Wait-free Dynamic Transactions for Linked Data Structures

Pierre LaBorde
University of Central Florida
Orlando, USA
pierrelaborde@knights.ucf.edu

Lance Lebanoff
University of Central Florida
Orlando, USA
lancelebanoff@knights.ucf.edu

Christina Peterson
University of Central Florida
Orlando, USA
clp8199@knights.ucf.edu

Deli Zhang
University of Central Florida
Orlando, USA
deli.zhang@outlook.com

Damian Dechev
University of Central Florida
Orlando, USA
dechev@cs.ucf.edu

Abstract

Transactional data structures support threads executing a sequence of operations atomically. Dynamic transactions allow operands to be generated on the fly and allows threads to execute code in between the operations of a transaction, in contrast to static transactions which need to know the operands in advance. A framework called Lock-free Transactional Transformation (LFTT) allows data structures to run high-performance transactions, but it only supports static transactions. We extend LFTT to add support for dynamic transactions and wait-free progress while retaining its speed. The thread-helping scheme of LFTT presents a unique challenge to dynamic transactions. We overcome this challenge by changing the input of LFTT from a list of operations to a function, forcing helping threads to always start at the beginning of the transaction, and allowing threads to skip completed operations through the use of a list of return values. We thoroughly evaluate the performance impact of support for dynamic transactions and wait-free progress and find that these features do not hurt the performance of LFTT for our test cases.

CCS Concepts • **Computing methodologies → Shared memory algorithms**; **Concurrent algorithms**.

Keywords Transactional Data Structures, Wait-Free, Transactional Memory, Non-blocking

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PMAM'19 , February 17, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309491

ACM Reference Format:
Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-free Dynamic Transactions for Linked Data Structures. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19), February 17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3303084.3309491

1 Introduction

The rise of multi-core systems has led to the development of highly concurrent non-blocking data structures [7, 19, 20, 29]. Traditionally, non-blocking data structures provide operations which meet the linearizability correctness condition. Linearizable operations appear to execute instantaneously, and respect the real-time ordering of operations. Lock-freedom and wait-freedom are two different kinds of non-blocking algorithms that guarantee at least one or all threads make progress in a finite amount of time, respectively. These algorithms are free from common pitfalls associated with locking such as deadlock, livelock, and priority inversion, by definition. Wait-free algorithms are also starvation-free, by definition.

A limitation of non-blocking containers is a lack of support for composite operations, which precludes modular design and software reuse. For example, inserting an element into a lock-free linked list, and incrementing a separate variable that stores the length of the linked list is not possible without breaking linearizability, as most non-blocking data structures can only guarantee atomic updates to a single memory word. The aforementioned composite operation could fail if two threads concurrently insert elements at non-adjacent positions in the linked list, concurrently read the size variable as ten, and then write the new value which they will both compute as eleven. The trade-off between correctness and support for composite operations in non-blocking data structures does not need to be made if the data structures are made transactional.

Implementing transactional containers has been the subject of several recent papers [2, 9, 10, 13, 14, 18]. Transactional execution is essential for applications that require

atomicity and isolation for a series of operations such as databases and data analysis applications. In this paper, we discuss data structure transactions, which are sequences of operations that are executed atomically, in isolation, on a shared memory data structure. Isolation means concurrent transaction executions appear to take effect in some sequential order.

The straightforward way to implement a transactional data structure from a sequential container is to use software transactional memory (STM) [26]. An STM instruments memory accesses by recording the locations a thread reads in a *read set*, and the locations it writes in a *write set*. If the read/write sets of different transactions overlap, only one transaction is allowed to commit, the concurrent transactions are aborted and restarted. A drawback of STM is that the runtime system that keeps track of read/write sets and detects conflicts can have a detrimental impact on performance [3].

The inherent disadvantage of STM concurrency control is that *low-level memory access conflicts do not necessarily correspond to high-level semantic conflicts*. Consider a set implemented as an ordered linked list, where each node has two fields, an integer value and a pointer to the next node. The initial state of the set is $\{0, 3, 6, 9, 10\}$. Thread 1 and Thread 2 intend to insert 4 and 1, respectively. Since these two operations commute, it is feasible to execute them concurrently [4]. Commutative data structure operations are those which have no dependencies on each other; reordering them yields the same abstract state of the container. Existing concurrent linked lists employing lock-free or fine-grained locking synchronizations allow concurrent execution of the two operations. Nevertheless, these operations have a read-/write conflict and the STM has to abort one of them.

An alternative approach called Lock-free Transactional Transformation (LFTT) [30] includes semantic conflict detection that uses information about the data structures and which operations are being executed, to prevent conflicting operations from unnecessarily causing transactions to abort. A significant advantage of using data structure transactions is that this semantic information is available to be used to increase throughput.

The main drawback of using LFTT is that it only supports *static transactions*—it requires all operations to declare their operands in advance, and a thread cannot execute any code between the operations of a transaction. LFTT needs a static list of operations and operands before the transaction begins, because it has threads help each other complete pending operations before starting new ones; this is how LFTT guarantees system-wide progress. This limitation restricts the applicability of LFTT to small applications whose inputs are known in advance. For example, the following code snippet could not be executed by LFTT. LFTT would need to transform this code into a list of operands and operations, but *result* is unknown, and operations cannot be executed conditionally.

```
if(!list.find(key)) {
result = ... // some computation
list.insert(result); }
```

In this paper, we present Dynamic Transactional Transformation (DTT), a framework that builds upon LFTT to increase its applicability; namely, to add support for (1) dynamic transactions, (2) wait-free progress, and (3) transactions among multiple data structures. First, *dynamic transactions* allow operands to be generated during the transaction, and threads can execute code between the operations of a transaction. In contrast to other transactional methodologies, LFTT presents a nontrivial challenge to support dynamic transactions due to its thread-helping scheme. To address this issue, we require that the user writes a function that encompasses all of the operations to be performed in the transaction as well as any code that the user wants to run between the operations. To maintain correctness in the presence of this code between operations, we modify the helping scheme so that helping threads always start at the beginning of the transaction. Then to reduce duplicate work, we maintain a list that contains the return value for each operation in the transaction, which allows helping threads to skip completed operations. Second, we support wait-free progress by employing the fast-path-slow-path technique [17]. Third, transactions among multiple data structures can be simply implemented by modifying a structure in LFTT representing an operation, to add a field that references the container on which to operate.

Like LFTT, our approach is applicable to the class of linked data structures that implement the set and dictionary abstract data types. We apply DTT to create dynamic transactional versions of a linked list [12], a skip list [8], an MDList [28], a dictionary [29], and a binary search tree [16]. Lock-free transactional versions of the linked list and skip list were presented as a proof of concept for LFTT in [30].

We analyze the impact of these new features on the performance of LFTT. Our evaluation shows that DTT performs on par with LFTT, while providing the benefits of dynamic transaction support, a stronger progress guarantee, and transactions among multiple containers. There is less than a 1% difference when averaged over all tested scenarios, and like LFTT, our approach is 3 times faster than STM.

This paper makes the following contributions:

- We present DTT, an extension to LFTT to provide support for dynamic data structure transactions, which allow code to be run between data structure operations.
- We add support for wait-freedom to DTT, so lock-free containers can be transformed into wait-free transactional versions.

- We analyze the performance impact of these features to LFTT and other transactional methodologies. Experimental results report that the throughput is on par with LFTT while improving its applicability.

2 Related Work

Significant research has been devoted to non-blocking linked data structures [12, 20, 22, 28]. Transactions on a data structure traditionally involve executing all shared memory accesses in coarse-grained *atomic sections*. High-level conflict detection approaches [2, 14, 18] avoid false conflicts due to low-level accesses.

2.1 Transactional Memory

Transactional memory, initially proposed as a set of hardware extensions by Herlihy and Moss [15]. HTM's cache-coherency based conflict detection causes spurious failures during page faults and context switches [6]. On Intel's Haswell microarchitecture, the performance of applications that frequently encounter data access conflicts will degrade to coarse-grained locking. This makes HTM undesirable for container implementations.

Since STM detects conflicts at the granularity of read and write accesses, excessive aborts due to frequent accesses substantially limit concurrency.

Spiegelman, et al. [27] propose an approach called Transactional Data Structure Libraries (TDSL) that collects a read-set and write-set for a transaction in a way similar to STM. While this approach eliminates rollbacks, any data structure that it is applied to cannot guarantee lock-freedom or wait-freedom due to the locks used for synchronization.

2.2 Transactional Boosting

Transactional boosting transforms non-blocking data structures into locking transactional data structures. Boosting fails to preserve the non-blocking property, because locks are used for transaction-level synchronization.

2.3 LFTT

Zhang et al. [30] present LFTT, a methodology for transforming high-performance lock-free linked data structures into lock-free transactional containers. The key advantage of using DTT over LFTT is that dynamic transactions allow the user to generate operands on the fly, which allows the developer to create non-trivial programs. Our approach also provides wait-free progress which is essential for applications that operate under strict deadlines, including hard real-time systems. Further, our approach allows the composition of operations on multiple data structures within a single transaction. These capabilities are desirable for large-scale database and data analysis applications.

3 Dynamic Transactional Transformation

In this section, we provide an overview of our approach, an example of how to use DTT, and the details of the implementation and extensions for wait-freedom and multi-container transactions. Our approach is built on Lock-free Transactional Transformation (LFTT) [30]. LFTT provides a framework that allows a developer to transform a non-blocking container into a lock-free transactional container. LFTT adds a new code path to the data structure that synchronizes transactions. In [1], the *cooperative technique* is presented, which is essential to LFTT's transactional synchronization. This technique is based on the observation that multiple threads can work together if they all "write down exactly what they are doing," in a *descriptor*. The descriptor contains the information necessary for other threads waiting on a transaction to help it finish before attempting to begin their own transactions. By ensuring all threads work together to finish pending operations before beginning new ones, system-wide progress is guaranteed, as specified by the definition of lock-freedom. Note, throughout the paper we refer to line number X of algorithm Y as $Y.X$.

LFTT eliminates the overhead of physical rollbacks by using logical rollbacks, which allow the effects of an aborted transaction to appear to be undone through an inverse interpretation of the status of a node. Semantic knowledge of the data structure is used to allow commutative operations to proceed concurrently in a lock-free manner. Conflicts for non-commutative method calls are identified through the node-based conflict detection. In order to reduce aborts due to conflicts, the thread that identifies a conflict will help complete the transaction associated with the node of interest.

3.1 Overview

Our goal is to design an algorithm that executes arbitrary side-effect free code within a transaction, while retaining the ability to undo any and all operations and code in between. We call code that is executed within a transaction *intra-transactional code*. We require intra-transactional code to be side-effect free so that conflicts can be avoided outside of data structure operations, and the entire transaction can be rolled back without our approach requiring semantic information about the code added to the dynamic transaction. In STM, all code within a transaction is delineated using annotations that mark the beginning and end of the transactional block. Since we already require users to treat their data structures as white boxes, we do not place additional burdens on the user that are inherent in annotation languages, such as additional compilation time to perform static analysis. We do not consider the use of a run-time system, as in STM, because we aim to produce performance that is comparable to LFTT. Instead we encapsulate calls to data structure operations of transactional containers, and intra-transactional

code, within a *transactional function*. A pointer to the transactional function is stored in the transaction descriptor, since threads need to access each other's transactional functions in order to help complete their transactions.

Now that we have added transactional functions to our descriptor, we need to add support for them to the rest of the algorithm. This means that we need to synchronize the additional code that exists between operations within a transaction. To synchronize this code, we must find a way to integrate our new transactional functions to the helping scheme. In LFTT, a helping thread is allowed to help a transaction starting from any of the transaction's operations. Since the transactional function may contain intra-transactional code that affects which operations are executed later in the transaction, we must always start transactions from the beginning, even if the helped thread has already performed some work on the transaction. This causes helping threads to perform duplicate work. To reduce the amount of work that is duplicated, we maintain a list of return values in the transaction descriptor. When a thread completes a data structure operation in a transaction, it stores the return value in the list. This allows helping threads to avoid duplicate work by checking the return values list before executing an operation, to possibly skip the operation and simply return the previously calculated return value.

Since we now support transactional functions, we also need a way to get data into and out of these functions. In LFTT, the user cannot specify variables other than the static list of operands for the data structure operations, and the user cannot obtain the return values of data structure operations. LFTT only returns true or false, to indicate the success of a specific data structure operation. These return values are meant for internal use so that transactions can abort if any operations failed. In DTT, the user creates an *input map*, which is a hash map containing variables that have been defined outside of the transactional function that the user wants to use inside the transaction. Any data structure could be used, but we choose a map because it allows the programmer to retrieve values by name, within the transactional function. We store this input map into the transaction descriptor so that helping threads can read these variables. Once we begin executing a transaction, we copy the input map into a *local map* so that a thread can keep track of the values of these variables throughout the execution of the transaction. We create the local map so that the variables can be modified without interference from helping threads. To allow the user to access these variables after the transaction has completed, we copy the final values of the variables from the local map into an *output map*, which is stored in the transaction descriptor.

3.2 Using DTT

We now explain how a developer uses DTT to perform dynamic transactions.

In our library, transactional functions are restricted to those in which all shared memory accesses occur through data structure calls, and all other instructions in the transaction must occur locally. A user of DTT begins with a block of code and wants it to be executed atomically. The user then transforms the block of code into two parts: a transactional function, and a library call.

The transformed code using our library is shown in Algorithm 1, including the corresponding library call and transactional function. First, the user creates an input map and populates it with the variables that are needed in the transaction (lines 1.2-1.3) Then the user calls the ExecuteTxn method to run the transactional function (line 1.5). After the transaction completes execution, the user can access variables from the output map (line 1.6). In the transactional function, data structure calls are replaced with invocations of the CallOp method, so that the library can handle these operations behind the scenes (lines 1.9, 1.11, and 1.13). Accesses to variables that were added to the *input map* are handled by accessing the local hash map (lines 1.8 and 1.10).

ALGORITHM 1: Example of Transformed Code

```
1  Function Main()
2      HashMap* inputMap ← new HashMap()
3      inputMap.Put("x", 3)
4      HashMap* outputMap ← Null
5      ExecuteTxn(TxFunction, inputMap, outputMap)
6      Print(outputMap.Get("val"))
7  Function TxFunction(Desc* desc, HashMap*
       localMap)
8      int x ← localMap.Get("x")
9      T val ← CallOp(desc, skiplist, Find, x)
10     localMap.Put("val", val)
11     bool success ←
         CallOp(desc, skiplist, Insert, 4, val)
12     if success = true then
13         CallOp(desc, skiplist, Delete, 5)
14     return success
```

3.3 Implementation Details

We now explain the details that allow DTT to execute the user's transactional function.

We list the data type definitions for LFTT in Algorithm 2. LFTT adds a new field to the nodes stored by the base lock-free data structure, called *info*, as seen in Node. NodeInfo stores *desc*, a reference to the shared transaction descriptor, and an index *opid*, which provides a record of the last access. The LFTT transaction descriptor, Desc, contains three variables. LFTT keeps track of the status of a transaction in *status*. The type of operation that is being executed

and its operands are kept in an array called *ops*, and its length, *size*, is also stored. Given a node *n*, we can identify the most recent operation that accessed the node as *n.info.desc.ops[n.desc.opid]*. A node is considered *active* when the last transaction that accessed the node had an active status, this is expressed as *n.info.desc.status* = Active. Our transaction descriptor stores all of the necessary context for helping finish a delayed transaction, and it shares the transaction status among all nodes participating in the same transaction.

ALGORITHM 2: Type Definitions

```
1  enum TxStatus
2      Active
3      Commited
4      Aborted
5  enum OpType
6      Insert
7      Delete
8      Find
9  struct
   Operation
10     OpType
         type
11     int key
12 struct Desc
13     int size
14     TxStatus
         status
15     Operation
         ops[ ]
16     int
         returnValues[ ]
17 struct NodeInfo
18     Desc* desc
19     int opid
20 struct Node
21     NodeInfo*
         info
22     int key
23     ...
```

The ExecuteTxn function, shown in Algorithm 3, is a wrapper function. We modify the corresponding method from LFTT by storing the transactional function, input map, and output map into the transaction descriptor (lines 3.4). Then we call the HelpTransaction method.

The HelpTransaction function is the entry point for transactional execution. Since threads in DTT can recursively help multiple transactions, we maintain a thread-local *help stack*. The procedure in lines 3.9-3.10 is inherited from LFTT to prevent the livelock situation described in Section 3.3.Then we copy the data contained in the input map into the local map (line 3.11). Copying to a local hash map allows threads to modify and maintain local values of variables in the transactional function without interfering with the corresponding variables in other threads. Then we invoke the transactional function (line 3.12). The transactional function contains data structure operations encapsulated in CallOp library method calls, along with intra-transactional code. An example transactional function is shown in Algorithm 1. The use of a transactional function in this way contrasts with LFTT, in which the thread would execute the transaction based on a simple list of Operation objects, which would not support dynamic code paths. The transactional function's return value indicates whether or not it successfully executed all of its operations. If so, we perform a CompareAndSwap operation to change the transaction descriptor's status to Committed; otherwise we change the descriptor's status to Aborted (lines 3.14-lines 3.14). After the transaction has completed (whether by committing or aborting), we copy the data from the local map into the output map if no other thread has done so yet (line 3.18). This allows the user to extract values of local variables from the output map after the transaction has executed.

ALGORITHM 3: Transaction Execution

```
1  thread_local Stack helpstack
2  Function ExecuteTxn(Function* func, HashMap*
   inMap, HashMap* outMap)
3      helpstack.Init()
4      Desc* desc ← new Desc(func, inMap, outMap)
5      HelpTransaction(desc)
6      return desc.status = Committed
7
8  Function HelpTransaction(Desc* desc)
9      // Search helpstack to prevent livelock
10     helpstack.Push(desc)
11     HashMap* localMap ← Copy(desc.inputMap)
12     bool ret ← desc.Func(desc, localMap)
13     helpstack.Pop()
14     if ret = true then
15         CAS(&desc.flag, Active, Committed)
16     else
17         CAS(&desc.flag, Active, Aborted)
18     desc.outputMap ← Copy(localMap)
```

ALGORITHM 4: Call Operation

```
1  Function CallOp(Desc* desc, Container c, OpType
   type, args...)
2      if desc.status = Aborted then return Null
3      int opid ← helpstack.GetOpid()
4      if desc.returnValues[opid] exists then
5          return desc.returnValues[opid]
6      NodeInfo* info ← new NodeInfo(desc, opid)
7      desc.ops[opid] ← new Operation(args)
8      if type = Find then
9          int ret ← c.Find(desc, info, opid, args)
10     else if type = Insert then
11         int ret ← c.Insert(desc, info, opid, args)
12     else if type = Delete then
13         int ret ← c.Delete(desc, info, opid, args)
14     desc.returnValues[opid] ← ret
15     helpstack.NextOp()
16     return ret
```

The CallOp method, shown in Algorithm 4, calls a data structure operation. Before performing the operation, we check if the transaction has already been aborted, and if so, it can be skipped (line 4.2). In DTT, the help stack is implemented such that it keeps track of the transactions that the thread is currently helping, as well as the index of the current operation within each transaction. In the first step of CallOp, we obtain the index of the current operation from the help stack (line 4.3). In the next step, we handle the problem of duplicate work, which is unique to DTT. LFTT avoids the problem of duplicate work by allowing a helper thread to start the transaction from any operation, including an operation in the middle of the transaction. However, DTT cannot employ this technique because helper threads not only need to execute the data structure operations, but they also need to execute the local intra-transactional code as well. Therefore, helper threads must always start at the beginning of the transaction, which causes them to perform unnecessary work. To address this problem, we store return values of completed operations in a *return values list*. At the beginning of the CallOp method, we check to see if the return values list contains an entry for the current operation (line 4.4). If so, that means that another thread has already performed this operation, so the current thread avoids duplicate work by simply returning the value from the return values list corresponding to the current operation (line 4.5). Otherwise, the thread performs the operation. As in LFTT, we create a NodeInfo object, which will be placed into the *info* field of the node is being accessed. Then we create a new Operation object and place it into the transaction descriptor's *ops* list (line 4.7). This contrasts with LFTT in that LFTT requires the user to input a list of pre-defined Operation objects at the start of the transaction. Instead, DTT requires the user to input a transactional function, and each CallOp method builds the list of operations dynamically over the course of the transaction. Then, we call the specific data structure operation specified from the transactional function. After performing the operation, the thread stores the return value into the return values list (line 4.14). Then, the thread's help stack is updated to increment the index of the current operation within the current transaction (line 4.15).

3.4 Transactions Among Multiple Data Structures

We add support for transactions among multiple data structures simply by modifying the information stored per operation in the transaction descriptor, to add a field that stores a reference to the container on which the data structure operation should be performed.

3.5 Wait-free Transactions

To guarantee wait-freedom, we modify the transactional code path that LFTT adds to the base data structure, by implementing the fast-path-slow-path approach [17]. As such, we limit the number of retries for any data structure operation to a user-defined constant which can be tuned to trade-off performance versus fairness. When the limit is reached the thread places their transaction descriptor in a global table, called the announcement table, which other threads periodically check. If a thread finds a transaction descriptor in the announcement table, then the thread helps execute the other transaction's operations regardless of whether or not that transaction's operations conflict with its own. Using the announcement table in conjunction with limiting the number of retries yields a wait-free approach [17].

4 Correctness

We have developed a proof of correctness similar to the one in [30] to prove that any data structure transformed by DTT guarantees the strict serializability correctness condition. However, we omit the proof for lack of space. We plan to distribute the full proof in an extended journal version of this paper.

We also check the correctness of the transactional data structures featured in this paper using the tool TxC-ADT [24]. The correctness conditions evaluated include causal consistency [25], serializability [23], strict serializability [23], and opacity [11]. All of the tested transactional data structures satisfy the evaluated correctness conditions.

5 Performance Evaluation

We compare the containers in DTT with transactional boosting, TDSL, STM, and LFTT versions. We also perform experimental evaluations to analyze the effect of the wait-free progress assurance scheme on the performance of DTT.

We perform our STM comparison using NOrec STM [5] from the Rochester STM package [21]. We make an exception for the skip list, as Fraser provides an implementation that uses his own object-based STM implementation [8]. For the TDSL approach, we apply their methodology to transform a lock-free linked list into a transactional linked list and compare to our transactional linked list. We also compare against LFTT. LFTT does not include a wait-free progress assurance scheme, a way to perform multi-container transactions, or support for dynamic transactions. We show this comparison to demonstrate the low performance overhead of the progress assurance scheme and dynamic transaction support.

The transactional boosting, TDSL, and STM algorithms all easily support dynamic transactions, so we do not need to modify them for our performance evaluation. LFTT does not support dynamic transactions, so we hard-code 500 cycles of busy work between each data structure operation for the purpose of our tests. Because each alternative approach performs memory management differently, we statically allocate all nodes at the beginning of the evaluation and disable node reclamation for a fair comparison of each approach's conflict management scheme.

5.1 Experimental Setup

We use a micro-benchmark to evaluate performance across three different operation distributions: read-dominated, mixed, and write-dominated. In this canonical evaluation method [5, 12], each thread repeatedly performs transactions with randomly chosen mixtures of Insert, Delete and Find operations. This loop continues to execute transactions for 10 seconds. The transaction size (i.e., the number of operations in a transaction) is chosen randomly for each transaction in the test up to a maximum size of 7 operations, as in [27]. We test on a 64-core NUMA system (4 AMD Opteron 6272 CPUs with 16 cores per chip @ 2.1 GHz). All code is compiled with GCC 4.8 with C++17 features and 03 optimizations. [1]

In this section, we graph the throughput and number of spurious aborts for each container. The throughput is measured in committed transactions per second. The number of spurious aborts takes into account the number of aborted transactions except self-aborted ones (i.e., those that abort due to failed operations). We include the number of spurious aborts as an indicator of the effectiveness of the contention management strategy. The three operation distributions are 15% Insert, 5% Delete, 80% Find (read-dominated); 33% Insert, 33% Delete, 34% Find (mixed); and 50% Insert, 50% Delete, 0% Find (write-dominated). To save space, we only display the graphs for the read-dominated and mixed scenarios, as they are the closest to real-world operation distributions [19]. The graphs for the write-dominated scenario are very similar to the other distributions, and we present the average results of the three distributions. Each wait-free transactional data structure is run with *HELP_DELAY* set to 10 and *MAX_FAILURES* set to 5. We denote LFTT as LFT, and transactional boosting as BST, in the graphs. The upper portion of the figures represents the throughput with the x-axis in logarithmic scale and the y-axis in linear scale. The key for all of the performance graphs is the same, and can be found in Figure 1. The bottom half of all figures represents the histogram of spurious aborts, with the x- and y-axes in logarithmic scale; the key is shown on the right half of Figure 1.

5.2 Overall Results

Across all data structure evaluations, DTT outperforms BST by an average of 108%, TDSL by an average of 247%, STM by an average of 196%, and LFT by an average of 26.9%. DTT gains an advantage over BST, TDSL, and STM because of its semantic conflict detection and logical interpretation, which allows it to avoid the costs of excessive aborts and physical rollbacks. DTT outperforms LFTT because it allows threads to avoid the cost of duplicate work by utilizing a return values list. Also, because the *MAX_FAILURES* parameter of the progress assurance scheme is set to 5, this means that a thread will wait until it has retried an operation five times before posting an announcement, which is rarely observed in practice [19]. As a result, threads rarely need to pause their own operations to help other threads. Therefore, our library provides dynamic transaction execution, wait-free progress, and multi-container transactions at no additional cost.

Figure 1. Key for Performance Graphs

5.3 Transactional List

We compare the throughput of five different implementations of transactional linked lists in Figure 2. The base data structure used by all of the implementations is the lock-free list by Harris [12]. Each thread in the transactional list performs transactions for 10 seconds with a key range of 10, 000.

In overall throughput, DTT outperforms BST by an average of 170%, TDSL by an average of 247%, and STM by an average of 461% across all operation distributions. The superior performance of DTT (as well as LFT) can be attributed to its logical status interpretation and cooperative contention management. When BST, TDSL, and STM encounter a conflict, they abort one of the conflicting transactions, decreasing the overall throughput. On the other hand, DTT and LFT avoid most of these spurious aborts because threads help each other to complete each other's transactions, allowing both transactions to commit. This can be observed in the number of spurious aborts shown in the bottom half of each graph in Figure 2. For example, in the case of the number of spurious aborts with 64 threads, BST experiences 3 times more than DTT and LFT, TDSL experiences three orders of magnitude more, and STM experiences four orders of magnitude more.

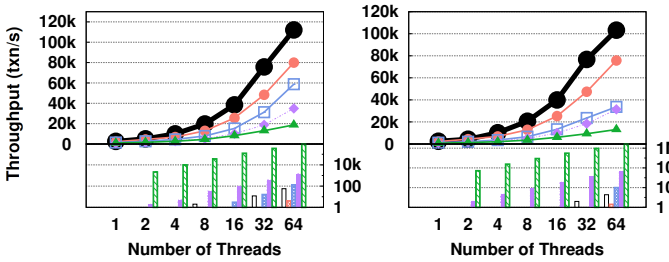
STM's throughput particularly suffers with more threads, due to the excessive aborts in response to memory access conflicts. In a linked list, all operations traverse the nodes at the beginning of the list, resulting in a high chance of memory access conflicts and aborts.

DTT outperforms LFT by 39.2% while also providing the benefits of dynamic transactions and wait-free progress. The overhead of DTT is low because it rarely needs to activate its wait-free progress assurance scheme. For each operation, the performance cost of traversing the linked list far outweighs the cost of the progress assurance scheme. In addition, by using a list of return values, DTT allows helper threads to avoid duplicate work.

5.4 Transactional Skip List

We compare the throughput of four different types of transactional skip lists in Figure 3. The implementations are based

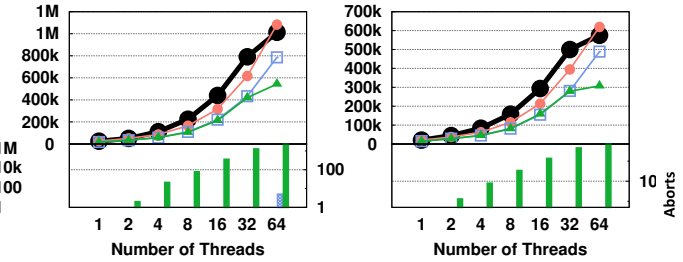
[1] All source code will be made available upon publication.



(a) 15% Insert, 5% Delete, 80% Find

(b) 33% Insert, 33% Delete, 34% Find

Figure 2. Transactional List Performance



(a) 15% Insert, 5% Delete, 80% Find

(b) 33% Insert, 33% Delete, 34% Find

Figure 3. Transactional Skip List Performance

on the skip list presented by Fraser [8]. Because skip lists have logarithmic search time, we increase the workload such that the skip list has a key range of 1,000,000.

The skip lists execute transactions much more efficiently than the linked lists, with a maximum throughput of 1,000,000 transactions per second (versus 80, 000 per second for the linked lists). Also, because of the increase in key range, concurrent transactions for LFT, DTT, and BST are less likely to encounter node-level conflicts. Because skip lists traverse through fewer nodes, concurrent STM transactions are also less likely to encounter conflicts. As a result, all implementations of the transactional skip list experience no more than 4% of the spurious aborts that the corresponding linked lists experience, with DTT and LFT experiencing no spurious aborts at all.

In overall throughput, DTT outperforms BST by an average of 68.7% and STM by an average of 77.8%, while performing 20.2% faster than LFT. As with the transactional linked list, the DTT version of the skiplist experiences low overhead compared to LFT.

5.5 Transactional MDList

Figure 4 shows the throughput and spurious aborts for the four types of transactional MDLists. The base data structure for the transactional MDList of all implementations is the lock-free MDList by Zhang et al. [28]. Like the skip list, the MDList has logarithmic search time, so we perform the evaluation with a key range of 1, 000, 000.

The results are similar to the transactional skip list in Section 5.4. In overall throughput, on average, DTT outperforms BST by 110%, STM by 149%, and LFT by 24.5%.

Recall that the throughput of the STM skip list increases with more threads. Conversely, the STM MDList's throughput increases until 16 threads and then decreases significantly. This difference is due to these factors: the MDList's insert method, STM's memory barriers, and inter-processor communication between remote cores in the NUMA system. Each node in an MDList has several child nodes. When an MDList inserts a node, some cases require the new node to "adopt" its successor node's children. The new node is associated with an *adoption descriptor* object. When another thread traverses to the new node, it must check the new node's adoption descriptor to see if it must help in the child adoption process. This greatly increases the number of shared memory locations read during traversal. For each of these reads, STM uses a memory barrier. To adhere to the memory barriers, concurrently executing cores must send messages according to the machine's cache coherence protocol. On the NUMA machine, inter-processor communication between cores on separate chips is expensive and slows the MDList traversal.

5.6 Transactional Dictionary

The graphs of the performance of the transactional dictionaries are omitted, because they are the same as those for the transactional MDLists in Section 5.5. The dictionary has the same memory layout and similar underlying code as the transactional MDList, with the addition of a value parameter attached to the insert and find operations.

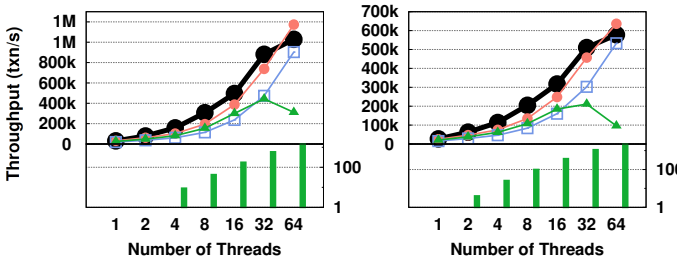
5.7 Transactional Binary Search Tree

Figure 5 shows the performance results of the four types of transactional binary search trees. The DTT, LFT, and BST implementations are based on the non-blocking binary search tree proposed by Howley [16]. Because the binary search tree provides logarithmic search time, we perform the evaluation with a key range of 1, 000, 000.

The performance results of the transactional binary search trees resemble those of the transactional MDLists in Section 5.5. In overall throughput, DTT outperforms BST by an average of 124% and STM by an average of 173%, and LFT by an average of 26.6%.

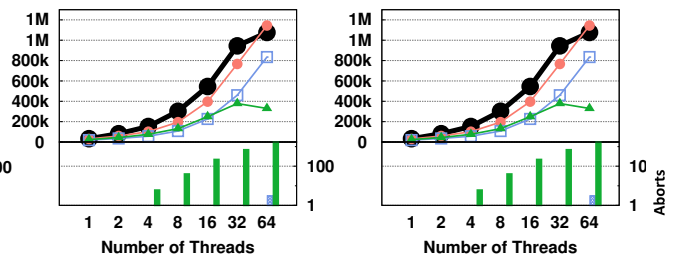
5.8 Wait-free Transactions

We perform experimental evaluations to study the effect of the wait-free progress assurance scheme on the performance of DTT. We observe the throughput and number of spurious aborts with the progress assurance scheme enabled,



(a) 15% Insert, 5% Delete, 80% Find **(b)** 33% Insert, 33% Delete, 34% Find

Figure 4. Transactional MDList Performance



(a) 15% Insert, 5% Delete, 80% Find **(b)** 33% Insert, 33% Delete, 34% Find

Figure 5. Transactional Binary Search Tree Performance

compared to when the scheme is disabled. When enabled, the data structure is run with *HELP_DELAY* set to 1 and *MAX_FAILURES* set to 1. These parameter settings are at the highest level in that they cause the progress assurance scheme to be invoked the most frequently possible. We set the parameters in this way to clearly observe the effects of the scheme in the most extreme case. We denote the approach with the wait-free progress assurance scheme enabled as WF (shown in the figure as black), and disabled as LF for the remainder of this section (shown in the figure as orange). In our test cases, we vary the number of threads between 1 and 64, and we vary the key range between 10 and 1, 000, 000. We only present the results for the transactional binary search tree, as they are representative of the other data structure results.

Overall, the results indicate that the progress assurance scheme has an insignificant impact on the performance of the transactional data structure, while offering the guarantee of wait-free progress. Across all of our test cases, the average throughput of WF is only 0.88% less than that of LF. For the extreme test case with a key range of 10, WF falls behind LF by 5.5%, due to an increase in the number of spurious aborts and other factors which we discuss in this section.

Figure 6 shows the performance results of the two approaches across varying key ranges. For each key range, the figure displays the average throughput (commits per second) and number of spurious aborts for all of the test cases with different numbers of threads. The trend we observe is that the progress assurance scheme has a lower impact on the performance of the data structure as the key range is increased. For a key range of 1, 000, 000, the progress assurance scheme has an insignificant effect on the throughput, with WF outperforming LF by 0.582%. For a key range of 10, the progress assurance scheme slightly reduces the throughput; WF falls behind LF by 5.76%. This trend can be explained by the difference in contention levels for each key range, which affects the frequency at which the progress assurance scheme is activated. A lower key range increases contention levels, which causes the progress assurance scheme to be invoked more often.

How does the progress assurance scheme diminish the throughput? There are three ways to explain this. (1) Posting to and reading from the announcement table incurs an overhead to the system. (2) Having threads help each other on the same transactions reduces parallelism. (3) Helper threads are delayed, resulting in more conflicts and therefore more spurious aborts. We observe this phenomenon in the data, as WF induces 7.97 times as many spurious aborts as LF. To explain this, we must first describe a type of abort that we refer to as *abort-on-helper*. Say a thread t_1 begins a transaction T_1 and then helps another transaction T_2 through the progress assurance scheme. An abort-on-helper occurs in the case that another thread t_3 running transaction T_3 finds that T_3 conflicts with T_1, so it aborts T_3. We find that for the test cases with a key range of 10, aborts-on-helper account for 67.7% of all spurious aborts. These results suggest that aborts-on-helper play a role in the difference in fake aborts between WF and LF. We believe that aborts-on-helper occur so frequently because when the helper thread t_1 helps another thread, its own transaction T_1 takes more time to complete and therefore increases the likelihood that another transaction T_3 will conflict with it, causing a spurious abort.

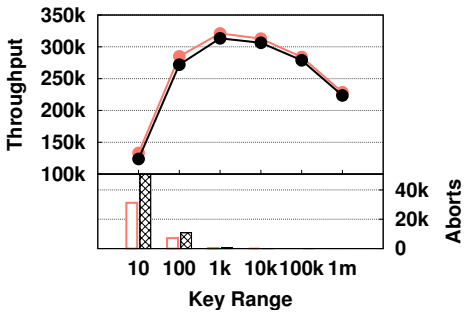


Figure 6. Wait-free Progress Assurance Scheme Overhead

6 Conclusion

While LFTT outperforms the transactional boosting, TDSL, and STM approaches, its lack of support for dynamic transactions makes it less applicable to general applications. With DTT, we remove this disadvantage while maintaining the competitive speed of LFTT, and add support for wait-free transactions on multiple data structures.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1717515 and Grant No. 1740095. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Greg Barnes. 1993. A Method for Implementing Lock-free Shared-data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/165231.165265

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

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

[4] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)* 32, 4 (2015), 10.

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

[6] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Early experience with a commercial hardware transactional memory implementation. (2009).

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

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

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

[10] Vincent Gramoli, Rachid Guerraoui, and Mihai Letia. 2013. Composing relaxed transactions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 1171–1182.

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

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

[13] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. On developing optimistic transactional lazy set. In *Principles of Distributed Systems*. Springer, 437–452.

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

[15] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[16] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. *Spaa* (2012), 161. https://doi.org/10.1145/2312005.2312036

[17] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2145816.2145835

[18] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. 2010. Coarse-grained transactions. *ACM Sigplan Notices* 45, 1 (2010), 19–30.

[19] Pierre LaBorde, Steven Feldman, and Damian Dechev. 2015. A Wait-Free Hash Map. *International Journal of Parallel Programming* (2015), 1–28. https://doi.org/10.1007/s10766-015-0376-3

[20] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Principles of Distributed Systems*. Springer, 206–220.

[21] Virendra J Marathe, Michael F Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N Scherer III, and Michael L Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*.

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

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

[24] Christina Peterson and Damian Dechev. 2017. A Transactional Correctness Tool for Abstract Data Types. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017), 37.

[25] Michel Raynal, Gérard Thia-Kime, and Mustaque Ahamad. 1997. From serializable to causal transactions for collaborative applications. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*. IEEE, 314–321.

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

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

[28] D. Zhang and D. Dechev. 2015. A Lock-free Priority Queue Design Based on Multi-dimensional Linked Lists. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2015), 1–1. https://doi.org/10.1109/TPDS.2015.2419651

[29] D. Zhang and D. Dechev. 2016. An Efficient Lock-Free Logarithmic Search Data Structure Based on Multi-dimensional List. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 281–292. https://doi.org/10.1109/ICDCS.2016.19

[30] Deli Zhang and Damian Dechev. 2016. Lock-free Transactions Without Rollbacks for Linked Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, New York, NY, USA, 325–336. https://doi.org/10.1145/2935764.2935780