty$ and thus $pred.key < key$. In Line

105 $n \leftarrow pred.next$ and if the return statement in Line 112 is executed in the first iteration then $n.key >= key$. Thus, the Claim holds for the first iteration.

For all other iterations i s.t. $i > 1$, $pred_i$ is equal to n_{i-1} for which $n_{i-1}.key < key$ (otherwise the while loop would have ended in iteration $i - 1$). In addition, $n_i = n_{i-1}.next = pred_i.next$ (Lines 107 and 114). If the return statement in Line 112 is executed in iteration i then $n_i.key >= key$. Thus, the Claim holds for iteration $i > 1$.

□

Claim 45. *A node n with an unmarked $next$ pointer is reachable.*

Proof. Proof by structural induction on the list:

Base: In the initial state of the list, there are only 2 nodes $head$ and $tail$, both with unmarked $next$ pointers. $head$ is a trivial successor of $head$, so it is reachable. We defined $head.next = tail$ so $tail$ is also a successor of $head$, and thus reachable.

Step: We assume that all nodes with unmarked $next$ pointers are reachable. We show that for every operation affecting reachability of nodes in the list the Claim still holds:

- We add a new node n to the list (Line 67). This means that the CAS in Line 67 succeeds. For the CAS to succeed $pred$'s $next$ pointer has to be unmarked, so from the assumption $pred$ is a successor of $head$. $pred$'s $next$ pointer now points to $newNode$ so $newNode$ is also a successor of $head$ and the Claim holds for $newNode$. n and its successors are still reachable (through $newNode$), so the Claim still holds for them as well. $pred$'s predecessors state hasn't changed, so the Claim holds for them as well.
- We remove a node n from the list (Line 109) when n's $next$ pointer is marked (Line 108. This is done by CAS-ing its predecessor $pred$ to its successor $succ$ and so the Claim for it still holds. All predecessors of n are still reachable (no $next$ pointer in their path was changed). The CAS in Line 109 also validates that $pred$ is still unmarked, so from the assumption it is a successor of $head$. Thus, $succ$ and all of its successors remain reachable from $head$ (through $pred$).

□

Claim 46. *For a node $n \neq tail$ which is reachable, its $next$ pointer points to a node n' s.t. $n.key < n'.key$*

Proof. Proof by structural induction on the list:

Base: In the initial state of the list, there are only 2 nodes $head$ and $tail$. We defined $head.key = -\infty, tail.key = \infty$ so $head.key < tail.key$, and the Claim holds.

Step:

- We add a new node n to the list (Line 67). The $newNode$'s $next$ pointer points to $curr$ which satisfies $curr.key > newNode.key$ (Claim 44 + Line 62). From Claim 44 we learn that $pred.key < newNode.key$. If the CAS in Line 67 indeed succeeds, then $pred$ is unmarked and from Claim 45 is reachable, which makes $newNode$ and $curr$ also reachable. Thus, the Claim

still holds for $pred$, $newNode$ and $curr$. The $next$ pointers of other nodes weren't changed, so the Claim still holds for them as well.

- We remove a node n from the list (Line 109) by CAS-ing its predecessor $pred$ to its successor $succ$. The only $next$ pointer of a reachable node to change is that of $pred$ which now points to $succ$. From the assumption $pred.key < n.key < succ.key$ so the Claim still holds for $pred$.

□

Claim 47. *A reachable node n has a key larger than all of its predecessors*

Proof. Prove by induction on i - the distance of n from *head* (number of $next$ pointers traversed). **Base**: for $i = 1$ we get that $n = head.next$, so it is reachable and from Claim 46 we get that $n.key > head.key$. $head$ is n's only predecessor so the Claim holds. **Step**: we assume the Claim holds for $i-1$ and prove for i. n is reachable so there exists some reachable node n' s.t. $n = n'.next$. n's distance from $head$ is i so n''s distance is $i - 1$ and from the assumption we get that n' has a key larger than all of its predecessors. From Claim 46 we get that $n.key > n'.key$ and so n has a key larger than all of its predecessors and the Claim holds. □

Claim 48. *If the node n returned from the find method has a key s.t. $n.key > key$, then no item with key exists in the list.*

Proof. From Claim 44 the *find* method returns two items $pred$ and n s.t $pred.key < key$ and $n.key \geq key$. From Claim 47 all of $pred$'s predecessors have a key smaller than $pred.key$ so all of n's predecessors have a key lower than key. From Claim 47 n's successors have a key larger than $n.key$ and so larger then key. If $n.key > key$, then $n.key \neq key$ and all other nodes have a greater or smaller then key and the Claim holds. □

Observation 49. *A transaction t labels a node n only when n's transaction pointer is null, as all cas operations of n's transaction pointer are from null to t.*

Claim 50. *A node n with a marked $next$ pointer is labeled.*

Proof. The marking of a node happens in the following Lines:

1. In *completeOp* when marking n (Line 95): *completeOp* is called on nodes labeled with t. Clearing a node's label happens in *completeOp* (Line 97) only if n should not be marked, but it is supposed to be marked. From Claim 49 no other transaction can label n. Thus n remains labeled with t when it is marked.
2. In *unlabelOp* when marking n (Line 83): *unlabelOp* is called on nodes labeled with t. Clearing a node's label happens in *unlabelOp* (Line 97) only if n existed prior to t, but it was marked since it did not exist, so. Meaning From Claim 49 no other transaction can label n. Thus n remains labeled with t when it is marked.
3. In *labelNode* when marking a node n s.t. n's is labeled with $removeT$ (Line 119). We mark the node and do not change its transaction pointer, and *completeOp* is never called with that node as it is not part of a transaction, so the transaction pointer cannot be cleared.

4. In the singleton *remove*, when it is successful in removing n (Line 157). The marking happens after the *labelNode* method returns *success* which happens only after it set n's transaction to $removeT$. *completeOp* is never called with that node as it is not part of a transaction, so the transaction pointer cannot be cleared. □

Claim 51. *The TNode $newNode$ returned from a $labelOp$ method called with operation op contains op's result and the information needed to complete or un-label op.*

Proof. We notice that each operation on the set requires adding a labeled node or labeling an existing node. For some operations we even add a dummy node just for labeling. Thus, the information we need in order to complete or unlabel op is whether op was performed successfully. i.e. an *add* operation is performed successfully when no other node with the same *key* is logically in the list. On the other hand, *contains, remove* and *get* operations are performed successfully when a node with the same *key* is logically in the list. We store this information in the transaction descriptor's $result$ field.

From the description above we notice that op's result equals this $result$ field for all operations except get, which requires $newNode$'s $data$ (the data associated with op's key), or null if $result$ is false. $newNode$'s $data$ is stored in $newNode$'s $data$ field and does not change (no API for changing a node's data). Thus, the descriptor's $result$ field combined with $newNode$'s data field contain op's result.

We next show that the descriptor's $result$ field is indeed correct. This field is updated in several places in the code. We show that it is correct for all operations:

1. In Line 128 we label n with a descriptor with $result$ field set to *false*. This happens when a node with key is reachable and we call *labelNode* to label it. The *labelNode* method reaches Line 128, which means that $shouldExist$ is *false* and n's next pointer is unmarked. $shouldExist$ is *false* when it is required that n does not logically exist in the list for the operation to succeed. This is required for *remove, contains,* and *get* operations. n's next pointer is unmarked, and it was unlabeled in Line 116 meaning that there was a point in which n was unmarked and unlabeled so it logically exists in the list. Thus, the result saved in its descriptor indeed indicates whether op was performed successfully.
2. In Line 64, we set *desc*'s $result$ to $\neg shouldExist$. We get to this Line when the *find* method returns a node n s.t. $n.key \neq key$ and from Claim 48 no node with key is logically in the list. If no node with key exists then the to determine the result we simply need to check whether the operation required for such a node to exist ($shouldExist$). Thus, we update the result accordingly and it indeed indicates whether op was performed successfully.
3. In Line 131, we set the $desc$'s $result$ to *true*. This happens when the operation requires that n logically exists in the list ($shouldExist$) and we successfully *cas* its transaction pointer to t.

This *cas* succeeds when n is unlabeled, thus from Claim 50 it is unmarked. From Claim 45 n is reachable and is logically in the list. Thus, op was indeed performed successfully.

□

Claim 52. *After an operation op in transaction t becomes labeled, op's result is unchanged until completeOp is called on it.*

Proof. From definition an operation op is labeled if there exists a node n with the operation's key which is labeled by t. Thus, other transactions trying to label n check it is unlabeled before and if it is they do not change it until it is unlabeled (Claim 49), which only happens in *completeOp*. In addition the *find* method removed a node only if it is marked, which happens only in *completeOp*. Thus, since an operation's result is determined using the logical state of a node n with its key and no other transaction changes this logical state until *completeOp* is called then the result remains unchanged. □

Claim 53. *completeOp called with node n, transaction t and parameters $DSop$ of operation op completes op according to its result.*

Proof. From Claim 52, we learn that op's result is unchanged until *completeOp* is called on it. We show that for each operation the *completeOp* completes it correctly:

- For an *add* operation with result *true*, n was added to the list since no node with key is logically in the list. Thus, we clear its label making it logically in the list. Otherwise, if op's result is false then n is an existing unmarked node with key labeled by t. n's state should not be changed as it was labeled only to ensure atomicity of all operations in t, so we clear its transaction pointer making it logically in the list again.
- For a *remove* operation with result *false* n is a dummy node added to the list by t since no node with key is logically in the list. Thus, n should not be logically in the list, and we mark its $next$ pointer making it logically not in the list. Otherwise, op's result is true, meaning n is an existing unmarked node with *key* labeled by t. n should be removed, so we mark its $next$ pointer making it logically not in the list.
- For *contains* and *get* operations with result *false* n is a dummy node added to the list by t since no node with key is logically in the list. Thus, n should not be in the list, and we mark its $next$ pointer making it logically not in the list. Otherwise, op's result is true, meaning n is an existing unmarked node with *key* labeled by t. n's state should not be changed as it was only labeled to ensure linearizability, so we clear its label.

□

Claim 54. *A call to the unlabelOp method with transaction t, operation op and node n reverses $labelOp$'s effect on n.*

Proof. *unlabelOp* performs its operation only if n is not null (Line 72) and it is labeled by t, meaning its descriptor points to t (Line 77). These conditions are met only if *labelOp* was called with op and returned n (Claim 10). *unlabelOp* checks if n existed prior to its labeling by op. n existed if (1) op was an unsuccessful *Add* operation which then labeled the already existing node n or (2) op was a successful *remove, contains* or *get* operation which labeled the existing node n. If n existed, we remove its label (added by *labelOp*). Otherwise, n was inserted to the list by *labelOp* and so it is logically deleted by marking its next pointer. In both cases the effect of *labelOp* is reversed and the claim holds. □

Claim 55. *Concurrent and subsequent calls to completeOp with the same parameters leave the data structure in an equal state.*

Proof. The *completeOp* method decides whether to mark a node or un-label it according to its parameters. Marking a node several times has the same effect as marking it a single time. Un-labeling a node is done using a *cas* of its transaction descriptor to null. Since it won't be labeled with this descriptor again this *cas* will only succeed once and after that won't have any effect. Thus, for the same parameters to *completeOp* it will perform the same operation which leaves the data structure in an equal state if called several times. □

Claim 56. *Concurrent and subsequent calls to labelOp do not result in an inconsistent state of the list.*

Proof. The *labelOp* either labels a node with key or adds a new node with key and labeled with t. Both of these operations can be performed on several nodes (when a race occurs). However, only the first node is saved in t, and this is one we perform *completeOp* on. Operations on all other nodes are reverted in *unlabelOp* either by marking newly added nodes or clearing the transaction pointer for nodes logically in the list. Thus, the Claim holds. □

Claim 57. *A labelOp method called with parameter op noticing a labeled conflicting operation $op' \in t'$ must call the transaction engine's help method with t' and retry its operation.*

Proof. As described above two operations on a set conflict if they operate on the same key. If op' is labeled, then from Claim 40, there exists a reachable node n with op''s key. Thus, when *labelOp* is called with op and calls the *find* method with op's key, it will return n (Claim 44) as $op.key = op'.key$ since they conflict. When a node with op's key is returned from *find*, it calls the *labelNode* method which will reach Line 123 (all prior validations will pass) and call *help* with t'. Thus, our claim holds. □

Claim 58. *Concurrent and subsequent calls to unlabelOp with the same parameters leave the data structure in an equal state.*

Proof. Similarly to *completeOp*, the *unlabelOp* method decides whether to mark a node or un-label it according to its parameters. Marking a node several times has the same effect as marking it a single time. Un-labeling a node is done using a *cas* of its transaction descriptor to null. Since it won't be labeled with this descriptor again this *cas* will only succeed once and after that won't have any effect. Thus, for the same parameters to *unlabelOp* it will perform the same operation which leaves the data structure in an equal state if called several times. □

Claim 59. *The data structure does not change the transaction's status.*

Proof. The code for the *set* does not change the transaction status in any of its methods. □

Set Singletons We next prove that all singleton methods return valid results.

Theorem 60. *The add singleton method returns true* $\iff$ *no other node with* key *is logically in the list and it successfully added a new node with* key*.*

Proof. The *add* singleton method returns *true* in Line 150. This happens when the node n returned from *find* has a key satisfying $n.key \neq key$ and the *cas* in Line 149 succeeds. From Claim 44 the *find* method returns two items $pred$ and n s.t $pred.key < key$ and $n.key \geq key$. Since $n.key \neq key$ when the new node is added, we get that $n.key > key$. If the *cas* succeeds then $pred$'s next pointer is unmarked and from Claim 45 we learn that $pred$ is reachable. Furthermore, if the *cas* succeeds then no other node was inserted between $pred$ and $curr$, thus no other node with key is reachable (otherwise from Claim 46 it would have been inserted between $pred$ and n) and therefore no node with key is logically in the list. If the *cas* in Line 149 fails then $newNode$ was not added to the list and we retry (and not return).

The *add* singleton method returns *false* in Line 145. This happens when the node n returned from *find* has a key satisfying $n.key = key$, it is unlabeled and it is unmarked, thus it is logically in the list. Hence, there exists a node with key which is logically in the list and the *add* operation should fail. □

Theorem 61. *The remove singleton method returns true* $\iff$ *a node with* key *which is logically in the list is removed by this method.*

Proof. The *remove* singleton method returns *true* in Line 158. This happens when the node n returned from *find* has a key satisfying $n.key = key$ and the *labelNode* method returns n (Line 155). *labelNode* returns n when called from *remove* (with $shouldExist = true$ and transaction $removeT$) only when n is successfully labeled with $removeT$ (Line 131). If the labeling succeeds

then n's next pointer is unlabeled and unmarked (Claim 50) and from Claim 45 we learn that n is reachable (and logically in the list). If the *cas* in Line 131 fails, meaning n is labeled by a concurrent transaction, we retry the operation.

The *remove* singleton method returns *false* in Line 160. This happens when the node n returned from *find* has a key satisfying $n.key \neq key$. From Claim 44 the *find* method returns two items $pred$ and n s.t $pred.key < key$ and $n.key \geq key$. Since $n.key \neq key$, we get that $n.key > key$. Thus no node with key is logically in the list (or it would have been a successor of $pred$ and a predecessor of n). Hence, there does not exists a node with key which is logically in the list and the *remove* operation should fail. □

Theorem 62. *The get singleton method returns non-null $data$ $\iff$ a node n with key is logically in the list and $n.data = data$.*

Proof. The *get* singleton method returns non-null *data* in the following cases:

- Line 171: n was returned from *find* and it is unlabeled. From Claim 50 we learn that a marked node *marked* is labeled, so n is unmarked and from Claim 45 it is reachable and we can return its *data*.
- Line 179 n was returned from *find* and it is labeled by t with status `Pending`. We position the get operation's Linearization point prior to the transaction's, so we check whether n logically existed prior to t by checking the operation op performed by t and its result. n existed prior to t if op was an unsuccessful *add* operation or a successful *remove, contains* or *get* operation. If n logically existed then we return its $data$.
- Line 183: n was returned from *find* and it is labeled by $desc.t$ with status `Done`. We help complete t and then check whether its $next$ pointer is marked. If it is unmarked, then it is logically in the list and we can return its $data$.

The *get* singleton method returns null in Lines:

- Line 167: n was returned from *find*. From Claim 48 this means that no item with key logically exists in the list and so we can return null.
- Line 179: Similar to the non-null case: if $exist$ is false, this means that the transaction t added n and it is was not logically in the list before the transaction and so we return null.
- Line 183: Similar to the non-null case, if n is marked then it is not logically in the list and we return null.

□

Theorem 63. *The contains singleton method returns false $\iff$ a node n with key is logically in the list.*

Proof. The *contains* method returns false $\iff$ the *get* method called with the same *key* returns non-null $data$. From Claim 62 *get* returns non-null $data$ $\iff$ a node n with key is logically in the list with $data = n.data$. So, the Claim holds. □

Chapter 10

Conclusions

We proposed a framework for performing lock-free transactions on data structures. The transactions are linearizable, ensuring all operations appear to happen simultaneously, with no intermediate state visible to other transactions. Our framework consists of: (1) a transaction engine which receives a group of operation and performs them atomically, and (2) lock-free data structures extended to support the LOFT API. To exemplify our framework we implemented a LOFT set, a LOFT queue, and a LOFT register supporting the required API. The transaction engine employs a helping mechanism to obtain lock-freedom, and an advanced lock-free contention management for handling cyclic helping conflicts. The contention manager reorders the conflicting transactions execution allowing all transactions to complete with minimal delay.

We evaluated the performance of LOFT transactions against that of TDSL and a TL2 STM. We examined different contention and workload parameters. Measurements show that our proposed solution outperforms general software transactional memory (TL2), and in most scenarios it also outperforms lock-based transactional data structures (TDSL), while providing lock-free progress guarantee and a reduced number of aborts. While TDSL may abort the same transaction over ten times on average, LOFT transactions abort rate is close to zero, even when contention is high.

Finally, we provide a correctness proof for our LOFT algorithm, which defines the requirements from LOFT data structures, and proves that all operations inside a transaction are performed atomically. We believe this work provides a solution for performing lock-free transactions on data structures efficiently. Future work may provide additional LOFT data structures, and also formally define an appropriate programming interface.

Bibliography

[1] Guy Golan-Gueta Alexander Spiegelman and Idit Keidar. Transactional data structure libraries. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2016.

[2] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: High performance lock-free priority queue. In *Euro-Par 2016, the 22nd International Conference on Parallel and Distributed Computing*, 2016.

[3] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, September 2008.

[4] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM.

[5] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly methodology for search structures. 02 2012.

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] M. Sagiv G. Golan-Gueta, G. Ramalingam and E. Yahav. Concurrent libraries with foresight. In *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 263–274, 2013.

[8] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. *SIGPLAN Not.*, 50(8):1–10, January 2015.

[9] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC'01*, 2001.

[10] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

[11] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[12] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. 08 2010.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

[15] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 31:1–31:16, 2016.

[17] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 59–68, New York, NY, USA, 2009. ACM.

[18] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[19] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

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

[21] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[22] Deli Zhang and Damian Dechev. Lock-free transactions without rollbacks for linked data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 325–336, New York, NY, USA, 2016. ACM.

לקבל מספר פעולות ולבצע אותם כטרנזקציה אטומית. אנו משתמשים במנגנון עזרה כדי לתמוך בחוסר-נעילה, ובמנגנון ניהול תחרות כדי להקל על ההשפעות של ביטול טרנזקציות. כאשר מזוהים קונפליקטי עזרה מעגליים, מנגנון ניהול התחרות מסדר מחדש את ביצוע הטרנזקציות שנמצאות בקונפליקט בצורה שמאפשרת להשלים את כל הטרנזקציות עם עיכוב מינימלי. בכל מעגל כזה טרנזקציה אחת מבוטלת וביצועה נדחה כדי לפתור קונפליקט שהיא יצרה. קונפליקטי העזרה המעגליים קורים לעתים נדירות והם הסיבה היחידה שאנו מבטלים טרנזקציות ולכן מספר ביטולי הטרנזקציות נמוך.

ייתכן כי יהיו פעולות בתוך טרנזקציה שיהיו תלויות במידע של פעולות קודמות או בשטף התכנית עד אליהם. אנו מתארים כיצד תלויות אלו יכולות להיתמך ע"י התשתית שלנו ע"י שינוי אופן קבלת הפעולות לביצוע. אנו מגדירים מתי תלויות כאלה אינן קיימות ומתארים אופטימיזציה אשר משתמשת במידע זה כדי לשפר את הביצועים. כדי להדגים את התשתית שלנו מימשנו קבוצה טרנזקציונית באמצעות רשימת דילוגים, תור טרנזקציוני ורגיסטר טרנזקציוני. אנו מציגים את מדידות הביצועים של המערכת שלנו אשר מראים כי הביצועים שלנו טובים יותר מזיכרון טרנזקציוני תוכנתי כללי ותחרותיים מול מבני נתונים טרנזקציוניים המבוססים על נעילות. זאת בעיקר מפאת צמצום כמות ביטולי הטרנזקציות ושימוש באלגוריתם חסר נעילות שלרוב מתמודד עם תחרות בצורה טובה יותר. בנוסף, כאשר ניתן למיין טרנזקציות השיפור בביצועים גדול אף יותר.

תקציר

מבני נתונים מקביליים נמצאים בשימוש רווח בארכיטקטורות מרובות-ליבות. מבנים אלו מאפשרים ביצוע אטומי של כל פעולה עליהם, ללא חשש מחפיפה עם פעולות המבוצעות במקביל. עם זאת, ישנם מקרים בהם יש צורך לבצע מספר פעולות על מספר מבני נתונים בצורה אטומית כאשר מצב ביניים של הפעולות לא חשוף לחוטים אחרים. לדוגמא, העברת איבר מתור אחד לתור אחר כאשר שומרים על שמורה שהאיבר נמצא בכל רגע ברשימה אחת בדיוק. דרישה זו איננה ניתנת למימוש בצורה מובנה מאליה ע"י מבני נתונים מקביליים ודורשת שימוש במנגנון נעילות או שימוש בזיכרון טרנזקציוני.

שימוש במנגנון נעילות אינו מבטיח התקדמות מערכתית ובנוסף רגיש לקיפאון, היפוך עדיפויות ועוד. שימוש בזיכרון טרנזקציוני תוכנתי דואג לכך שכל פעולות הקריאה וכתיבה בתכנית קורות ברגע נתון בזמן (או בכלל לא) ומצבי ביניים אינם חשופים לתהליכונים אחרים. הפשטות הקונספטואלית של גישה זו היא מאוד ידידותית למתכנת ואינה דורשת התמודדות עם קיפאון. עם זאת, גישה זו טומנת בחובה סנכרון רב שפוגע רבות בביצועים, הן בגלל הצורך לזהות קונפליקטים והן בגלל הצורך לבצע טרנזקציות מחדש כאשר קונפליקטים צצים. קיים מחקר רב בנושא זיכרון טרנזקציוני תכנתי אשר מנסה לצמצם את הפגיעה בביצועים, לרוב על ידי הגבלה כלשהי על הפעולות הנתמכות.

פתרון שהוצע לאחרונה עבור ביצוע טרנזקציות המורכבות מפעולות על מבני נתונים מנסה להתמודד עם החסרונות לעיל. פתרון זה משתמש במנגנון הבנוי ממבני נתונים מקביליים קיימים אשר מורחבים לתמוך בטרנזקציות ומשתמשים בידע הספציפי על מימוש מבנים אלו כדי להגביל את כמות פעולות הקריאה והכתיבה שיש לסנכרן. כך ישנם פחות קונפליקטים בין טרנזקציות שונות ופחות צורך לבצע טרנזקציות מחדש, דבר המשפר את הביצועים בצורה משמעותית. עם זאת, כאשר ישנה תחרות רבה עדיין יש צורך לבטל טרנזקציות ולבצען מחדש. כמו כן, פתרון זה משתמש במנעולים ולא מבטיח התקדמות מערכתית.

בעבודה זו אנו מציגים עיצוב של תשתית טרנזקציונית אשר תומכת בטרנזקציות אטומיות שמורכבות ממספר פעולות על מבני נתונים באופן חסר-נעילות. העיצוב שלנו מורכב משני חלקים: (1) מבני נתונים מקביליים חסרי נעילות אשר תומכים בממשק טרנזקציוני ו-(2) מנוע טרנזקציות אשר אחראי

המחקר בוצע בהנחייתו של פרופ' ארז פטרנק, בפקולטה למדעי המחשב בטכניון.

אני מודה לטכניון על התמיכה הנדיבה במשך השתלמותי.

מבני נתונים טרנזקציוניים חסרי נעילות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי המחשב

אבנר אליזרוב

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

כסלו התשע״ט חיפה נובמבר 2018

מבני נתונים טרנזקציוניים חסרי נעילות

אבנר אליזרוב