2018

Practical Dynamic Transactional Data Structures

Pierre LaBorde
University of Central Florida

PRACTICAL DYNAMIC TRANSACTIONAL DATA STRUCTURES

by

PIERRE LABORDE
M.S. University of Central Florida, 2013
B.S. University of Central Florida, 2011

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, FL

Summer Term
2018

Major Professor: Damian Dechev

ABSTRACT

Multicore programming presents the challenge of synchronizing multiple threads. Traditionally, mutual exclusion locks are used to limit access to a shared resource to a single thread at a time. Whether this lock is applied to an entire data structure, or only a single element, the pitfalls of lock-based programming persist. Deadlock, livelock, starvation, and priority inversion are some of the hazards of lock-based programming that can be avoided by using non-blocking techniques.

Non-blocking data structures allow scalable and thread-safe access to shared data by guaranteeing, at least, system-wide progress. In this work, we present the first wait-free hash map which allows a large number of threads to concurrently insert, get, and remove information. Wait-freedom means that all threads make progress in a finite amount of time — an attribute that can be critical in real-time environments. We only use atomic operations that are provided by the hardware; therefore, our hash map can be utilized by a variety of data-intensive applications including those within the domains of embedded systems and supercomputers.

The challenges of providing this guarantee make the design and implementation of wait-free objects difficult. As such, there are few wait-free data structures described in the literature; in particular, there are no wait-free hash maps. It often becomes necessary to sacrifice performance in order to achieve wait-freedom. However, our experimental evaluation shows that our hash map design is, on average, 7 times faster than a traditional blocking design. Our solution outperforms the best available alternative non-blocking designs in a large majority of cases, typically by a factor of 15 or higher.

The main drawback of non-blocking data structures is that only one linearizable operation can be executed by each thread, at any one time. To overcome this limitation we present a framework for developing dynamic transactional data containers. Transactional containers are those that execute

a sequence of operations atomically and in such a way that concurrent transactions appear to take effect in some sequential order. We take an existing algorithm that transforms non-blocking sets into static transactional versions (LFTT), and we modify it to support maps. We implement a non-blocking transactional hash map using this new approach. We continue to build on LFTT by implementing a lock-free vector using a methodology to allow LFTT to be compatible with non-linked data structures.

A static transaction requires all operands and operations to be specified at compile-time, and no code may be executed between transactions. These limitations render static transactions impractical for most use cases. We modify LFTT to support dynamic transactions, and we enhance it with additional features.

Dynamic transactions allow operands to be specified at runtime rather than compile-time, and threads can execute code between the data structure operations of a transaction. We build a framework for transforming non-blocking containers into dynamic transactional data structures, called Dynamic Transactional Transformation (DTT), and provide a library of novel transactional containers. Our library provides the wait-free progress guarantee and supports transactions among multiple data structures, whereas previous work on data structure transactions has been limited to operating on a single container. Our approach is 3 times faster than software transactional memory, and its performance matches its lock-free transactional counterpart.

To my parents, Bobby and Sandy, for their loving support and forbearance.

ACKNOWLEDGMENTS

I thank my committee members Dr. Gary Leavens, Dr. Damla Turgut, and Dr. Eduardo Mucciolo for their time and effort.

I am thankful to all of the current and former lab members for fruitful discussions and enlightening presentations, some of their names follow in alphabetical order: Andrew Tyler Barrington, Victor Cook, Amruth Dakshinamurthy, Dr. Steven Feldman, Daniel Gabriele, Ramin Izadpanah, Kenneth Lamar, Lance Lebanoff, Carlos Valera-Leon, Brendan Lynch, Zachary Painter, Christina Peterson, Dr. Deli Zhang, et al. In particular, I would like to thank Steven and Lance for being excellent collaborators, and Amruth and Steven for being with me from the beginning. I would also like to thank Steven, Brendan, and Amruth for the adventures we had traveling to conferences. I thank Christopher Giles, Dr. Jonathan Cazalas, and Dr. Sean Szumlanski for good conversations, including some good advice.

I thank all of my friends and family for supporting me throughout this process. I would not have been able to do any of this without the sacrifices that my parents made, and for that I am eternally grateful. I am particularly grateful to my siblings Swain Tiwari and Petal LaBorde, for listening when I needed to talk.

Most of all, I thank my advisor and mentor Dr. Damian Dechev, for providing me with the freedom to explore several different research areas, and for believing in me, even when I could not. Throughout my thesis work, Dr. Dechev has provided guidance that led me to become not just a better researcher, but a better person. I cannot adequately express how thankful I am.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

In this dissertation, we present dynamic transactional data structures that provide the wait-free progress guarantee. First, we present a wait-free hash map that is not transactional, as an introduction to non-blocking programming techniques, and to demonstrate the drawbacks of non-transactional data structures. Then, we discuss an extension of a lock-free transactional transformation methodology (LFTT), that has been applied to the wait-free hash map. Finally, we design and implement a new approach to transactional transformation that allows dynamic wait-free transactions to be executed on multiple containers within a single transaction. Our experimental results demonstrate that our performance is at least on par with state of the art approaches, and in all but one case surpasses them.

Our design is motivated by the need for applications and algorithms to change and adapt as modern architectures evolve. These adaptations have become increasingly difficult for developers as they are required to effectively manage an ever-growing variety of resources such as a high degree of parallelism, single-chip multi-processors, and the deep hierarchies of shared and distributed memories. Developers writing concurrent code face challenges not known in sequential programming, most importantly, the correct manipulation of shared data.

Currently, the most common synchronization technique is the use of mutual exclusion locks. Blocking synchronization can seriously affect the performance of an application by diminishing its parallelism [30]. The behavior of mutual exclusion locks can sometimes be optimized by using a fine-grained locking scheme [34], [51] or context-switching. However, the interdependence of processes implied by the use of locks, even efficient locks, introduces the dangers of deadlock, livelock, starvation, and priority inversion — our design avoids these drawbacks.

The rise of multi-core systems has led to the development of highly concurrent non-blocking data

structures [40, 43, 11, 58]. 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.

We deliver a hash map that provides both *safety* and *high performance* for multi-processor applications.

The hardest problem encountered while developing a parallel hash map is how to perform a global resize, the process of redistributing the elements in a hash map that occurs when adding new buckets. The negative impact of blocking synchronization is multiplied during a global resize, because all threads will be forced to wait on the thread that is performing the involved process of resizing the hash map and redistributing the elements. Our wait-free implementation avoids global resizes through new array allocation. By allowing concurrent expansion this structure is free from the overhead of an explicit resize, which facilitates concurrent operations.

The presented design includes dynamic hashing, the use of sub-arrays within the hash map data structure [42]; which, in combination with perfect hashing, means that each element has a unique final, as well as current, position. It is important to note that the perfect hash function required by our hash map is trivial to realize as any hash function that permutes the bits of the key is suitable. This is possible because of our approach to the hash function; we require that it produces hash values that are equal in size to that of the key. We know that if we expand the hash map a fixed number of times there can be no collision as duplicate keys are not provided for in the standard semantics of a hash map. The aforementioned properties are used to achieve the following design

goals:

(a) *Wait-free*: a progress guarantee, provided by our data structure, that requires all threads to complete their operations in a finite number of steps [30].

(b) *Linearizable*: a correctness property that requires seemingly instantaneous execution of every method call; the point in time that this appears to occur is called a linearization point, which implies that the real-time ordering of calls is retained [30].

(c) *High performance*: our wait-free hash map design outperforms, by a factor of 15 or more, state of the art non-blocking designs. Our design performs a factor of 7 or greater faster than a standard blocking approach.

(d) *Safety*: our design goals help us achieve a high degree of safety; our design avoids the hazards of lock-based designs.

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 [17, 2, 26, 19, 18, 23, 39]. 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 on a concurrent shared memory data structure. We require a transactional data structure to execute transactions atomically, and in isolation. In this context, 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) [53, 28]. 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 while the other 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*. For example, two *insert* operations executed on a linked list would conflict even if they were different keys that were not in the list. There would be a low-level conflict on the *head* node. 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) [61] 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. The greatest advantage of using data structure transactions is that this semantic information is

available to be used to increase throughput. The biggest drawback of using LFTT is that it only supports *static transactions*—it requires all operations to declare their operands at compile-time, and a thread cannot execute any code between the operations of a transaction. LFTT needs a static list of operations and operands at compile-time, 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 at compile-time. 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); }
```

We present Dynamic Transactional Transformation (DTT), a framework for transforming non-blocking data structures into containers that support *dynamic transactions*, which allow operands to be generated at runtime rather than compile-time, and threads can execute code between the operations of a transaction. Our approach is applicable to the large class of linked data structures that implement the set and dictionary abstract data types. We apply DTT to create a library of data structures from five existing lock-free data structures. After transformation we obtain dynamic transactional versions of a linked list [20], a skip list [14], an MDList [60], a dictionary [59], and a binary search tree [33]. Lock-free transactional versions of the linked list and skip list were presented as a proof of concept in [61].

Our library leverages traditional non-blocking data structure designs so that developers can write transactional programs without knowledge of the underlying algorithms for wait-free progress or transaction synchronization. The library is linearizable, because the linearizability correctness con-

dition is composable, and each of the containers is linearizable. In addition to providing multiple transactional data structures, our library enables the composition of transactional data structures within a single transaction.

Using the software design patterns (which we call templates) in Section 30, developers can implement their own dynamic transactional containers and extend the library with them. Our evaluation shows that our approach performs on par with its static transactional counterpart, while providing the benefits of dynamic transaction support, a stronger progress guarantee, and transactions among multiple containers. There is less than a one percent difference when averaged over all tested scenarios, and our approach is 3 times faster than STM.

Another drawback of LFTT is that its applicability is limited to linked data structures. To address this limitation, we implement a transactional vector that stores elements contiguously in a two-level array. We add a global descriptor to synchronize operations that modify the size of the vector.

This work makes the following contributions:

- We present the first wait-free hash map. Our design outperforms, by a factor of 15 or more, state of the art non-blocking designs. Our design performs a factor of 7 or greater faster than a standard blocking approach.
- We adapt LFTT to support map data structures. We implement a transactional hash map.
- We introduce the first methodology that provides dynamic lock-free data structure transactions, which we call Dynamic Transactional Transformation. Our approach allows developers to transform existing lock-free data structures into containers that support dynamic transactions, which allow code to be run between operations, and operations do not need to be known in advance. We provide templates for this approach which guide a software engineer through the transformation process.

- We add support for wait-freedom to DTT, so lock-free containers can be transformed into wait-free transactional versions.
- We propose an extension to DTT that allows any transaction to perform operations on multiple data structures. Users of our library can develop non-blocking programs without having to write a specialized transactional operation for every combination of atomic updates to every container.
- We apply our extended version of DTT to create a library of five wait-free transactional data structures, three of which have no prior transactional counterparts. With this library, a developer can write transactional programs without knowledge of the underlying algorithms for wait-free progress or transaction synchronization.
- We extend LFTT by adapting it to support contiguous data structures. We implement a transactional vector.

All of the above has been or will be released as open source software.

The pseudocode convention used in this document is $XX.YY.ZZ$, where XX is the chapter number, YY is the algorithm number within the chapter, and ZZ is the line number. All figures are also numbered within their chapter.

The rest of the dissertation is organized as follows. Chapter 2 describes background information and related work. Chapter 3 discusses the design and implementation of the wait-free hash map. Chapter 4 presents the dynamic transactional transformation methodology. Chapter 5 provides details on the non-blocking transactional hash map. Chapter 7 explains our experimental setup and shows our results. Chapter 8 is the conclusion, and future work. Finally, the Appendices contain correctness proofs for the wait-free hash map (Appendix A), DTT (Appendix B), and the transactional hash map (Appendix C).

This dissertation interpolates content from three papers by the author [40], [41], [62]. Chapter 3 includes content from [40], coauthored with Steven Feldman and Damian Dechev. Chapter 4 is based on [41], coauthored with Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Finally, Chapter 5 uses material from [62], coauthored with Deli Zhang, Lance Lebanoff, and Damian Dechev. Some material from each of these papers has been used in this introductory chapter, and the following chapter.

CHAPTER 2: BACKGROUND

As defined by Herlihy et al. [30] [31], a concurrent object is *lock-free* if it guarantees that *some* process in the system makes progress in a *finite* number of steps. An object that guarantees that *each* process makes progress in a *finite* number of steps is defined as *wait-free* [30]. By applying atomic primitives such as CAS, non-blocking algorithms, including those that are lock-free and wait-free, implement a number of techniques such as optimistic speculation and thread collaboration to provide for their strict progress guarantees. As a result of these requirements, the practical implementation of non-blocking containers is known to be difficult.

DTT is built on lock-free transactional transformation (LFTT) [61]. 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 $Y.X$.

We list the data type definitions for LFTT in Algorithm 2.1. LFTT adds a new field to the nodes stored by the base lock-free data structure, $info$, 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.

In LFTT, descriptors need to store the keys of every operation in a transaction so that conflicts can be detected if concurrent transactions attempt to operate on the same node. Since descriptors must contain all operands before a transaction begins execution, the user cannot execute any code in between the operations of a single transaction. Dynamic transactions lift this restriction, which allows developers to write applications that use transactional data structures in a way similar to STM programs, but without the drawbacks, such as, high overhead and excessive aborts due to false conflicts. Our approach preserves LFTT's semantic conflict detection.

The EXECUTEOPS function, detailed in Algorithm 4.3, is the entry point for transactional execution in LFTT. Since a transaction may be helped at any point during the execution, the $opid$ parameter indicates the operation to start at in the transaction. If at any point in the transaction one of the operations fails, the transactional execution will halt and the descriptor status will be updated by CAS to ABORTED. If all operations in the transaction successfully complete, then the descriptor status will be updated by CAS to COMMITTED.

The ISNODEPRESENT function, shown in Algorithm 2.2, determines if a node with a specific key exists in the container. This function is called prior to starting an operation. If the node exists, then the thread checks the status of the transaction descriptor at that node. If there is an active transaction, operating on that node, then the thread that identified the conflict must follow the procedure to update the node's information, detailed in Algorithm 2.3. Prior to updating the

node information, the calling thread will help complete the transaction associated with the node in conflict on line 2.3.7. Upon completing the transaction, the existence of the desired key is logically interpreted by the ISKEYPRESENT function of Algorithm 2.2. If the last transaction that accessed a node was a committed INSERT, then the key is present. If the last access was by an aborted transaction that attempted an INSERT, then it should appear as though the key was not inserted. For example, if the key is searched for, false is returned. Even though the key is present physically, there is a node linked in the data structure that contains the key, it is not logically present. This logical interpretation allows the effects of an aborted transaction to *appear to have been undone*, thus preventing the need for a physical rollback.

Nodes are logically deleted, until a transaction commits at which point they may be physically deleted. We employ the pointer marking technique described by Harris [21] to designate logically deleted nodes using a flag called $Mark$. The $Mark$ flag is set to true by setting the least significant bit of the $info$ pointer.

Conflicts occur when separate transactions contain non-commutative method calls. LFTT's helping scheme requires threads to help other transactions complete, if the threads need to operate on the same key. This helping mechanism is vulnerable to livelock if two threads access two of the same nodes in the opposite order. In order to detect and recover from livelock, each thread maintains a local help stack that contains pointers to transaction descriptors. Each thread must push the descriptor onto the help stack prior to starting a transaction, and will pop the descriptor from the help stack upon completing the transaction. A duplicate descriptor in the help stack indicates a cyclic dependency, in which the thread that detects the dependency will abort the transaction associated with the duplicate descriptor. In addition to preventing livelock, the helping mechanism reduces the number of aborts due to node access conflict to near zero. It is possible that a helping thread could be suspended just after it helps another transaction complete, and then be forced to help another transaction when it resumes. This situation could occur infinitely often, and cause the

thread to starve, which limits the progress guarantee of LFTT data structures to lock-free at best. We discuss how we support wait-freedom, which implies starvation-freedom, in Section 19.

Algorithm 2.1: 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 struct NodeInfo
17     Desc* desc;
18     int opid;
19 struct Node
20     NodeInfo* info;
21     int key;
22     ...
```

Algorithm 2.2: Logical Status

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

Algorithm 2.3: Update NodeInfo

```
1  Function UpdateInfo(Node* n, NodeInfo* info, bool wantkey)
2      NodeInfo* oldinfo ← n.info;
3      if IsMarked(oldinfo) then
4          Do_Delete(n);
5          return retry
6      if oldinfo.desc ≠ info.desc then
7          HelpTransaction(oldinfo.desc)
8      else if oldinfo.desc, oldinfo.opid + 1 then
9          return success
10     bool haskey ← IsKeyPresent(oldinfo) ;
11     if (!haskey and wantkey) or (haskey and !wantkey) then
12         return fail
13     if info.desc.status ≠ Active then
14         return fail
15     if CAS(&n.info, oldinfo, info) then
16         return success
17     else
18         return retry
```

Related Work

Overview of Work Related to Non-blocking Data Structures

Research into the design of non-blocking data structures includes: linked-lists [22], [46]; queues [49], [56], [50]; stacks [25], [50]; hash maps [46], [50], [16]; hash tables [52]; binary search trees [15], and vectors [8].

Hash Map

There are no pre-existing wait-free hash maps in the literature; as such, the related work that we discuss consists entirely of lock-free designs. In [46], Michael presents a lock-free hash map that uses linked-lists to resolve collisions; this design differs from ours in that it does not guarantee constant-time for operations after a resize is performed [52] [46]. In [16], Gao et al. present an openly-addressed hash map that is *almost* wait-free; it degrades in performance to lock-free during a resize.

In [52], Shalev and Shavit present a linked-list structure that uses pointers as shortcuts to logical buckets that allow the structure to function as a hash table. In contrast to our design, the work by Shalev and Shavit does not present a hash map and it is lock-free. There was a single claim of a wait-free hash map that appeared as a presentation by Cliff Click [5]; the author now claims lock-freedom. Moreover, the work by Click was not published. A popular concurrent hash map that is part of Intel's Threading Building Blocks (TBB) [34] library is claimed to be lock-free, but is also unpublished.

Vector

The design of our transactional vector is based on the two-level arrays used in [8]. Dechev et al. present the first lock-free vector in the literature. Their design uses descriptor objects to synchronize updates to the vector in a linearizable manner. The design in [8] does not support transactions or bounds-checking.

In [57], the authors modify the vector in [8], by adding a lock-free version of flat combining. Flat combining [24], batches operations together, which allows for performance gains. These batches are similar to transactions, except they are only executed by a single thread. The batches are also

dissimilar from transactions, because the user cannot decide which operations are included in a batch, the batches are created by the flat combining algorithm.

In [13], the authors discuss an implementation of a wait-free vector that uses a software multi-word compare-and-swap operation to provide a vector with an API that is more similar to that of the C++ STL vector. Feldman et al. provide another benefit over [8], bounds-checking; we also support bounds-checking in our vector.

Overview of Work Related to Transactional Data Structures

Significant research has been devoted to non-blocking linked data structures [21, 43, 58, 47] because their distributed memory layout provides data access parallelism and scalability under high levels of contention. Both STM and hardware transactional memory (HTM) are considered as potential candidates for achieving the atomicity required for non-blocking data structure operations. Transactions on a data structure by traditional methods involve executing all shared memory accesses in coarse-grained *atomic sections*. High-level conflict detection approaches [26, 39, 2] avoid false conflicts due to low-level accesses, but the performance degrades to coarse-grained locking in the presence of non-commutative operations. DTT overcomes these challenges by performing high-level conflict detection while providing the strongest progress guarantee of wait-freedom, in addition to supporting multi-container transactions and dynamic transactions.

Transactional Memory

Transactional memory, initially proposed as a set of hardware extensions by Herlihy and Moss [29], was intended to facilitate the development of lock-free data structures. The potential for advancing concurrent programming led to the development of Intel's Haswell microarchitecture, which offers

support for HTM. However, HTM's cache-coherency based conflict detection causes transactions to be vulnerable to spurious failures during page faults and context switches [9]. Under Intel's proposed solution, the performance of applications that frequently encounter data access conflicts will degrade to coarse-grained locking. These shortcomings make HTM undesirable for data structure implementations.

The first STM, proposed by Shavit et al. [53], is lock-free but only supports a static set of data items. Herlihy et al., later presented DSTM [28] that supports transactions for dynamic-sized data structures and guaranteed the weaker progress guarantee of obstruction-freedom [27]. Since STM detects conflicts at the granularity of read and write accesses, excessive aborts due to frequent accesses on a data structure such as the head node substantially limit concurrency. In order to deliver high-performance large-scale transactional applications, DTT enables transactions comprising multiple data structures that do not suffer from performance degradation due to low-level conflicts. We leverage knowledge of each data structure in our presented library to detect conflicts only on non-commutative operations and enact a cooperative transaction execution, which eliminates false conflicts and significantly reduces aborts.

Spiegelman, et al. [54] 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. For every write operation, TDSL creates a write element that has a next field, a value and a boolean deleted field. The write element is inserted into the write set at the time an operation is performed. When a transaction is ready to commit, it locks the nodes in the write-set, and then it validates that all nodes in the read-set are unchanged by checking their version numbers. It then proceeds to update the nodes in the write set by changing their next field, value, and deleted field according to the write element that is mapped to the node in the write set. 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, unlike DTT. Also, as multiple operations occur in a transaction,

the information stored in the next field of the write element may no longer be valid. This problem surfaces when operations are performed on adjacent nodes. TDSL currently does not update write elements when operations in the transaction may cause the write element information to change. This causes some nodes to be linked to logically deleted nodes at commit time. An operation must re-traverse the list if it encounters a logically deleted node, leading to a continuous re-traversal if the deleted node is never physically removed from the list. We do not compare to their approach due to the deadlock situation resulting from this continuous re-traversal.

Transactional Boosting

The penalty of aborted transactions due to conflicts has motivated semantic-based approaches that propose to identify conflicts at a high-level [26, 39, 2, 23, 19, 18] which enables greater parallelism. Since transactions that are semantically independent may have low-level memory access conflicts, the transactions can proceed concurrently while using some other concurrency control protocol to protect accesses to the underlying data structure. Transactional boosting [26] is a semantic-based methodology for transforming linearizable concurrent data structures into transactional data structures. If two operations commute, they are allowed to proceed concurrently using thread-level synchronization within the operations; otherwise, their enclosing transactions need to be synchronized.The base data structure is treated as a black box and the use of *abstract locking* ensures that non-commutative method calls do not occur concurrently. For each operation in a transaction that does not commute with an operation in a concurrent transaction, the boosted data structure must acquire the abstract lock associated with the method. A transaction aborts if it fails to acquire an abstract lock, and it performs a physical rollback by invoking the inverses of operations which have already been executed. Our approach overcomes performance penalties of invoking the inverse operations for aborted transactions by observing that the operations of an aborted transaction need only appear to be undone. We instead perform a logical rollback by

inversely interpreting the status of a node, thus avoiding the overhead of a physical rollback.

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. DTT provides the strongest progress guarantee of wait-freedom for transaction-level synchronization. In contrast to transactional boosting, we provide a library of containers for developers, so that they can start writing programs that use transactional data structures without being required to transform some first.

LFTT

Zhang et al. [61] present LFTT, a methodology for transforming high-performance lock-free linked data structures into lock-free transactional containers. 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.

The key advantage of using DTT over LFTT is that dynamic transactions allow the user to 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.

CHAPTER 3: WAIT-FREE HASH MAP

Algorithms

In this section we define a semantic model of the hash map's operations, address concerns related to memory management, and provide a description of the design and the applied implementation techniques. The presented algorithms have been implemented, in both ISO C and ISO C++, and designed for execution on an ordinary, multi-threaded, shared-memory system; we require only that it supports atomic single-word read, write, and CAS instructions.

Structure and Definition

Our hash map is a multi-level array which has a structure similar to a tree; this is shown in Figure 3.1. Our multi-level array differs from a tree in that each position on the tree could hold an array of nodes or a single node. A position that holds a single node is a $\mathrm{dataNode}$ which holds the hash value of a key and the value that is associated with that key; it is a simple struct holding two variables. Since a $\mathrm{dataNode}$ is at least two memory words we cannot read it atomically, so we must have a way to prevent interference with nodes that are being read or are otherwise in use; we call our method of doing this, "watching" (see Section 3). A $\mathrm{dataNode}$ in our multi-level array could be marked. A $\mathrm{markedDataNode}$ refers to a pointer to a $\mathrm{dataNode}$ that has been bitmarked at the least significant bit (LSB) of the pointer to the node. This signifies that this $\mathrm{dataNode}$ is contended. An expansion must occur at this node; any thread that sees this $\mathrm{markedDataNode}$ will try to replace it with an $\mathrm{arrayNode}$; which is a position that holds an array of nodes. The pointer to an $\mathrm{arrayNode}$ is differentiated from that of a pointer to a $\mathrm{dataNode}$ by a bitmark on the second-least significant bit.

Our multi-level array is similar to a tree in that we keep a pointer to the root, which is a memory array that we call head. The length of the head memory array is unique, whereas every other $\mathrm{arrayNode}$ has a uniform length; a normal $\mathrm{arrayNode}$ has a fixed power-of-two length equal to the binary logarithm of a variable called $\mathrm{arrayLength}$. The maximum depth of the tree, $\mathrm{maxDepth}$, is the maximum number of pointers that must be followed to reach any node. We define $\mathrm{currentDepth}$ as the number of memory arrays that we need to traverse to reach the $\mathrm{arrayNode}$ on which we need to operate; this is initially one, because of head.

Our approach to the structure of the hash map uses an extensible hashing scheme; we treat the hash value as a bit string and rehash incrementally [12]. We use $\mathrm{arrayLength}$ to determine how many bits are necessary to ascertain the location at which a $\mathrm{dataNode}$ should be placed within the $\mathrm{arrayNode}$. The hashed key is expressed as a continuous list of $\mathrm{arrayPow}$-bit sequences, where $\mathrm{arrayPow}$ is the binary logarithm of the $\mathrm{arrayLength}$; e.g. $A - B - C - D$, where A is the first $\mathrm{arrayPow}$-bit sequence, B is the next $\mathrm{arrayPow}$-bit sequence, and so on; these represent positions on different $\mathrm{arrayNodes}$. These bit sequences are isolated using logical shifts. We use R to designate the number of bits to shift right, in order to isolate the position in the $\mathrm{arrayNode}$ that is of interest. R is equal to $\log_2 \texttt{arrayLength} * \texttt{currentDepth}$. For example, in a memory array of length $64 = 2^6$, we would take $\mathrm{R} = 6$ bits for each successive $\mathrm{arrayNode}$.

The total number of arrays is bounded by the number of bits in the key (which is stored in a variable called $\mathrm{keySize}$) divided by the number of bits needed to represent the length of each array. For example, with a 32-bit key and an $\mathrm{arrayLength}$ of 64, we have a $\mathrm{maxDepth}$ of 6, because $\lceil 32 / \log_2 64 \rceil = 6$. This places no limit on the total number of elements that can be stored in the data structure; the hash map expands to hold all unique keys that can be represented by the number of bits in the key (even beyond the machine's word size). We have tested with multiword keys, such as the 20 bytes needed for SHA1. Neither an $\mathrm{arrayNode}$ nor a $\mathrm{markedDataNode}$ can be present in an $\mathrm{arrayNode}$ whose $\mathrm{currentDepth}$ is equal to $\mathrm{maxDepth}$, because no hash collisions

can occur there.

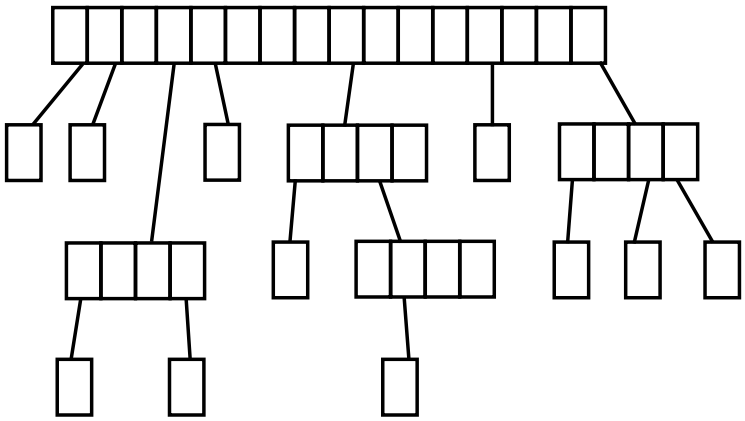


Figure 3.1: An illustration of the structure of the hash map.

Traversal

Traversing the hash map is done by performing a right logical shift on the hashed key to preserve R bits, and examining the pointer at that position on the current memory array. If the pointer stores the address of an $\mathrm{arrayNode}$, then the $\mathrm{currentDepth}$ increases by one, and that position on the new memory array is examined.

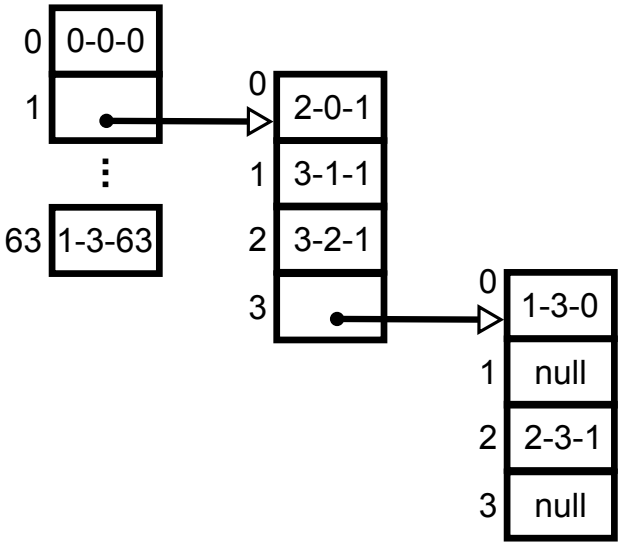


Figure 3.2: An example of data stored in the hash map (values not shown).

We discuss the traversal of the hash map using Figure 3.2 as an illustration of this process. In our example, the arrayNodes have a length of four, which means that exactly two bits are needed to determine where to store our $\mathrm{dataNode}$ on any particular $\mathrm{arrayNode}$, except for head which has a larger size than every other $\mathrm{arrayNode}$ (see Section 3). The hashed key is expressed as a finite list of two-bit sequences e.g. $A - B - C$, where C is the first three-bit sequence, and so on; these sequences represent positions at various depths.

For example, if we need to find the key 0-4-2, in the hash map shown in Figure 3.2, then we first need to hash the key. We assume that this operation yields 2-3-1. To find 2-3-1 we first take the right-most set of bits, and go to that position on head. We see that this is an $\mathrm{arrayNode}$, so we take the next set of bits which leads us to examine position 3 on this $\mathrm{arrayNode}$. This position is also an $\mathrm{arrayNode}$, so we take the next set of bits which equal 2, and examine that position on this $\mathrm{arrayNode}$. That position is a $\mathrm{dataNode}$, so we compare its hashed key to the hashed key that we are searching for. The comparison reveals that the hash values are both equal to 2-3-1, so we return

the value associated with this dataNode.

Main Functions

In this section we provide a brief overview of the main operations implemented by our hash map. Unless otherwise noted, all line numbers refer to the current algorithm being discussed. In other sections of the paper, the main functions are referred to by the first letter of the function name followed by the line number of interest; supporting functions are referred to by their full name. In all algorithms, local is the name of the arrayNode that an operation is working on and pos is the position on local that is of interest. The variable failCount is a thread-local counter that is incremented whenever a CAS fails and the thread must retry its attempt to update the hash map. Instances of this variable are compared to the maxFailCount which is a user-defined constant used to bound the maximum number of times that a thread retries an operation after a CAS operation fails. If this bound is reached, then an expansion is forced at the position that the failing operation is attempting to modify.

The CAS operation that we use is part of C++11; the function that we use returns the value that the memory address held before the execution of the operation. If our functions are implemented in a system that does not have a sequentially consistent memory model, then memory fences are needed to preserve the relative order of critical memory accesses [46].

Algorithm 1 - insert (key, value)

The insert function is used to insert a key-value pair into the hash map. The function returns true if the key is not in the hash map, and false if the key is already there; this allows us to prevent the user from performing unintended overwrites of elements in the hash map. We provide an update

operation for the case wherein a user would like to change the value that is associated with a key that is already in the hash map (see Section 3).

An insert operation traverses the hash map as described in Section 3 until it finds a position that is null or that contains a dataNode. If the position is null, then a CAS is performed; this is shown on line 13. If the CAS is successful, then the function returns true. If a dataNode whose key matches the key that is being inserted, is encountered during the traversal, then the function returns false. If it is a dataNode whose key is different, then the thread calls expandMap at the position (resolving the hash collision); if the expansion is successful, then the thread continues its traversal from the new arrayNode that was added.

If the CAS at line 13 failed, then the CAS operation has returned either a dataNode or an arrayNode. If an arrayNode was returned, then the thread continues traversal from the arrayNode. If the result is a dataNode whose key matches the key that is being inserted, then the function returns false; if it does not match, then it calls expandMap at the position.

If a call to expandMap fails, then the failCount is incremented and the return value is examined. If failCount equals maxFailCount, then an atomic bitmark is placed on the contents of local at pos, and expandMap is called. When expandMap returns, the thread continues traversal from the arrayNode that is guaranteed to be returned (see Section 3). For this situation to arise, the position that this thread wants to insert into must be highly-contended, so new arrayNodes are added until the thread can insert without interference from another thread.

The linearization point of this operation, when it returns true, is the CAS on line 13. The same CAS is one of the linearization points when the function returns false, the other two are the atomic reads on lines 8 and 23.

Algorithm 3.1 insert $key, value$

```
 1: hash=hashKey(key);
 2: local=head;
 3: for int r=0; r <keySize−arrayPow;r+=arrayPow do
 4:     pos=hash&(arrayLength−1);
 5:     hash=hash>>arrayPow;
 6:     failCount=0;
 7:     node=getNode(local,pos);
 8:     while true do
 9:         if failCount>maxFailCount then
10:             node=markDataNode(local,pos);
11:         if node==null then
12:             insertThis=allocateNode(value,hash);
13:             if (node=CAS(local[pos],null, insertThis))==null then
14:                 watch(null);
15:                 return true;
16:             else
17:                 free(insertThis);
18:         if isMarked(node) then
19:             node=expandMap(local,pos,r);
20:         if isArrayNode(node) then
21:             local=node;
22:             break;
23:         else
24:             watch(node);
25:             node2=getNode(local,pos)
26:             if node != node2 then
27:                 failCount++;
28:                 node=node2;
29:                 continue;
30:             else if node−>hash == hash then
31:                 watch(null);
32:                 return false;
33:             else
34:                 node=expandMap(local,pos,r);
35:                 if isArrayNode(node) then
36:                     local=node;
37:                     break;
38:                 else
39:                     failCount++;
40: free(insertThis);
41: watch(null);
42: pos=hash&(arrayLength−1);
43: currValue=local[pos];
44: if currValue == null then
45:     return (CAS(local[pos],null, value)==null);
46: else
47:     return false;
```

Algorithm 2 - Update (key, expectedValue, newValue)

The update function is used to update the value associated with a key that is present in the hash map. This function takes three arguments: the first is the key whose value we would like to update, called key; the second is the value that we expect to be associated with this key, called expectedValue; and the third is the value that we would like to associate with this key, called

newValue. The update function returns true, if it successfully replaces a dataNode whose key and value matches the key and expectedValue of this operation. If the key is not present in the hash map, or if the key's associated value does not match expectedValue, then the function returns false. In order to reason about the results of a failed CAS operation we require expectedValue to be different from newValue.

The update operation traverses the hash map as described in Section 3, until it finds a position that is null, or that contains a dataNode. If a markedDataNode is found during the traversal, then expandMap is called and the thread continues its traversal. If it is a dataNode whose key matches the one being updated, and the value in the dataNode matches expectedValue, then a CAS is performed which replaces the current dataNode with one containing newValue.

If the CAS fails, then the return value is examined. If it is a marked version of the node that the CAS attempted to replace, then the thread calls expandMap and continues its traversal. If the value returned is an arrayNode, then the thread continues its traversal. An arbitrary dataNode, null, or a dataNode whose key and value matches could have been returned as well; the first two indicate that the operation should return false. The return of a dataNode whose key and value matches may seem like a successful result; however, it is actually an indication that we may be experiencing the ABA problem. The reasoning is that because we placed the constraint that expectedValue may not be equal to newValue, then there must have been a state where the key was not present, or the value associated with the key did not match expectedValue in order for the CAS to have failed, so we return false in this case. If the traversal is completed without finding a dataNode with a key-value pair that matches key and expectedValue, then the function returns false.

There are several linearization points. Two of these are the atomic reads in the calls to getNode at lines 7 and 19; another two of these are the CAS operations at lines 37 and 54. If update returns true, then it linearizes upon the return of the appropriate CAS operation. If any of the four

lines returns null or a pointer to a dataNode whose key and value does not match the key and expectedValue of this operation, causing update to return false, then it is at that point that the operation linearizes. The third point occurs when a failed CAS operation returns a pointer to a dataNode whose key and value matches the expected, then the linearization point is between the atomic read in getNode and the the completion of the CAS operation. There must have been a state when either the key was not in the map, or the value associated with the key did not match expectedValue, and it is at this state that the operation linearizes.

In the worst case, this operation requires expandMap to be called until maxDepth is reached, at which point it is not possible for there to be any more expansions, by definition of maxDepth and the constraints on the hash function. Therefore, at this point, the thread will be able to finish its operation with a single CAS or atomic read.

Algorithm 3 - get (key)

The get operation traverses the hash map as described in Section 3, until it finds a position that is null, or that contains a dataNode. If it is a dataNode whose key matches, then the value associated with the key is returned; otherwise, null is returned.

The point at which this operation linearizes is the atomic read in the call to getNode (see lines 7 and 17). If a dataNode is read, then this thread must announce that it is about to read the node, by calling the watch function. If the value changed between the read and the call to watch, then the thread retries. If it retries more than maxFailCount times, then the thread will mark the address as highly-contended and force an expansion; the number of times that this can occur is equal to maxDepth. If maxDepth is reached, then the thread can no longer read dataNodes, only null or values, as such the thread simply returns the value that it reads at this level (see Section 3).

Algorithm 3.2 Update $key, expectedValue, newValue$

```
 1: hash=hashKey(key);
 2: local=head;
 3: result=false;
 4: for int r=0; r<keySize−arrayPow;r+=arrayPow do
 5:     pos=hash&(arrayLength−1);
 6:     hash=hash>>arrayPow;
 7:     node=getNode(local,pos);
 8:     if isArrayNode(node) then
 9:         local=node;
10:     else if isMarked(node) then
11:         local=expandMap(local,pos,r);
12:     else if node==null then
13:         break;
14:     else
15:         watch(node);
16:         if node != getNode(local,pos) then
17:             failCount=0;
18:             while node != getNode(local,pos) do
19:                 node=getNode(local,pos);
20:                 watch(node);
21:                 failCount++;
22:                 if failCount>maxFailCount then
23:                     markDataNode(local,pos);
24:                     local=expandMap(local,pos,r);
25:                     break;
26:             if isArrayNode(node) then
27:                 local=node;
28:                 continue;
29:             else if isMarked(node) then
30:                 local=expandMap(local,pos,r);
31:                 continue;
32:             else if node==null then
33:                 break;
34:         if node−>hash == hash then
35:             if node−>value != expectedValue then
36:                 break;
                insertThis=allocateNode(newValue,hash);
37:             if (node2=CAS(local[pos],node,insertThis))==node then
38:                 result= true;
39:                 break;
40:             else
41:                 free(insertThis);
42:                 if isArrayNode(node2) then
43:                     local=node2;
44:                 else if isMarked(node2)∧unmark(node2)==node then
45:                     local=expandMap(local,pos,r);
46:                 else
47:                     break;
48:         else
49:             break;
50: if r ¿= keySize−arrayPow then
51:     pos=hash&(arrayLength−1);
52:     currValue=local[pos];
53:     if currValue == expectedValue then
54:         result= (CAS(local[pos], expectedValue, newValue) == expectedValue);
55:     else
56:         result=false;
57: else if result then
58:     safeFreeNode(node);
59: watch(null);
60: return result;
```

Algorithm 3.3 get key

```
 1: hash=currHash=hashKey(key);
 2: local=head;
 3: result=null;
 4: for int right=0;right<keySize−arrayPow;right+=arrayPow do
 5:     pos=hash&(arrayLength−1);
 6:     hash=hash>>arrayPow;
 7:     node= getNode(local,pos);
 8:     if isArrayNode(node) then
 9:         local=node;
10:     else if node==null then
11:         break;
12:     else
13:         watch(node);
14:         if node != getNode(local,pos) then
15:             failCount=0;
16:             while node != getNode(local,pos) do
17:                 node=getNode(local,pos);
18:                 watch(node);
19:                 failCount++;
20:                 if failCount>maxFailCount then
21:                     markDataNode(local,pos);
22:                     local=expandMap(local,pos,r);
23:                     break;
24:             if isArrayNode(node) then
25:                 local=node;
26:                 continue;
27:             else if isMarked(node) then
28:                 local=expandMap(local,pos,r);
29:                 continue;
30:             else if node==null then
31:                 break;
32:         if node−>hash == currHash  then
33:             result=node−>value;
34:         break;
35: if r ¿= keySize−arrayPow then
36:     pos=hash&(arrayLength−1);
37:     result=local[pos];
38: watch(null);
39: return  result;
```

Algorithm 4 - remove (key, expectedValue)

The remove operation is nearly identical to the update operation, it can be treated as a specialized version of update where the only difference is that instead of replacing a dataNode with another dataNode, it replaces it with null. It has the same logic for determining when an operation returns true or false, the same bound on the number of loop iterations, and the same linearization points.

Algorithm 3.4 remove $key, expectedValue$

```
 1: currHash=hash=hashKey(key);
 2: local=head;
 3: result=false;
 4: for int r=0; r<keySize−arrayPow;r+=arrayPow do
 5:     pos=hash&(arrayLength−1);
 6:     hash=hash>>arrayPow;
 7:     node=getNode(local,pos);
 8:     if isArrayNode(node) then
 9:         local=node;
10:     else if isMarked(node) then
11:         local=expandMap(local,pos,r);
12:     else if node==null then
13:         break;
14:     else
15:         watch(node);
16:         if node != getNode(local,pos) then
17:             failCount=0;
18:             while node != getNode(local,pos) do
19:                 node=getNode(local,pos);
20:                 watch(node);
21:                 failCount++;
22:                 if failCount>maxFailCount then
23:                     markDataNode(local,pos);
24:                     node=expandMap(local,pos,r);
25:                     break;
26:             if isArrayNode(node) then
27:                 local=node;
28:                 continue;
29:             else if isMarked(node) then
30:                 local=expandMap(local,pos,r);
31:                 continue;
32:             else if node==null then
33:                 break;
34:         if node−>hash == currHash then
35:             if node−>value != expectedValue then
36:                 break;
37:             if (node2=CAS(local[pos],node,null))==node then
38:                 safeFreeNode(node);
39:                 result= true;
40:                 break;
41:             else
42:                 if isArrayNode(node2) then
43:                     local=node2;
44:                 else if isMarked(node2)∧unmark(node2)==node then
45:                     local=expandMap(local,pos,r);
46:                 else
47:                     break;
48:         else
49:             break;
50: if r ¿= keySize−arrayPow then
51:     free(insertThis);
52:     pos=hash&(arrayLength−1);
53:     currValue=local[pos];
54:     if currValue ==expectedValue then
55:         result = (CAS(local[pos], expectedValue, null) == expectedValue);
56:     else
57:         result=false;
58: watch(null);
59: return result;
```

Algorithm 5 - expandMap (local, pos, right)

This function is used to expand the map when there is a hash collision. If the current value at pos in local is marked, then it is guaranteed that when the function returns, the contents of pos in local are an arrayNode that holds an unmarked version of the node that was there before.

First, expandMap reads the current value at pos. If it is not an arrayNode, then it allocates a new one, calculates the position where the node that was there previously belongs on thearrayNode, and sets the pointer at that position equal to the location of the node. Next, it uses a CAS to attempt to replace that node with the arrayNode (see line 10). This function returns the allocated arrayNode, if the CAS is successful; otherwise, it returns false.

The atomic read in the call to getNode on line 1 is the linearization point, if this operation returns false; the CAS on line 10 is the linearization point, if this operation returns true.

An optimization that we use in the implementation is that if an operation is attempting to insert a node that collides with a node that is currently in the map, then the expandMap algorithm creates an arrayNode or a series of them, that contains both nodes, and then performs the CAS.

Algorithm 3.5 expandMap $local, pos, right$

```
 1: node= getNode(local,pos);
 2: watch(node);
 3: if isArrayNode(node) then
 4:     return node;
 5: if node !=(node2=getNode(local,pos)) then
 6:     return node2;
 7: aNode=alloc(sizeof(arrayNode));
 8: newPos=(node->hash>>(right+arrayPow))& (arrayLength-1);
 9: aNode[newPos]=node;
10: if (node2=CAS(local[pos]), node, aNode)) == node then
11:     return aNode;
12: else
13:     aNode[newPos]=null;
14:     free(aNode);
15:     return node2;
```

Memory Management

This section discusses the allocation and reuse of memory. When designing concurrent applications, choosing an appropriate memory management scheme is important, and the one chosen must be thread-safe. As the standard memory allocator is blocking, special provisions must be made for lock-free and wait-free programs. In order for the hash map to behave in a wait-free manner, the user must choose a memory allocator that can manage memory in a wait-free manner [55].

Furthermore, this memory manager must be able to handle the ABA problem [7] correctly, because this problem is fundamental to all CAS-based systems [48]. To prevent the ABA problem we ensure that the values stored in the $\mathrm{dataNode}$ remain unchanged while any thread is using that $\mathrm{dataNode}$. Any update to the value associated with a key is done by replacing the $\mathrm{dataNode}$ that is associated with that key with a new one with the same key. To achieve this we used Michael's ABA-free approach to safe memory-reclamation, called hazard pointers [48].

Hazard pointers work by having each thread announce the address of the memory it is about to access [48]. In our algorithm each thread performs an atomic read at a position on an $\mathrm{arrayNode}$ and if it is a $\mathrm{dataNode}$, the thread writes the address of the $\mathrm{dataNode}$ to a global array. The thread then checks to ensure that, between reading the $\mathrm{dataNode}$ and writing to the global array, the node was not removed from that location. If it was removed, then the thread retries; this retrying is what makes some other algorithms that use hazard pointers lock-free. In our algorithm we using the atomic bitmark and expansion to bound the number of times a retry is attempted. In practice retrying rarely occurs. Additionally, since values and not $\mathrm{dataNodes}$ are stored on the $\mathrm{arrayNodes}$ located at max depth, there is no need to perform a hazard pointer read at max depth, and the value read can be operated on without concern.

Michael's hazard pointer implementation is wait-free if you can place a reference into the watched

address list in a wait-free manner. This consists of reading the contents of an address, storing the value read into the global list, re-reading the contents, and comparing the two values to ensure that they are the same. If they are different it must retry until they are the same. In most algorithms this process is lock-free, because the number of times the algorithm must retry is not bounded. That is not the case in our algorithm, because of how we use atomic bitmarks and the fact that an $\mathrm{arrayNode}$ cannot be removed. The wait-free property of hazard pointers and the minor adjustments made to implement this algorithm in our code mean that watch and $\mathrm{safeFreeNode}$ are both wait-free (see [48]).

There are several existing approaches to wait-free memory management. An approach that includes wait-free memory allocation and reclamation is found in [55]. For testing purposes we use the Lockless library [44] for lock-free memory allocation, and hazard pointers for wait-free memory reclamation as presented in [48]. To make the entire system wait-free, the user would have to supply their own wait-free memory allocator as the system calls involved in the allocation of memory are beyond the scope of this paper.

Algorithm 6 - $\mathrm{watch}\ (\mathrm{value})$

This function uses a thread-local variable, $\mathrm{threadID}$, and a global array, $\mathrm{watchedNodes}$, to alert other threads of the node a particular thread is using. Watching is done before any read or write operations on the hash map. Each thread has a unique value form 0 to $\mathrm{Threads}$ as their $\mathrm{threadID}$, this corresponds to the position on the $\mathrm{watchedNodes}$ array where it stores the node that it is about to use. For more information please review Section 3.

Algorithm 3.6 watch $value$

```
1: watchedNodes[threadID]=value;
```

Algorithm 7 - safeFreeNode (nodeToFree)

This function is used to ensure that memory is not freed while another thread is using it. It checks the watchedNodes array for the address of nodeToFree, and if it is not present, then the node is freed. If it is present, then the nodePool (a thread-local linked list that holds pointers to nodes that we want to remove from the map, but cannot because they are in watchedNodes) is checked for nodes that are no longer being used, if one is found then that node is freed and this node takes its place in the nodePool. Otherwise, additional space is added for this node.

Algorithm 3.7 safeFreeNode $nodeToFree$

```
 1: freeable=true;
 2: for int i=0; i<Threads; i++ do
 3:     if i==threadID then
 4:         continue;
 5:     else if nodeToFree == watchedNodes[i] then
 6:         freeable=false;
 7:         break;
 8: if freeable then
 9:     free(nodeToFree);
10: else
11:     list=nodePool[threadID];
12:     while list != null do
13:         node=list−¿value;
14:         freeable=true;
15:         for int i=0; i<Threads; i++ do
16:             if i==threadID then
17:                 continue;
18:             else if node == watchedNodes[i] then
19:                 freeable=false;
20:                 break;
21:         if freeable then
22:             free(list−¿value);
23:             list−¿value=nodeToFree;
24:             return  ;
25:         else
26:             list=list−¿next;
27:     pNode=allocate();
28:     pNode−¿next = list;
29:     pNode−¿value=nodeToFree;
30:     nodePool[threadID]=pNode;
```

Algorithm 8 - allocateNode (value, hash)

This function reuses nodes that have been stored in the nodePool; if no node is available, then a new node is allocated. The thread first checks its thread-local nodePool for a node that is no longer

being referenced; if a node is found, then the thread returns a pointer to that node; otherwise, the thread allocates a new node.

Algorithm 3.8 allocateNode $value, hash$

```
 1: ppNode=pNode=nodePool[threadID];
 2: node = null;
 3: while pNode != null do
 4:     freeable=true;
 5:     for int i=0; i<Threads; i++ do
 6:         if i==threadID then
 7:             continue;
 8:         else if pNode−¿value == watchedNodes[i] then
 9:             freeable=false;
10:             break;
11:     if freeable then
12:         if ppNode==pNode then
13:             nodePool[threadID]=pNode−¿next;
14:         else
15:             ppNode−¿next=pNode−¿next;
16:         node=pNode−¿value;
17:         free(pNode);
18:         break;
19:     else
20:         ppNode=pNode;
21:         pNode=pNode−¿next;
22: if node == null then
23:     node=allocate();
24: node−¿value=value;
25: node−¿hash = hash;
26: return node;
```

Supporting Functions

This section briefly describes the supporting functions referenced in the pseudocode of the preceding algorithms.

Algorithm 3.9 getNode $local, pos$

```
1: res=&local[pos];
2: return res;
```

Algorithm 3.10 isMarked $node$

```
1: res=(node&0x1);
2: return res;
```

Algorithm 3.11 isArrayNode $node$

```
1: res=(node&0x2);
2: return res;
```

Algorithm 3.12 markDataNode $local, pos$

```
1: address=&local[pos];
2: res= atomic_OR_and_fetch(address,0x1)
3: return res;
```

Algorithm 3.13 unmark $node$

```
1: res=(node — 0x1);
2: return res;
```

CHAPTER 4: DYNAMIC TRANSACTIONAL TRANSFORMATION

In this chapter, we provide a broad overview of our approach. Then, we provide an example of how to use DTT. Finally, we describe the details of the implementation, and extensions for wait-freedom and multi-container transactions.

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 arbitrary side-effect free 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 of code. 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 while supporting more features. 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 others 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 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.

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.

Algorithm 4.1 shows an example of a block of code written for STM, where the transaction's start and end are marked by TX_BEGIN and TX_END, respectively. The transformed code using our library is shown in Algorithm 4.2, 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 4.2.2-4.2.3) Then the user calls the EXECUTETRANSACTION method to run the transactional function (line 4.2.5). After the transaction completes execution, the user can access variables from the output map (line 4.2.6). 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 4.2.9, 4.2.11, and 4.2.13). Accesses to variables that were added to the *input map* are handled by accessing the local hash map (lines 4.2.8 and 4.2.10).

Implementation Details

We now explain the details of the library's underlying methods that allow it to execute the user's transactional function.

Algorithm 4.1: Example of Original Code

```
1 Function OriginalCode()
2     int x ← 3;
3     TX_BEGIN();
4     T val ← skiplist.FIND(x);
5     bool success ← skiplist.INSERT(4, val);
6     if success = true then
7         skiplist.DELETE(5)
8     TX_END();
9     PRINT(val);
```

Algorithm 4.2: Example of Transformed Code

```
1 Function Main()
2     HashMap* inputMap ← new HashMap();
3     inputMap.PUT("x", 3);
4     HashMap* outputMap ← Null;
5     EXECUTETRANSACTION(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;
```

The EXECUTETRANSACTION function, shown in Algorithm 4.3, 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 4.3.4-4.3.6). Then we call the HELPTRANSACTION method, also shown in Algorithm 4.3.

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*. In line 4.3.13, we check for a cyclic dependency in the help stack. If so, we abort the transaction

(line 4.3.14). Otherwise, we can proceed by adding the current transaction to the thread's help stack (line 4.3.16). This procedure is inherited from LFTT to prevent the livelock situation described in Section 2. Then we copy the data contained from the input map into the local map (line 4.3.17). 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 4.3.18). 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 4.2. 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 4.3.20-lines 4.3.20). 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 4.3.26). This allows the user to extract values of local variables from the output map after the transaction has executed.

The CALLOP method, shown in Algorithm 4.4, calls a data structure operation. Before performing the operation, we check if the transaction has already been aborted in the case of a cyclic dependency or failed operation, and if so, it can be skipped (line 4.4.3). 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 4.4.4). 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 be-

Algorithm 4.3: Transaction Execution

```
1  thread_local Stack helpstack;
2  Function ExecuteTransaction(Function* func, HashMap* inputMap, HashMap*
     outputMap)
3      helpstack.INIT() ;
4      desc.func = func;
5      desc.inputMap = inputMap;
6      desc.outputMap = outputMap;
7      HELPTRANSACTION(desc) ;
8      return desc.status = Committed
9  ;
10 Function HelpTransaction(Desc* desc)
11     bool ret ← true;
12     set delnodes;
13     if helpstack.CONTAINS(desc) then
14         CAS(&desc.flag, Active, Aborted) ;
15         return
16     helpstack.PUSH(desc) ;
17     HashMap* localMap ← COPY(desc.inputMap);
18     ret ← desc.FUNC(desc, localMap);
19     helpstack.POP() ;
20     if ret = true then
21         if CAS(&desc.flag, Active, Committed) then
22             MARKDELETE(delnodes, desc)
23     else
24         CAS(&desc.flag, Active, Aborted)
25     if desc.outputMap does not exist then
26         desc.outputMap ← COPY(localMap)
```

cause 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 4.4.5). If so, that means that another thread has already performed this operation, so

Algorithm 4.4: Call Operation

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

the current thread avoids duplicate work by simply returning the value from the return values list corresponding to the current operation (line 4.4.6). 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 4.4.9). 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. Although DTT does not require a list of operations as input, the *ops* list is still needed for the logical interpretation scheme inherited from LFTT, discussed in Section 2. 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 4.4.17). Then any helping threads that encounter this operation in the future will be able to observe that the work has already been done for this operation and skip it, as in line 4.4.3. Then, the thread's help stack is updated to increment the index of the current operation within the current transaction (line 4.4.18).

Transactions Among Multiple Data Structures

Our methodology for transactions among multiple data structures adopts the node-based conflict detection and logical rollback presented in LFTT.

A drawback of LFTT is that atomic operations among containers are not possible. An example of the need for atomic operations with multiple data structures is moving elements between sets without duplicates, with the restriction that elements not become inaccessible to other transactions for any period of time. We need to remove an item from one set, and insert it into another in what appears to be one indivisible step. We can achieve atomicity in this case by modifying the transaction descriptor, so that each operation stores a reference to the container on which the operation should be performed.

We add a *container* field to an operation, which stores a reference to the data structure that a particular operation should be executed on. This information is added to every operation in a transaction descriptor. Once all of the operations have been created, the EXECUTE function of the transaction descriptor is called. The EXECUTE function is a wrapper function that calls the EXECUTEOPS function of the data structure referenced by the *container* field.

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 [38]. 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 [38].

Wait-free Transactions - Pseudocode

We ensure wait-free progress for each operation within a transaction through our progress assurance scheme, as shown in Algorithm 4.5. Let n be the number of threads in the system. An announcement table of length n, shown on line 4.5.1, is maintained such that a delayed thread t_i may post a descriptor, a NODEINFO, at position i to alert the other threads that it needs help completing an operation. Prior to starting an operation within a transaction, a thread will increment a delay counter $delayCount$. Once the $delayCount$ reaches a constant $HELP\_DELAY$ operations on line 4.5.8, the thread will check the announcement table to determine if the thread it is assigned to help has a pending operation. If the thread to be helped has posted a transaction in the announcement table, the helping thread will execute the entire transaction starting from the current operation id, shown on line 4.5.12. Upon completing the transaction or determining that no help is required, the $helpId$ is updated to the next thread to be helped on line 4.5.13 and the thread will proceed to begin its own operation. Each thread is given $MAX\_FAILURES$ attempts to

complete its operation in a lock-free manner, shown on line 4.5.26. After $MAX\_FAILURES$ attempts, a thread will post its transaction information in the announcement table on line 4.5.18 and continue to attempt to execute its own transaction. The transaction information is removed from the announcement table when the transaction has either committed or aborted.

The progress assurance scheme guarantees wait-free progress because in a worst case scenario, all threads will eventually reach a delayed thread's transaction information in the announcement table. In this case, all $n - 1$ threads will be assigned to complete the delayed thread's transaction. Since all threads are working towards completing the delayed thread's transaction, all operations in the transaction are guaranteed to be completed by some thread. If a conflict on a node is detected while attempting to perform an operation, the thread that detected the conflict will help complete the transaction associated with the node. While helping this conflicting transaction, a thread is still required to check the announcement table according to the progress assurance scheme in order to ensure wait-free progress in the presence of conflicts.

A Transactional Transformation Template

In this section, we will use the MDList as an example to introduce the methodology of transforming non-blocking data structures into transactional containers. A multi-dimensional list (MDList) partitions a linked list into shorter lists where each node contains multiple links to the child nodes arranged according to the dimension.

The INSERT and DELETE operation commute if they access different nodes. The transformed INSERT, detailed in Algorithm 4.6, checks if a node is present in the set on line 4.6.10. If the node does not exist in the set, then no conflict is detected and the INSERT operation of the base data structure is called on line 4.6.15. However, if the node does exist in the set, then a conflict is detected and UPDATEINFO is invoked on line 4.6.11 in order to finish the transaction associated

with the node and logically interpret if the key exists in the set. If the key does not logically exist in the set, the node's transaction information is updated to the NODEINFO of the calling thread and true is returned; otherwise, false is returned.

The transformed DELETE, detailed in Algorithm 4.8, checks if a node is present in the set on line 4.8.10. If the node does not exist in the set, then DELETE returns $fail$ on line 4.8.13. However, if the node does exist in the set, UPDATEINFO is invoked on line 4.8.11 in order to finish the transaction associated with the node and logically interpret if the key exists in the set. If the key logically exists in the set, the node's transaction information is updated to the NODEINFO of the calling thread and true is returned; otherwise, false is returned.

The transformed FIND, similar to the INSERT of Algorithm 4.6, checks if a node is present in the set.

The transformed FIND, in Algorithm 4.7, checks if a node is present in the set on line 4.7.10. If the node does not exist in the set, then FIND returns $fail$ on line 4.7.13. If the node exists in the set, UPDATEINFO is invoked on line 4.7.11 in order to finish the transaction associated with the node and logically interpret if the key exists in the set. Given that the key logically exists in the set, the node's transaction information is updated to the NODEINFO of the calling thread and true is returned; otherwise false is returned. The templates for the transformed INSERT, DELETE, and FIND are applicable to the MDList, dictionary, linked list, and skip list.

The dictionary, linked list, skip list, and binary search tree also provide the set operations INSERT, DELETE, and FIND. They all use the same templates for the transformed versions of each of those functions, as described above.

The base data structure of the dictionary is based on the node layout of the MDList, described above. The linked list in DTT is a lock-free linked list that was presented by Harris [20]. The

skip list was published by Fraser [14]. The tree is a non-blocking binary search tree proposed by Howley [33].

Algorithm 4.5: Progress Assurance Scheme

```
1  NodeInfo*[ ] announcementTable ← new NodeInfo*[THREAD_COUNT];
2  thread_local int threadId;
3  thread_local int delayCount ← 0;
4  thread_local int helpId ← 0;
5  thread_local int failures ← 0;
6  ;
7  Function CheckForAnnouncement()
8      if delayCount = HELP_DELAY then
9          delayCount ← 0;
10         NodeInfo* info ← announcementTable[helpId];
11         if info ≠ null then
12             HELPTRANSACTION(info.desc, info.opid + 1);
13         helpId ← (helpId + 1) mod THREAD_COUNT;
14     else
15         delayCount ← delayCount + 1;
16 ;
17 Function MakeAnnouncement(NodeInfo* info)
18     announcementTable[threadId] ← info;
19     HELPTRANSACTION(info.desc, info.opid + 1);
20     announcementTable[threadId] ← NULL;
21 ;
22 Function ResetFailures()
23     failures ← 0;
24 ;
25 Function HasReachedMaxFailures()
26     if failures = MAX_FAILURES then
27         return true
28     else
29         failures ← failures + 1;
30         return false
```

Algorithm 4.6: Template for Transformed Insert Function

```
1  Function Insert(int key, Desc* desc, int opid)
2      NodeInfo* info ← new NodeInfo;
3      info.desc ← desc, info.opid ← opid;
4      ResetFailures();
5      CheckForAnnouncement();
6      while true do
7          if HasReachedMaxFailures() then
8              MakeAnnouncement();
9          Node* curr ← Do_LocatePred(key);
10         if IsNodePresent(curr, key) then
11             ret ← UpdateInfo(curr, info, false)
12         else
13             Node* n ← new Node;
14             n.key ← key, n.info ← info;
15             ret ← Do_Insert(n)
16         if ret = success then
17             return true
18         else if ret = fail then
19             return false
```

Algorithm 4.7: Template for Transformed Find Function

```
1  Function Find(int key, Desc* desc, int opid)
2      NodeInfo* info ← new NodeInfo;
3      info.desc ← desc, info.opid ← opid;
4      ResetFailures();
5      CheckForAnnouncement();
6      while true do
7          if HasReachedMaxFailures() then
8              MakeAnnouncement();
9          Node* curr ← Do_LocatePred(key);
10         if IsNodePresent(curr, key) then
11             ret ← UpdateInfo(curr, info, true)
12         else
13             ret ← fail
14         if ret = success then
15             return true
16         else if ret = fail then
17             return false
```

Algorithm 4.8: Template for Transformed Delete Function

```
1  Function Delete(int key, Desc* desc, int opid)
2      NodeInfo* info ← new NodeInfo;
3      info.desc ← desc, info.opid ← opid;
4      ResetFailures();
5      CheckForAnnouncement();
6      while true do
7          if HasReachedMaxFailures() then
8              MakeAnnouncement();
9          Node* curr ← Do_LocatePred(key);
10         if IsNodePresent(curr, key) then
11             ret ← UpdateInfo(curr, info, true)
12         else
13             ret ← fail
14         if ret = success then
15             del ← curr;
16             return true
17         else if ret = fail then
18             del ← NIL;
19             return false
20 ;
21 Function MarkDelete(set delnodes, Desc* desc)
22     for del ∈ delnodes do
23         if del = NIL then
24             continue
25         NodeInfo* info ← del.info;
26         if info.desc ≠ desc then
27             continue
28         if CAS(del.info, info, SetMark(info)) then
29             Do_Delete(del)
```

CHAPTER 5: NON-BLOCKING TRANSACTIONAL HASH MAP

In this chapter, we demonstrate the application of our lock-free transactional transformation on hash maps. Map data structures store keys and their associated values. Maps also provide an update operation to change the value associated with a particular key, in addition to the insert, find, and remove operations that are present in set data structures. To support map data structures we add a VALUE field to the OPERATION and NODE structs present in Figure 2.1, and an UPDATE to the OPTYPE enumeration.

The reason we make these changes to Algorithm 2.1 is to provide two different places to save a node's value, one in the node itself, and one in the node's descriptor. We use these two locations to preserve the current value of a node, and buffer pending updates in the node's descriptor. This allows FIND operations from the same transaction to return the correct VALUE, held in the descriptor, if the transaction commits, without erroneously overwriting the value stored in the node by the most recently committed transaction. If we use the FIND operation from Algorithm 4.7, we will erroneously overwrite pending updates as the FIND operation will place its descriptor at a node, overwriting the node descriptor of the active UPDATE operation. To solve this problem, we note that the node descriptor of a FIND operation only needs to store the key that it is searching for, and its operation type. In this case, the pending update can be preserved by copying its value from the old node descriptor of the UPDATE operation to the node descriptor of the FIND operation which is now placed at the node. In this way, the pending writes to a node's value can be preserved without overwriting the node's current value, which would prevent inverse interpretation on transaction abort. This is an extension of logical status interpretation, as we choose a different VALUE depending on whether or not the transaction is ABORTED, ACTIVE, or COMMITTED.

Our addition of value fields to the OPERATION and NODE structs in Figure 2.1 allows us to perform

logical status interpretation on key-value pairs, as we will be able to recover the previous value associated with a key if an UPDATE operation is aborted. We propagate this change to ISKEYPRESENT by treating the UPDATE operation the same way we treat a FIND, as neither operation changes the presence of a key.

To perform logical status interpretation of a key-value pair we implement an ISVALUEPRESENT function as the UPDATE operation buffers writes to a node in the node's descriptor until it commits. The pseudocode for this function is presented in Algorithm 5.1. This function returns whether or not the value present in the NODE should be treated as present; if not, the value in the NODEINFO descriptor is used, because there is a pending update from the same transaction descriptor whose buffered write is logically interpreted as the node's value.

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

We copy the pending updates, which are buffered in a node's descriptor, to the node in a lazy fashion. Once the transaction with the pending updates is committed, the next FIND or UPDATE operation that attempts to update the NODEINFO descriptor of that node will examine that node descriptor in order to determine whether or not the last operation that was performed was a committed FIND or UPDATE. If it sees a FIND operation's descriptor that holds a different valid VALUE from the node's current value, then the MAPUPDATEINFO algorithm will update the node's value. If the operation sees an UPDATE descriptor at the node, with any value other than that currently stored in the node, then the operation copies the value stored in the descriptor to the node. Once

the operation completes the copy, it can perform its operation as usual. This copy preserves correct semantics for all operations, as the old VALUE will be ignored by an INSERT or DELETE operation, and a FIND or UPDATE operation uses the new value in the node's VALUE field. The pseudocode for the MAPUPDATEINFO algorithm is displayed in Algorithm 5.2. The lazy update of the node's value occurs in the if-then statement on line 5.2.17.

Algorithm 5.1: Logical Status for Maps

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

As with the transactional linked list and skiplist, we encapsulate the base data structure's methods for locating, inserting, and deleting nodes. We describe the templates for each of the four canonical map operations INSERT, DELETE, FIND, and UPDATE. The only change we make to the templates for the INSERT, DELETE, and FIND operations as shown in Algorithms 4.6, 4.8, and 4.7 is that

we must set the VALUE field of the new node in an INSERT to the value that is associated with the key. No other changes are necessary as these operations do not examine the value associated with a key. Additionally, the underlying DO_LOCATEPRED algorithm must search for a node using the hashed key instead of the key itself. The template for the UPDATE function is similar to that of the INSERT function (except we call MAPUPDATEINFO with TRUE as the final argument), since the MAPUPDATEINFO function lazily updates the VALUE of a node.

Algorithm 5.2: Update NodeInfo for Maps

```
1  Function MapUpdateInfo(NodeInfo* info, bool wantkey)
2      NodeInfo* oldinfo ← n.info;
3      if IsMarked(oldinfo) then
4          Do_Delete(n);
5          return retry
6      if oldinfo.desc ≠ info.desc then
7          ExecuteOps(oldinfo.desc, oldinfo.opid + 1)
8      else if oldinfo.desc, oldinfo.opid + 1 then
9          return success
10     bool haskey ← IsKeyPresent(oldinfo) ;
11     if (!haskey and wantkey) or (haskey and !wantkey) then
12         return fail
13     if info.desc.status ≠ Active then
14         return fail
15     Operationop ← info.desc.ops[info.opid];
16     Operation oldOp ← oldinfo.desc.ops[oldinfo.opid];
17     if op.type == Update || op.type == Find then
18         if oldOp.value ! = node.value && oldinfo.desc.status ==
             Committed && IsValuePresent(oldinfo) then
19             n.value ← oldOp.value;
20     if CAS(&n.info, oldinfo, info) then
21         if op.type == Find then
22             if oldOp.type == Update || (oldOp.type == Find && oldOp.value ! =
                 INVALID) then
23                 n.info.value ← oldOp.value;
24                 return n.info.value;
25             else
26                 return n.value;
27         return success
28     else
29         return retry
```

CHAPTER 6: NON-BLOCKING TRANSACTIONAL VECTOR

In this chapter, we describe the extension of the original LFTT algorithm for contiguous data structures, especially those with a single point of contention. The vector that we discuss is based on the design in [8]. As such, we use a two-level array to grow the capacity of the vector without needing to move elements during a resize operation.

LFTT resolves conflicts by detecting node-level conflicts. In the original LFTT paper this is all described using the KEY field of a node. To preserve semantic conflict detection in an array, vector, or another contiguous container that does not store key-value pairs it is sufficient to use any unique identifier, such as the index of the element.

To synchronize transactions in a data structure that has a single point of contention, we must add a global node to our transaction synchronization. A global node is identical to any other node with regard to its structure, however we use the global node to synchronize all pushBack and popBack operations. Any transaction that wants to perform a pushBack or popBack must place its transaction descriptor not only in the node's transaction descriptor pointer, but also in the global node's transaction descriptor pointer. The global node also stores a value that is equal to the pending size, that is the size the vector will be if the current transaction succeeds. By storing the pending size we allow multiple pushBack operations within the same transaction to complete successfully.

Listings 6.1 and 6.2 display the code for the read and write operations which perform bounds-checking before attempting their respective operations. After bounds-checking the traditional LFTT semantic conflict detection, and helping process are executed. Listings 6.3 and 6.4 show the pushBack and popBack operations that use our new global node to synchronize transactions that use these operations. After helping any pending transactions, the pushBack operation attempts

to compare-and-swap its transaction descriptor into the global node. If successful, the new node is created with a reference to the global node's transaction descriptor, so that no concurrent read or write operations can be executed on this node which has not yet been logically inserted. The global node's value is then updated by adding one to signify the new size if the transaction commits. Listing 6.5 presents the size function which reads from the global node's value if the transaction executing the size operation is the same as the pending transaction that has its transaction descriptor store in the global node. Otherwise, the size class variable is returned, which reflects the current size based only on committed transactions.

As we are building on LFTT, we need only to show that the vector operations are linearizable. See Appendix D for the proof of correctness.

```
inline ReturnFlag Read
(uint32_t pos, TxDesc* desc, TxInfo* info, uint8_t opid)
{
    while(true)
    {
        if(globalNode->info->desc == info->desc)
            size = globalNode->val;
        else
            size = this->size;


        if ( pos <= size )
        {
            Node* curr = at(pos);
            TxInfo* oldCurrInfo = curr->info;

            FinishPendingTxn(oldCurrInfo, desc);

            if(IsSameOperation(oldCurrInfo, info))
                return RET_SKIP;

            if(IsKeyExist(oldCurrInfo))
            {
                TxInfo* currInfo = curr->info;

                if(desc->status != ACTIVE)
                {
                    return RET_FAIL;
                }
                    currInfo = CAS(&curr->info, oldCurrInfo, info);
```

```
                    if(currInfo == oldCurrInfo)
                 return RET_OK;
            }
            else
                return RET_FAIL;
        }
        else
            return RET_FAIL;
    }
}
```

Listing 6.1: Transactional Vector Read

```
inline ReturnFlag Push_Back
(uint64_t key, TxDesc* desc, TxInfo* info, uint8_t opid)
{
    while(true)
    {
        TxInfo* oldCurrInfo = globalNode->info;

        FinishPendingTxn(globalNode->info, desc);

        if(IsSameOperation(globalNode->info, info))
            return RET_SKIP;

        if(desc->status != ACTIVE)
            return RET_FAIL;

        TxInfo* currInfo = globalNode->info;

            currInfo = CAS(&globalNode->info, oldCurrInfo, info);

            if(currInfo == oldCurrInfo)
            {

                Node* curr = at(globalNode->val);
                curr->info = info;
                curr->val = key;
                globalNode->val++;

                return RET_OK;
            }
    }
}
```

Listing 6.2: Transactional Vector Push_Back

```
inline ReturnFlag Write
(uint32_t pos, uint64_t val, TxDesc* desc, TxInfo* info, uint8_t opid)
```

```
{
    uint32_t size = 0;

    while(true)
    {
        if(globalNode->info->desc == info->desc)
            size = globalNode->val;
        else
            size = this->size;

        if ( pos <= size )
        {
            Node* curr = at(pos);
            TxInfo* oldCurrInfo = curr->info;

            FinishPendingTxn(oldCurrInfo, desc);

            if(IsSameOperation(oldCurrInfo, info))
                return RET_SKIP;

            if(IsKeyExist(oldCurrInfo))
            {
                TxInfo* currInfo = curr->info;

                if(desc->status != ACTIVE)
                    return RET_FAIL;

                    currInfo = CAS(&curr->info, oldCurrInfo, info);

                    if(currInfo == oldCurrInfo)
                        return RET_OK;
            }
            else
                return RET_FAIL;
        }
        else
            return RET_FAIL;
    }
}
```

Listing 6.3: Transactional Vector Write

```
inline ReturnFlag Pop_Back
(TxDesc* desc, TxInfo* info, uint8_t opid)
{
    while(true)
    {
        TxInfo* oldCurrInfo = globalNode->info;

        FinishPendingTxn(globalNode->info, desc);
```

```
        if(IsSameOperation(globalNode->info, info))
            return RET_SKIP;

        if(desc->status != ACTIVE)
            return RET_FAIL;

        TxInfo* currInfo = globalNode->info;

            currInfo = CAS(&globalNode->info, oldCurrInfo, info);

            if(currInfo == oldCurrInfo)
            {
                if (globalNode->val != 0)
                {
                    Node* curr = at(globalNode->val);
                    curr->info = info;
                    globalNode->val--;

                    UpdateSize(info, true);

                    return RET_OK;
                }
                else
                    return RET_FAIL;
            }

    }
}
```

Listing 6.4: Transactional Vector Pop_Back

```
inline ReturnFlag Size
(TxDesc* desc, TxInfo* info, uint8_t opid)
{
    uint32_t size = 0;

    if(globalNode->info->desc == info->desc)
        size = globalNode->val;
    else
        size = this->size;

    return RET_OK;
}
```

Listing 6.5: Transactional Vector Size

CHAPTER 7: PERFORMANCE EVALUATION

Wait-free Hash Map

We tested several algorithms against our wait-free implementation; we tested with two different values for $\mathrm{arrayLength}$, to show the space-time trade-off that this parameter represents. The values that we chose for the $\mathrm{arrayLength}$ were four (WaitFree-4) and six (WaitFree-6). As there are no other wait-free hash maps in the literature we chose the best available lock-free maps as well as a standard locking algorithm to test against. The locking solution that we include is the C++11 standard template library hash map protected by an optimized global lock (Lock-STL) [35]. The lock-free algorithms, from the literature, that we compare against are Split-Ordered Lists (Split-Ordered) [52] and Michael's lock-free hash map (Michael) [46]. We use the freely available implementations of Split-Ordered Lists and Michael's hash map that are provided by the Concurrent Data Structures library [36].

We also compare against two versions of Click's hash map. The first version is provided by him, and is written in Java (Click-Java) [5]. In order to avoid an unfair comparison by comparing C/C++ implementations to Java code, we include the second version which is provided by nbds (Click-C++) [10], and is written in C++. We also compare against Intel TBB's implementation (TBB) [34], because it is known to have high performance.

Careful attention has been paid to the comparability of the different implementations; for example, all tested data structures are able to accept different initial capacities. We only timed the operations of the hash map, avoiding any performance overhead of memory management and any overhead due to the testing itself. All data shown is the average of thirty runs, which were made to minimize the effects of any extraneous factors in the system. All tests were run on a SuperMicro server with

four sockets, each populated by a sixteen-core AMD Opteron 6272 processor at 2.1 GHz, and a total of 64 gigabytes of RAM. The machine was running 64-bit Ubuntu Linux version 11.04, and all code was compiled with g++4.7, with level three optimizations enabled. The testing variables for the graph presented in Figure 4 include creating a hash map that has an initial capacity of 2^{10} elements. This hash map was filled to its capacity and then we performed one million operations.

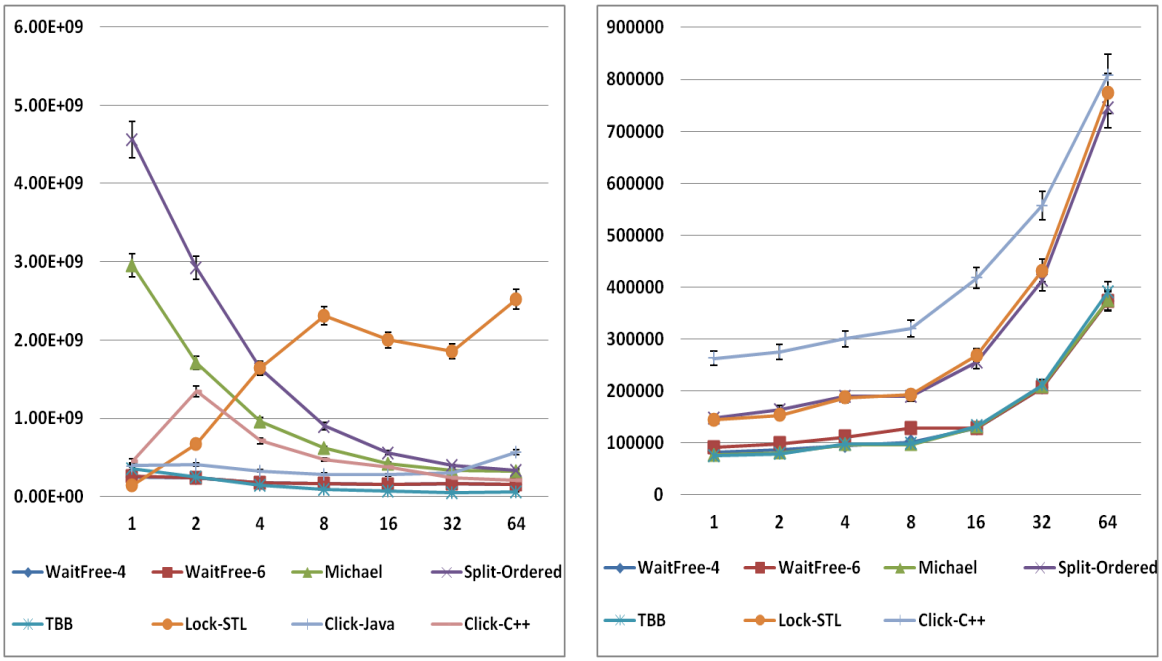
We divided our operations into three different kinds of distributions. The first type of distribution is based on a reported typical operation mix for hash maps [52]. This mix was reported without mention of an update function. We run the reported distribution, 88% get, 10% insert, 0% update 2% remove and a modified version that includes calls to update, 88% get, 8% insert, 2% update 2% remove. The second kind of distribution involves inverting the two versions of the aforementioned typical usage distribution within reason by moving the focus from the get operation to the insert and update operations; this yields the following operation mixes: 10% get, 88% insert, 0% update 2% remove; 10% get, 70% insert, 18% update 2% remove; and 10% get, 18% insert, 70% update 2% remove. The third distribution consists of a more even mix of operations. We have two of these distributions; one includes update: 25% get, 25% insert, 25% update 25% remove; one does not include update: 34% get, 33% insert, 0% update 33% remove.

The performance results in Figure 4 show that, on average, our wait-free algorithm outperforms the traditional blocking design by a factor of 7 or more, and it performs faster than the lock-free algorithms typically by a factor of 15. The lack of scalability of the blocking solution is a result of the fact that the lock is applied to all operations, not only those that conflict. Both lock-free solutions scale; however, they perform worse when more insert operations are performed, because the insert operations trigger more global resizes. Due to the incremental approach that we take to resizing the hash map, we see performance improvements over the other designs in the tested scenarios except for TBB. The other lock-free designs show an average of a 17.5 times performance decrease when compared to Intel's TBB implementation. In contrast, our approach is competitive

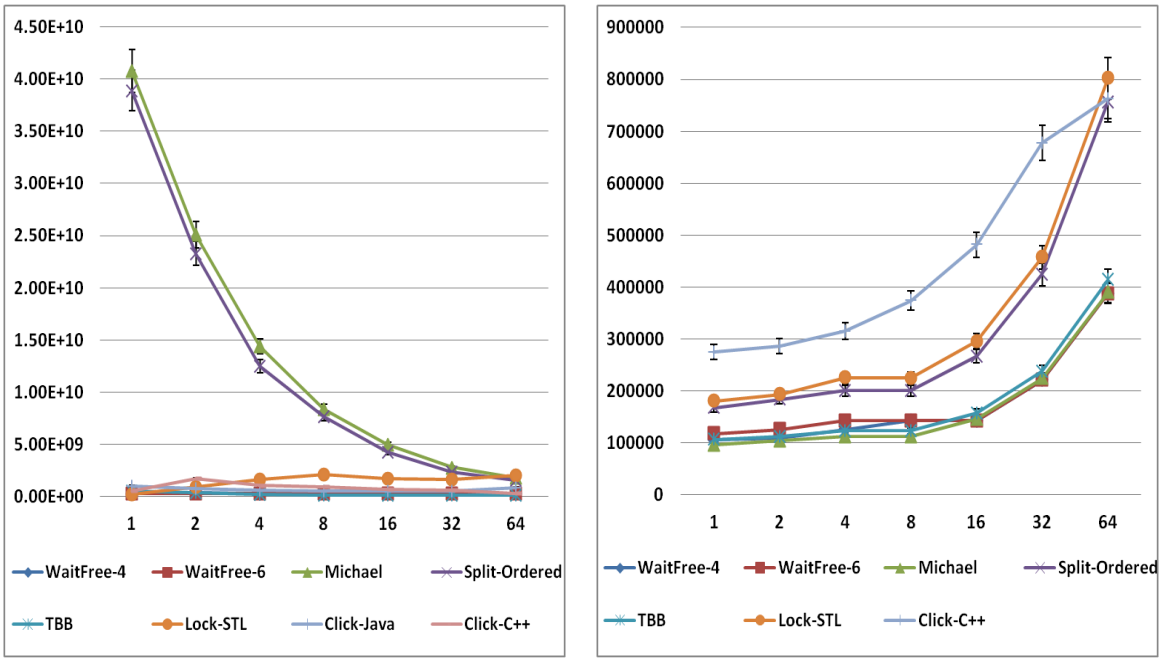
with only a 14% loss in performance to provide the stronger progress guarantee of wait-freedom.

On average, the lock-free algorithms use 1.8 times more memory than our algorithm, and the blocking approaches use 1.4 times more memory than our design. When we compare the two different configurations of our algorithm, we see that when we set the $\mathrm{arrayLength}$ to 6 we use 4% more memory, but complete the test runs 5% faster. In general, it is advisable to set the size of the main array equal to the ceiling of the binary logarithm of the expected number of elements; this allows the hash map to perform a minimal number of resizes, without using too much memory. The $\mathrm{arrayPow}$ determines how much space is added when a hash collision occurs; it should be set based on the expected number of hash collisions. The $\mathrm{maxFailCount}$ should be set to the expected number of threads that will compete for a single location in the hash map; in practice, the $\mathrm{failCount}$ never surpassed 3, but a value of 10 was used for testing. If $\mathrm{maxFailCount}$ is set too low, then the hash map may be unnecessarily expanded.

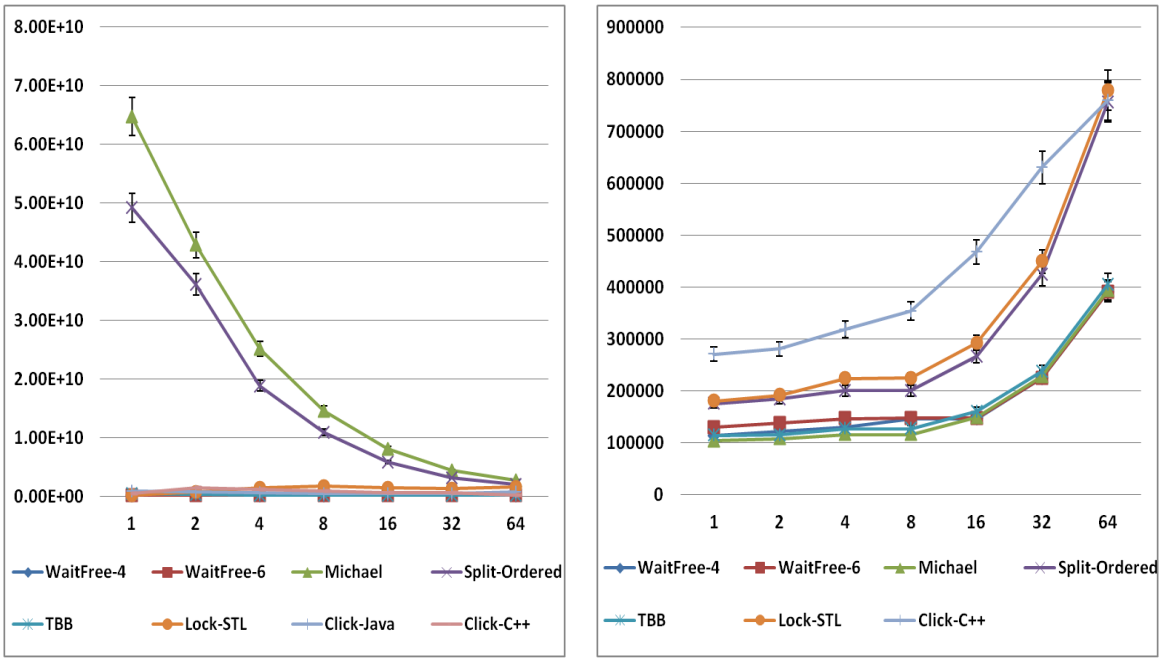
The following graphs show the average number of nanoseconds per thread that each operation took to execute the test versus the number of threads, and the average number of kilobytes per thread for each test. These graphs contain error bars which represent a 95% confidence interval for the results. The memory results for the Java version of Click's hash map were not able to be completely separated from the overhead of the virtual machine; so, these are not reported here.



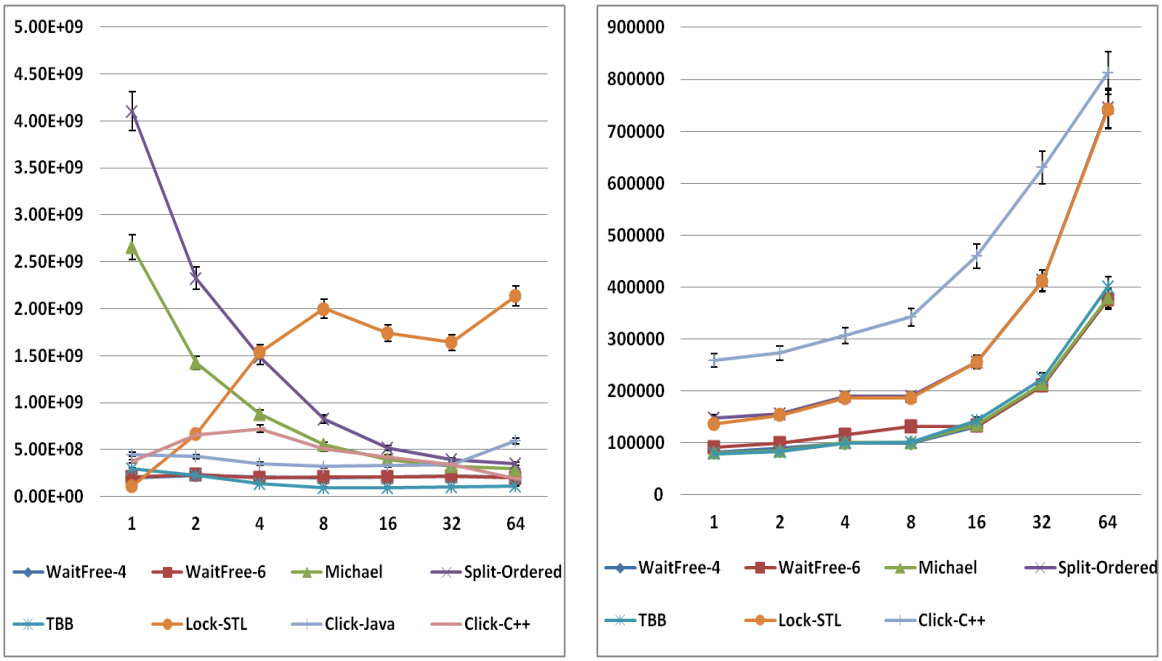
(a) 10% Get, 18% Insert, 70% Update, 2% Remove



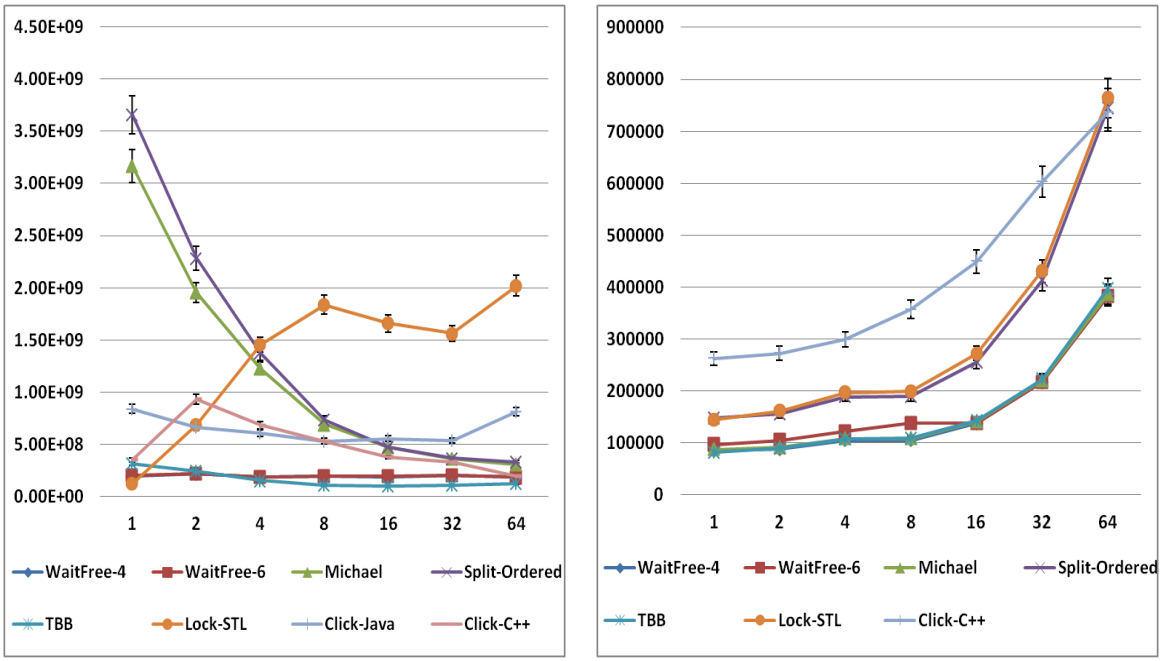
(b) 10% Get, 70% Insert, 18% Update, 2% Remove



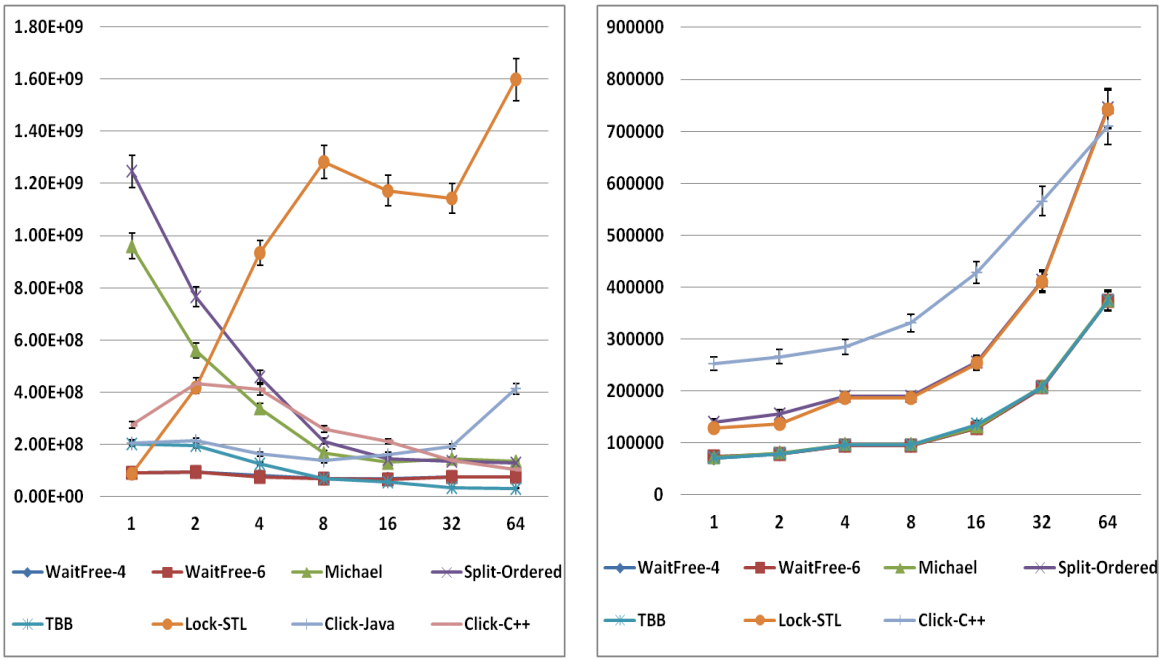
(c) 10% Get, 88% Insert, 0% Update, 2% Remove



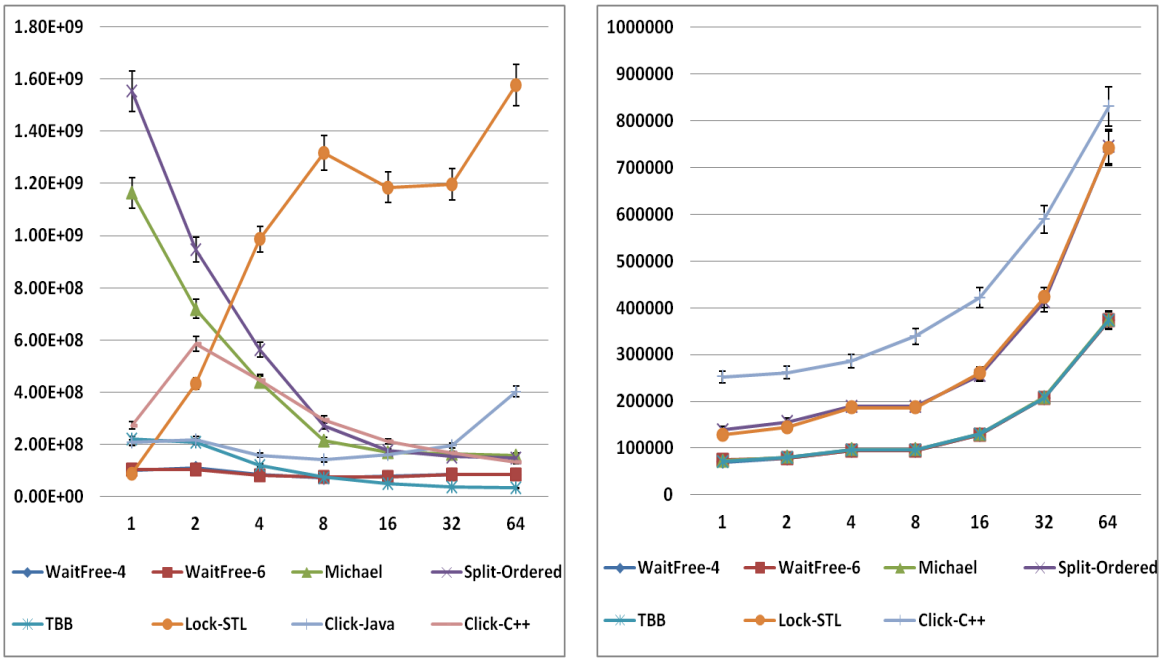
(d) 25% Get, 25% Insert, 25% Update, 25% Remove



(e) 34% Get, 33% Insert, 0% Update, 33% Remove



(f) 88% Get, 8% Insert, 2% Update, 2% Remove



(g) 88% Get, 10% Insert, 0% Update, 2% Remove

Figure 7.1: Hash Map Performance Results for Different Operation Mixes

Dynamic Transactions

We compare the containers in DTT with transactional boosting and STM versions. As word-based STM is the most commonly used approach to transactions, we perform our STM comparison using the Rochester STM package [45], which contains over one dozen STM implementations. Of the approaches in RSTM, NOrec STM [6] is the fastest implementation on our machine, and we use it for comparison with the list, MDList, dictionary, and binary search tree. We make an exception for the skip list, as Fraser provides an open-source implementation of the skip list that uses his own object-based STM implementation [14]. Because modern word-based STM

implementations inherently support dynamic transaction execution, we do not need to modify them for our performance evaluation.

We compare against the state-of-the-art transactional boosting approach. Transactional boosting is designed to be used with STMs for replaying undo logs; however, we scrap the STM environment as it is not necessary for our test case. The removed STM environment is replaced with a lightweight per-thread undo log. This replacement reduces the runtime overhead for a fair comparison. Like STM, the transactional boosting algorithm does not require any major modifications to support dynamic transaction execution.

We also compare against LFTT, which is the methodology on which DTT builds. LFTT does not include a wait-free progress assurance scheme, or a way to perform dynamic multi-container transactions. We show this comparison to demonstrate the low performance overhead of the progress assurance scheme and dyanmic transaction support. 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.

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 [6, 21], 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 [54]. The tests are conducted on a 64-core NUMA system (4 AMD Opteron 6272 CPUs with 16 cores per chip @ 2.1 GHz). The library of

data structure implementations (plus the micro-benchmark) is compiled with GCC 4.8 with C++11 features and O3 optimizations. [1]

In this section, we present the performance results for each container as graphs that include the throughput and number of spurious aborts. 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 [40]. The graphs for the write-dominated scenario is 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, DTT as DTT, transactional boosting as BST, and software transactional memory as STM for all performance graphs.

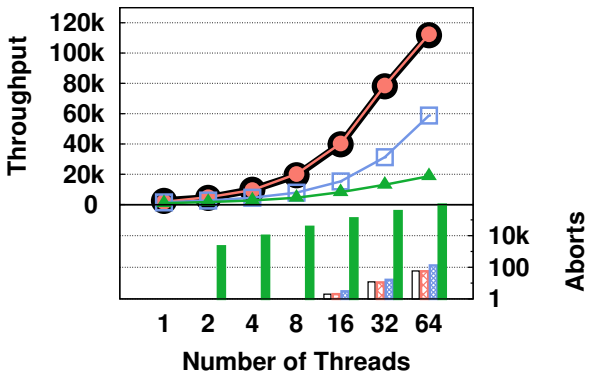
The upper portion of the figures represents the throughput with the x-axis in logarithmic scale and the y-axis in linear scale. The key for all of the performance graphs is the same, and can be found in Figure 7.1. The bottom half of all figures represents the histogram of spurious aborts, with the x- and y-axes in logarithmic scale. The key for the lower plot, of aborted transactions, is shown on the right half of Figure 7.1.

[1] All source code will be made available upon publication at https://github.com/ucf-cs/tlds

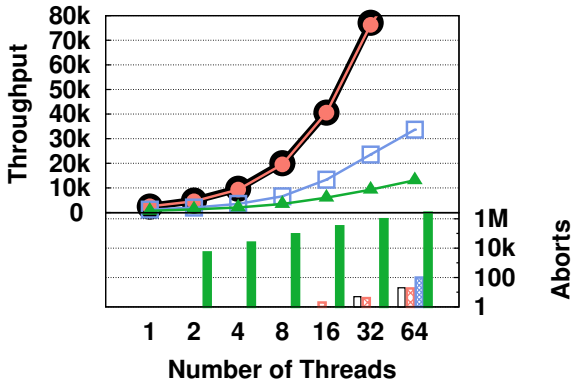
Overall Results

Across all data structure evaluations, DTT outperforms BST by an average of 118%, STM by an average of 203%, and LFT by an average of 0.767%. DTT gains an advantage over BST and STM because of its semantic conflict detection and logical interpretation, which allows it to avoid the costs of excessive aborts and physical rollbacks. The reason that DTT achieves the same performance as LFT is 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 in the announcement table, which is rarely observed in practice [40]. As a result, threads rarely need to pause their own operations to help other threads. Also, DTT allows threads to avoid the cost of duplicate work by utilizing a return values list. Therefore, our library provides dynamic transaction execution, wait-free progress, and multi-container transactions at no additional cost.

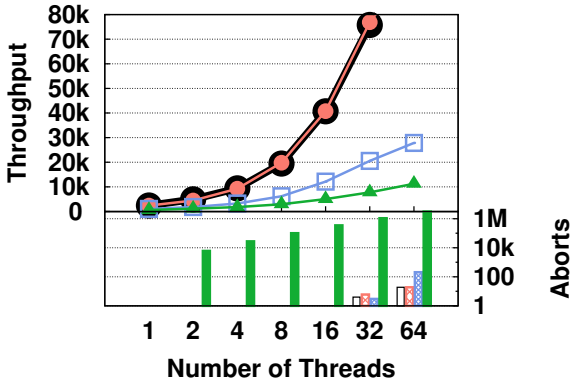
Figure 7.1: Key for Performance Graphs



(a) 15%INSERT, 5%DELETE, 80%FIND

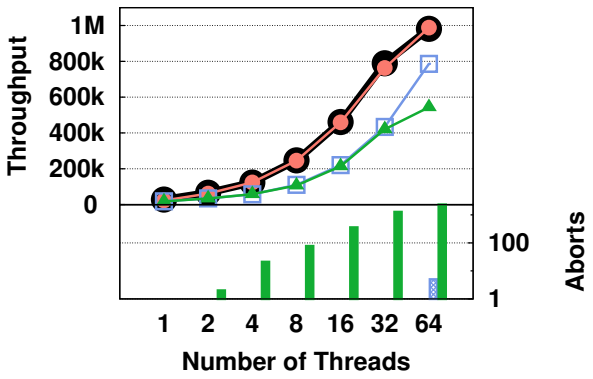


(b) 33%INSERT, 33%DELETE, 34%FIND

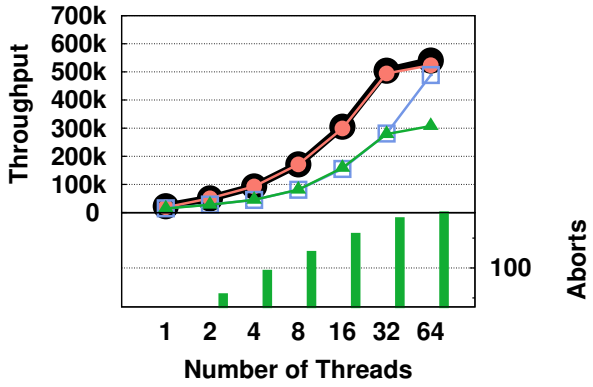


(c) 50%INSERT, 50%DELETE, 0%FIND

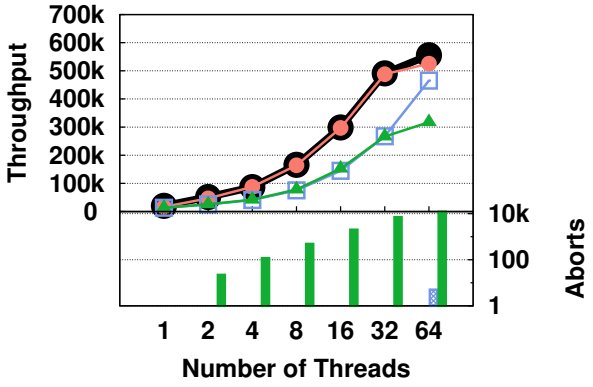
Figure 7.2: Transactional List Performance



(a) 15%INSERT, 5%DELETE, 80%FIND



(b) 33%INSERT, 33%DELETE, 34%FIND



(c) 50%INSERT, 50%DELETE, 0%FIND

Figure 7.3: Transactional Skip List Performance

Transactional List

We compare the throughput of four different implementations of transactional linked lists in Figure 7.2. The base data structure used by all of the implementations is the lock-free list by Harris [20]. 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 168% and STM by an average of 459% 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 and STM encounter a conflict, they abort one of the conflicting transactions, decreasing the overall throughput. On the other hand, DTT avoids most of these spurious aborts because threads help each other to complete each other's transactions, allowing both transactions to commit. This phenomenon can be observed in the number of spurious aborts shown in the bottom half of each graph in Figure 7.2. For example, in the case of 64 threads, BST experiences 3 times more spurious aborts than DTT and LFT, and STM experiences four orders of magnitude more spurious aborts.

STM's throughput particularly suffers when the number of threads increases, 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 subsequent aborts.

DTT outperforms LFT by 1.93% while also providing the benefits of dynamic transaction execution and wait-free progress. The overhead of DTT is low because it rarely needs to activate its wait-free progress assurance scheme. Also, 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.

Transactional Skip List

We compare the throughput of four different types of transactional skip lists in Figure 7.3. The implementations are based on the skip list presented by Fraser [14]. 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 $80,000$ transactions 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 82.1% and STM by an average of 90.9%, while performing 4.53% faster than LFT. As with the transactional linked list, the DTT version of the skiplist experiences low overhead on top of LFT.

Transactional MDList

Figure 7.4 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. [60]. Like the skip list, the MDList has logarithmic search time, so we perform the evaluation with a key range of $1,000,000$.

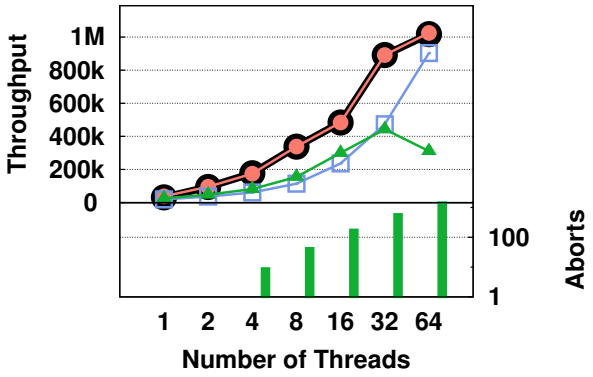
The results are similar to those concerning the transactional skip list in Section 7. In overall throughput, DTT is on par with LFT, performing 0.398% faster, and it outperforms BST by an average of 110% and STM by an average of 149%.

A noteworthy difference can be found between the throughput of the STM skip list and STM MDList. The throughput of the STM skip list increases with the number of threads. Conversely, the STM MDList's throughput increases until 16 threads and then decreases significantly. This

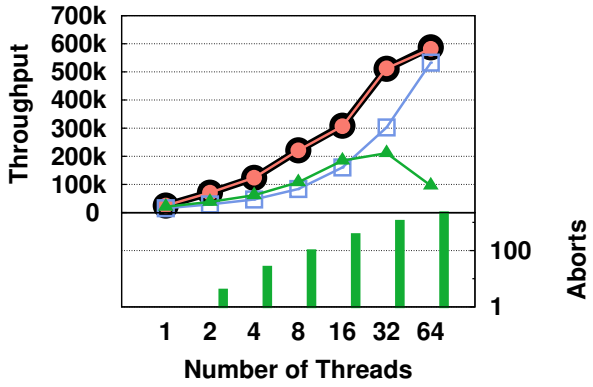
phenomenon can be attributed to a combination of factors: the MDList's unique method of node insertion, STM's use of memory barriers, and the cost of inter-processor communication between remote cores in the NUMA system. Each node in an MDList has several child nodes. When an MDList inserts a node, some cases require the new node to "adopt" its successor node's children. Since this process takes some time, 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 to read during the traversal. For each of these reads, STM uses a memory barrier to prevent incorrect instruction re-orderings. 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.

Transactional Dictionary

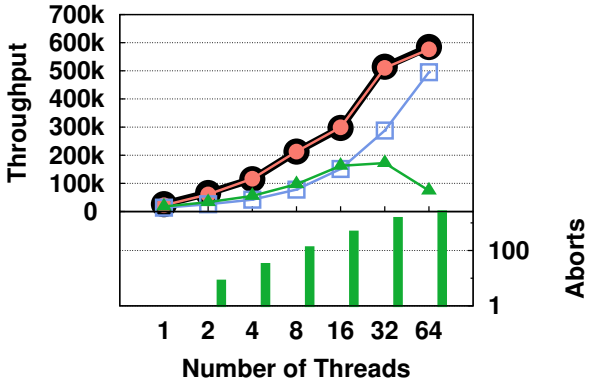
The graphs of the performance of the transactional dictionaries are omitted, because they are the same as those for the transactional MDLists in Section 7. The dictionary has the same memory layout and similar underlying code as the transactional MDList, with the addition of a value parameter attached to the insert and find operations.



(a) 15%INSERT, 5%DELETE, 80%FIND

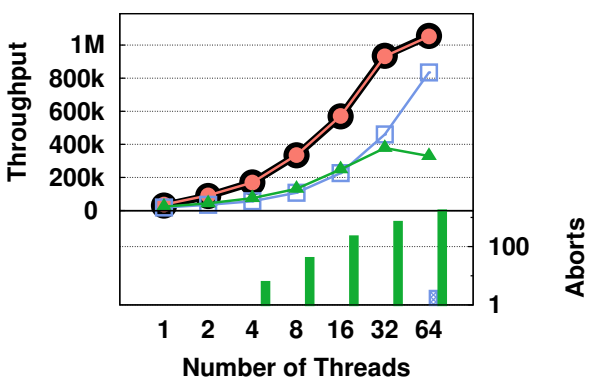


(b) 33%INSERT, 33%DELETE, 34%FIND

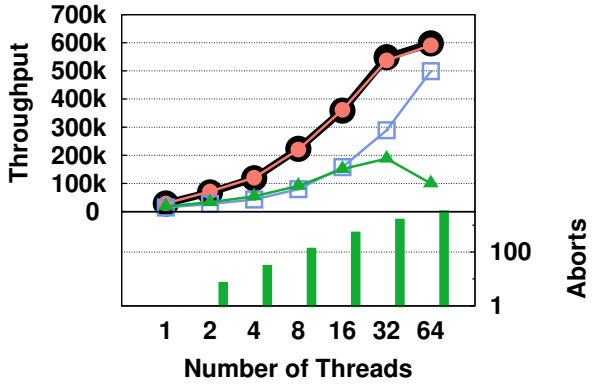


(c) 50%INSERT, 50%DELETE, 0%FIND

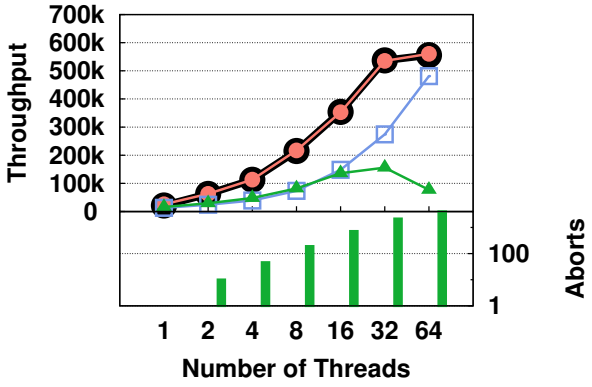
Figure 7.4: Transactional MDList Performance



(a) 15%INSERT, 5%DELETE, 80%FIND



(b) 33%INSERT, 33%DELETE, 34%FIND



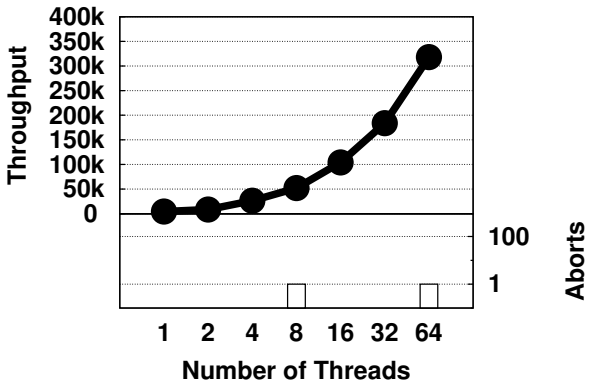
(c) 50%INSERT, 50%DELETE, 0%FIND

Figure 7.5: Transactional Binary Search Tree Performance

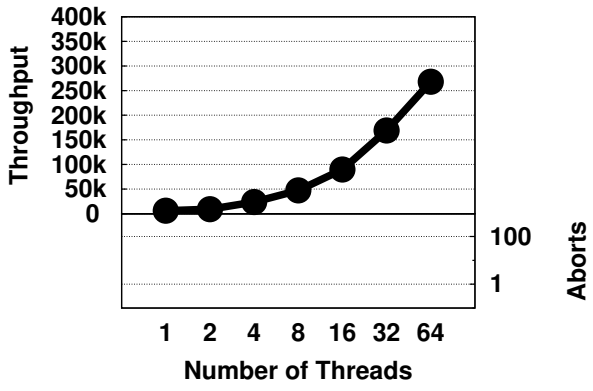
Transactional Binary Search Tree

Figure 7.5 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 [33]. Because the binary search tree provides logarithmic search time, we perform the evaluation with a key range of $1,000,000$.

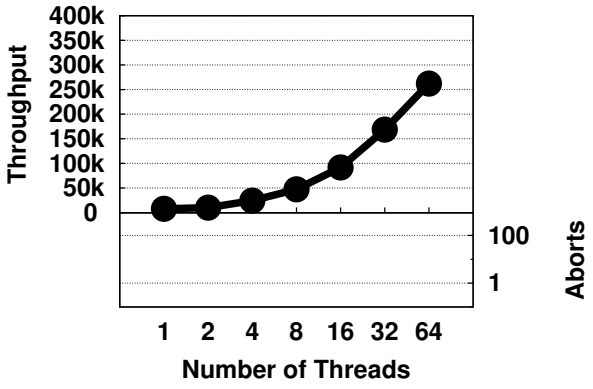
The performance results of the transactional binary search trees resemble those of the transactional MDLists in Section 7. In overall throughput, DTT performs 1.98% slower than LFT and outperforms BST by an average of 124% and STM by an average of 173%.



(a) 15%INSERT, 5%DELETE, 80%FIND



(b) 33%INSERT, 33%DELETE, 34%FIND



(c) 50%INSERT, 50%DELETE, 0%FIND

Figure 7.6: Wait-free Multi-Container Performance

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, 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, and disabled as LF for the remainder of this section. 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 7.7 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 impact of the progress assurance scheme on the performance of the data structure increases as the key range is reduced. 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 invoke more often.

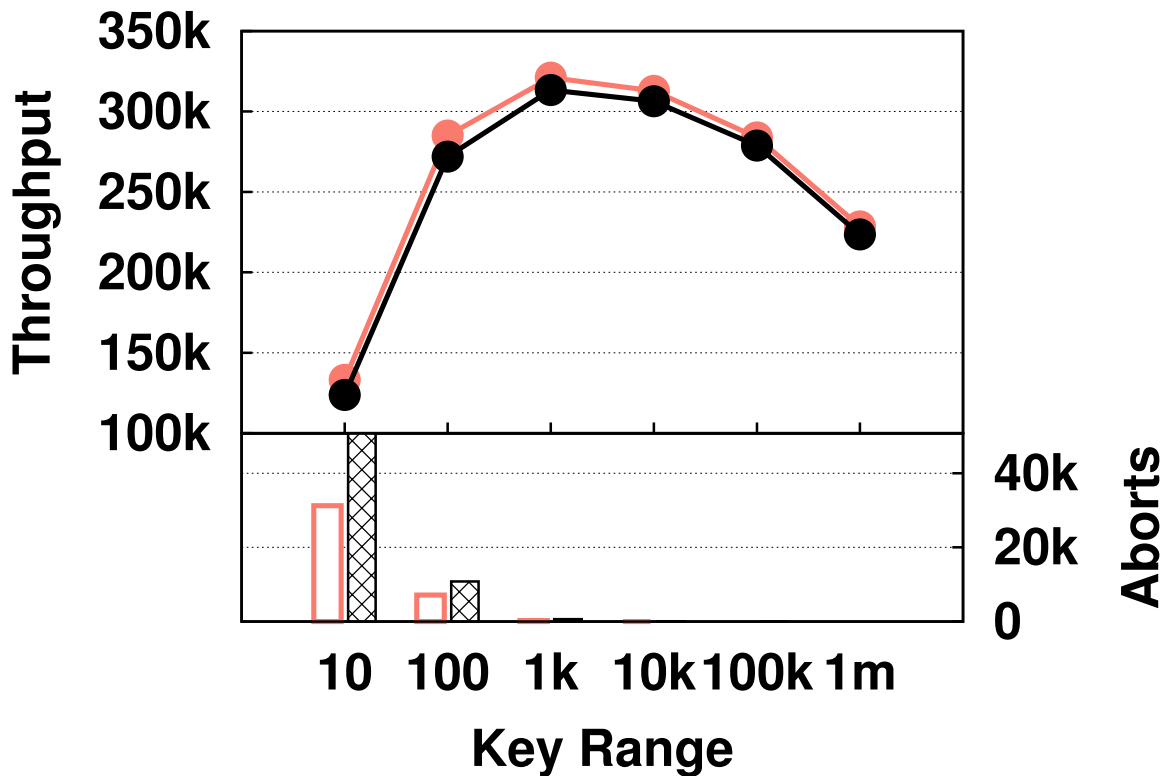


Figure 7.7: Wait-free Progress Assurance Scheme Overhead

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 con-

flicts 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.

One interesting result we encounter is that for a key range of 10, the ratio of spurious aborts to commits for WF is extremely high compared to LF (53.3% versus 13.7%), yet WF still has a similar number of commits per second to LF, reaching 94.3% of the throughput of LF. If WF processes transactions at the same speed as LF but has more spurious aborts, then we would expect that WF would have a lower number of commits. On the contrary, what we encounter is that although a greater percentage of transactions are aborting, this is mostly counteracted by the fact that all transactions are being processed more quickly. This surprising increase in transaction processing speed could be attributed to the following: that in this case of high contention, having threads work together on transactions reduces contention and therefore offsets some of the costs of the progress assurance scheme.

Transactions Among Multiple Data Structures

We perform experimental evaluations on transactions that span multiple containers, and the performance results are shown in Figure 7.6. Our experiments include a transactional linked list, skip list, MDList, dictionary, and binary search tree. We use one instance of each container type in this evaluation, although multiple instances of each container type can be used. Each thread performs transactions for 10 seconds with a key range of $10,000$. The evaluation method on multiple containers is similar to the evaluation method on a single container, but each operation in a transaction

is randomly chosen to be executed on one of the five containers. We only present the DTT results as DTT is the only methodology to support transactions that span multiple data structures.

Transactional Hash Map

We compare the overhead and scalability of our lock-free transactional list and skiplist against the implementations based on transaction boosting, NOrec STM from Rochester Software Transactional Memory package [45] and Fraser's lock-free object-based STM [14]. RSTM is the best available comprehensive suite of prevailing STM implementations. In our test, TML [6] and its extension NOrec [6] are among the fastest on our platform. They have extremely low overhead and good scalability due to elimination of ownership records. We choose NOrec as the representative implementation because its value-based validation allows for more concurrency for readers with no actual conflict.

For transaction boosting, we implement the lookup of abstract locks using Intel TBB's concurrent hash map. Although the transaction boosting is designed to be used in tandem with STMs for replaying undo logs, it is not necessary in our test case as the data structures are tested in isolation. To reduce the runtime overhead, we scrap the STM environment and implement a lightweight per-thread undo log for the boosted data structures. We employ a micro-benchmark to evaluate performance in three types of workloads: write dominated, read dominated, and mixed. This canonical evaluation method [6, 21] consists of a tight loop that randomly chooses to perform a fixed size transaction with a mixture of INSERT, DELETE and FIND operations according to the workload type. We also vary the transaction size (i.e., the number of operations in a transaction) from 1 to 16 to measure the performance impact of rollbacks. The tests are conducted on a 64-core NUMA system (4 AMD opteron 6272 CPUs with 16 cores per chip @2.1 GHz). Both the micro-benchmark and the data structure implementations are compiled with GCC 4.7 with C++11

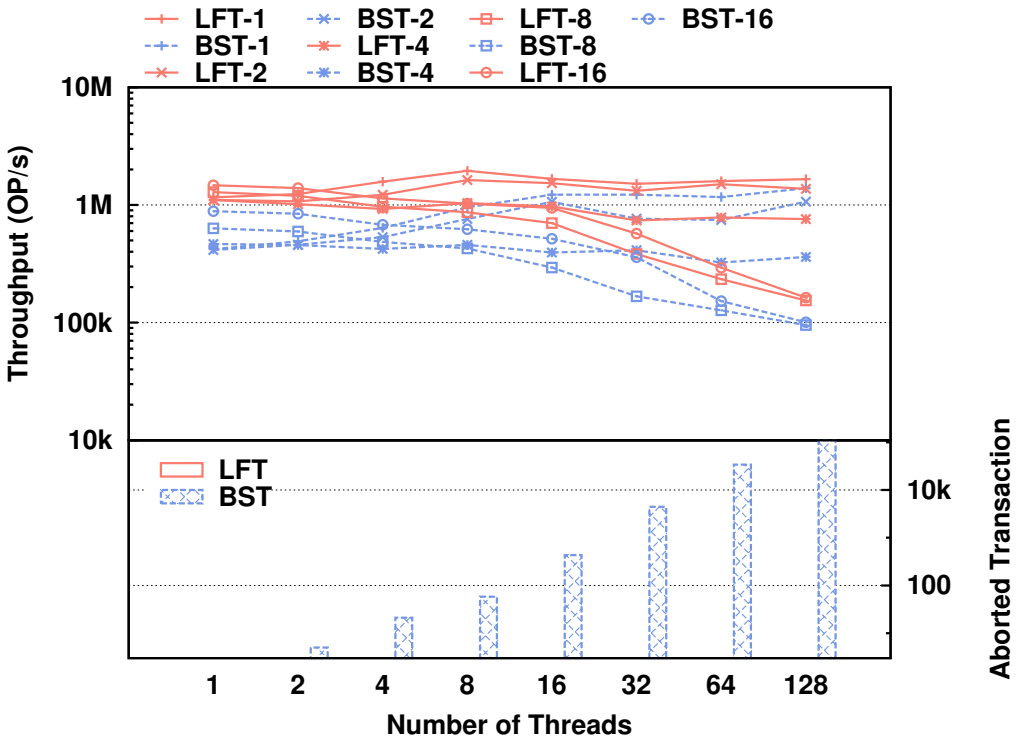
features and O3 optimizations. [2]

Transactional Map

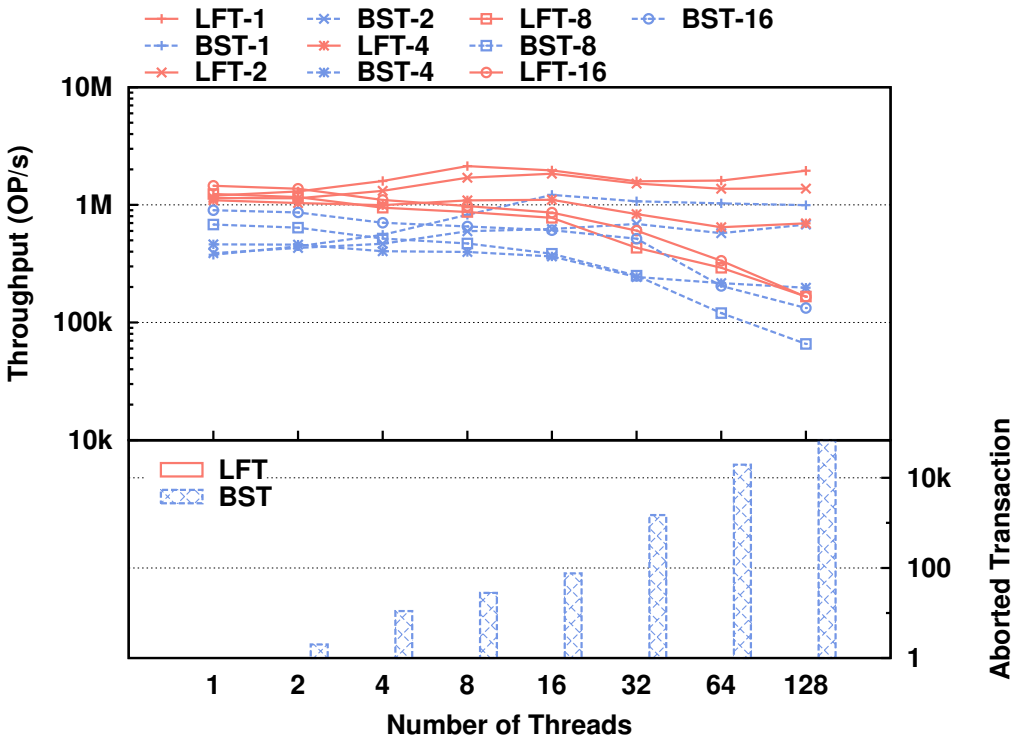
In Figure 7.7, we show the throughput for the lock-free transactional hash map. The LFTT map, and the transaction boosting version that we compare to, are based on the wait-free hash map in [40].

In order to test the LFTT map with a large workload, we apply the same evaluation procedure as in Section 7, giving each thread a workload of 1 million transactions and setting the key range to 1 million. As we can see in Figure 7.8a, large transactions such as LFT-8 and LFT-16 achieve maximum throughput on a single thread, then their throughput steadily falls as the number of threads increases. This is the same behavior observed in Figure 7.3c. The trend is weaker for the graph shown in Figure 7.7c, because the 75% FIND operations leads to a greater number of operations in the larger transaction sizes committing without failing. An example of this is shown for the transaction size of 8 in Figure 7.7c, which seems to level out compared to the other two operation distributions with the same transaction size. The transactional boosting version of the hash map follows the same trends, in these cases, but has lower performance due to executing transactions which must be rolled back when they abort. In comparison, our approach, does not suffer from any spurious aborts in any of the tested scenarios. Overall, with a peak throughput of more than 2.6 million (OP/s), transaction execution on our hash map is considerably more efficient than on linked lists, and comparable to skiplists despite the extra overhead on the UPDATEINFO function due to the UPDATE operation. On average, the LFTT hash map is 74% faster than the transactional boosting version.

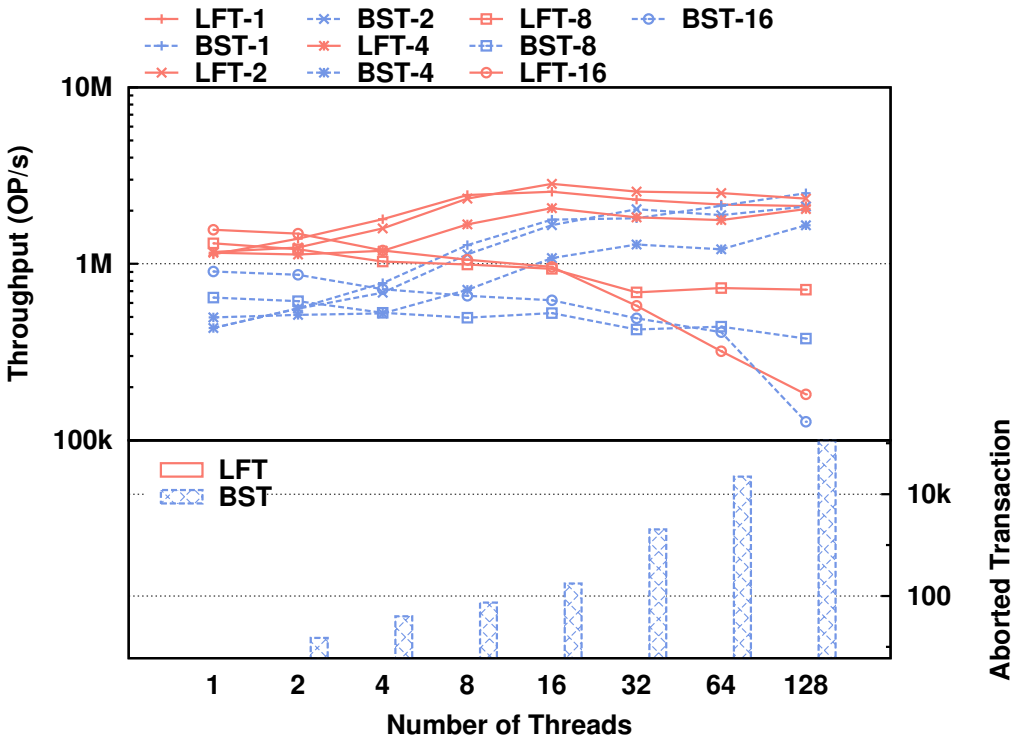
[2] All source code can be downloaded from https://github.com/ucf-cs/tlds



(a) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 50% INSERT, 50% DELETE, 0% UPDATE, 0% FIND



(b) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 25% INSERT, 25% DELETE, 25% UPDATE, 25% FIND



(c) Throughput for the Lock-free Transactional Hash Map (1M Key Range): 15% INSERT, 5% DELETE, 5% UPDATE, 75% FIND

Figure 7.7: Throughput for the Lock-free Transactional Hash Map (1M Key Range)

CHAPTER 8: CONCLUSION

In this dissertation, we present dynamic transactional data structures that provide the wait-free progress guarantee.

We present a wait-free hash map that is not transactional, this approach demonstrates non-blocking programming techniques, and the drawbacks of traditional non-blocking data structures. Our design outperforms state of the art non-blocking designs, and standard blocking approaches by 15 and 7 times, respectively.

We discuss an extension of a lock-free transactional transformation methodology (LFTT), that has been applied to the wait-free hash map. We adapt LFTT to support map data structures, and use this extended version to implement a transactional hash map.

We introduce the first methodology that provides dynamic lock-free data structure transactions, DTT. This design allows dynamic wait-free transactions to be executed on multiple containers within a single transaction. Our experimental results demonstrate that our performance is at least on par with state of the art approaches, and in all but one case surpasses them. We apply our extended version of DTT to create a library of five wait-free transactional data structures. With this library, a developer can write transactional programs without knowledge of the underlying algorithms for wait-free progress or transaction synchronization.

All of the above has been or will be released as open source software, so that the maximum impact can be made in industry and academia.

Future Work

An idea for future work is to update DTT, so that it is no longer necessary to separate transactional code into transactional functions; this would increase the usability of DTT.

APPENDIX A: CORRECTNESS OF THE WAIT-FREE HASH MAP

In this section we outline a correctness proof. For brevity, we give informal proofs; these follow the style in [46]. Several useful definitions follow. Abbreviations of the form $\mathrm{U11}$ are used; the letter is the first letter of the corresponding operation e.g. U11 refers to the eleventh line of the update algorithm pseudocode.

(1) For all times t, a node is in the hash map at t, if and only if at t it is reachable by following pointers starting from the head.

(2) For all times t, the state of the hash map is represented as $S_{n,m,p}$ where n, m, and p are defined as follows.

   (a) n : the number of $\mathrm{dataNodes}$ in the hash map at t.

   (b) m : the number of $\mathrm{markedDataNodes}$ in the hash map at t.

   (c) a : the number of $\mathrm{arrayNodes}$ in the hash map at t (this excludes the main array).

For example, the hash map is in state $S_{2,1,0}$ if it contains exactly two $\mathrm{dataNodes}$, one $\mathrm{markedDataNode}$, and zero $\mathrm{arrayNodes}$.

Lemma 1. *The hashed key of a* $\mathrm{dataNode}$ *never changes while it is in the hash map.*

Lemma 2. *A* $\mathrm{markedDataNode}$ *is not unmarked until the corresponding expansion has occurred.*

Lemma 3. *An* $\mathrm{arrayNode}$ *is never removed from the hash map.*

Safety

To prove safety, we attempt to prove Claim 1.

The hash map is in a valid state, if and only if it matches the definition of some state $S_{n,m,a}$ that is reachable, through the specified transitions, from the initial state $S_{0,0,0}$. The state of the map

changes upon the successful execution of any of the following lines: $\mathrm{markDataNode}$ line 2, I13, R37, or E10 (see Section 3). In Figure A.1, these lines are abbreviated as follows: $\mathrm{markDataNode}$ line 2 which marks a node becomes M, I13 which inserts a $\mathrm{dataNode}$ becomes I, R37 which removes a node becomes R, and E10 which unmarks a $\mathrm{markedDataNode}$ and adds a new $\mathrm{arrayNode}$ becomes N. Transitions that occur on the execution of $\mathrm{markDataNode}$ line 2 from $S_{1,1,0}$ and $S_{2,1,0}$ have been omitted for clarity.

Claim 1. *All transitions are consistent with the hash map's semantics. If the hash map is in a valid state, then if a CAS succeeds a correct transition occurs, as shown in the state transition diagram in Figure A.1.*

In the case of a successful update operation the state triple does not change; however, the set of all $\mathrm{dataNodes}$ that exist in the map is changed (see Section 3). Specifically, a $\mathrm{dataNode}$ is atomically removed from the set and replaced by a $\mathrm{dataNode}$ with the same key but a different associated value, this occurs at line U38.

We prove Claim 1 by induction. In the basis step, we assume that the hash map is in the valid, initial state $S_{0,0,0}$. We take Claim 1 to be the induction hypothesis. In the inductive step, we show that, at any time t, the application of any transition on a valid state yields a valid state.

Lemma 4. *If successful, the atomic* OR *operation in line* $\mathrm{I11}$ *takes the hash map to a valid state, and marks a* $\mathrm{dataNode}$*.*

Lemma 5. *If successful, the CAS on line* $\mathrm{I13}$ *takes the hash map to a valid state, and inserts a* $\mathrm{dataNode}$ *into the set.*

Lemma 6. *If successful, the CAS on line* $\mathrm{U38}$ *does not change the state, and updates the value associated with a key.*

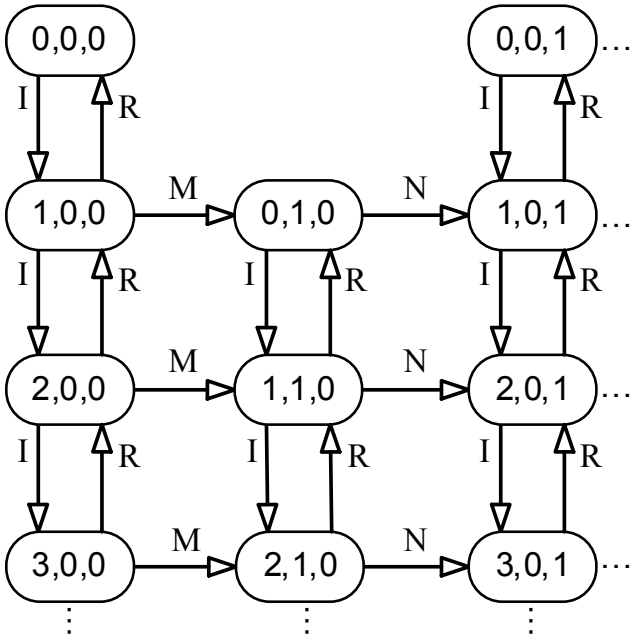


Figure A.1: A state transition diagram for the hash map.

Lemma 7. *If successful, the CAS on line* R37 *takes the hash map to a valid state, and removes a* dataNode *from the set.*

Lemma 8. *If successful, the CAS on line* E10 *takes the hash map to a valid state and replaces a* markedDataNode *with an* arrayNode *that contains an unmarked version of the* markedDataNode.

Theorem 1. *Claim 1 is true at all times.*

Linearizability

Our hash map is linearizable, because all of its operations have linearization points (see Section 3 for details).

The linearization points below are presented for each operation, when executed concurrently with any other operation of the hash map. If there is no concurrent execution, then linearizability is not applicable, because the definition of a linearization point is meaningless when defined on a single operation. In the case of a single operation, that of sequential execution, correctness of the algorithms becomes much easier to prove; such proofs are omitted. The linearization points of the supporting algorithms are trivial to prove. Due to the composability of linearizability we do not need to further consider the supporting functions. See Section 3 for a discussion of the linearizability of the memory management functions.

Lemma 9. *Every* get *operation takes effect upon its read on line* G06.

Lemma 10. *Every* update *and* remove *operation that returns* true *takes effect upon its CAS on lines* U38 *and* R37*, respectively.*

Lemma 11. *Every* update *and* remove *operation that returns* false *takes effect when a* dataNode *with a different key is encountered during traversal (see Section 3).*

Lemma 12. *Every* insert *operation that returns true takes effect upon its CAS on line* I13.

Lemma 13. *Every* insert *operation that returns false takes effect upon its CAS on line* I13*, its atomic read on line* I08*, or its atomic read at line* I23.

Given the derived linearization points, we are able to provide a valid sequential history from every concurrent execution of the hash map's operations; this proves Theorem 2.

Theorem 2. *The hash map's operations are linearizable.*

Wait-Freedom

To prove wait-freedom we must show that every call to insert, update, get, and remove returns in a bounded number of steps [37]. This is trivial to prove for the get, update, and remove operations as they are bounded by a for-loop, that runs at most $maxDepth$ times, and the progress of these operations is unhindered by the side effects of any combination of concurrent operations. To prove wait-freedom for insert we need to show that the number of operations that may linearize before a particular insert operation is bounded [37].

We need only consider those insert operations that act on the same position in the hash map, as disjoint operations may proceed in parallel without issue. Furthermore, operations that attempt to insert the same key at the same position at the same time do not break the wait-free progress guarantee, because one operation will complete the CAS successfully, and the others will fail and will not retry. However, when concurrent insert operations with different keys attempt to work on the same position they would retry infinitely if it were not for $\mathrm{maxFailCount}$ (see Section 3), which is an upper bound on the number of times that the insert operations would conflict before an expansion occurred at that position. In the worst-case, the expansions would be performed until $\mathrm{maxDepth}$ was reached, with $\mathrm{maxFailCount}$ attempts at expansion being needed every time.

All of these operations complete in a finite number of steps; this is expressed in Theorem 3. Theorem 4 follows directly from Theorems 1, 2, and 3.

Lemma 14. *The* insert *operation completes in a number of steps that is bounded by* $maxDepth * maxFailCount$.

Lemma 15. *The* update *operation completes in a number of steps that is bounded by* $\mathrm{maxDepth}$.

Lemma 16. *The* get *operation completes in a number of steps that is bounded by* $\mathrm{maxDepth}$.

Lemma 17. *The* remove *operation completes in a number of steps that is bounded by* maxDepth.

Theorem 3. *All operations of the algorithm are* $\in O(1)$*, in the worst case.*

Theorem 4. *The algorithm is wait-free.*

APPENDIX B: CORRECTNESS OF DYNAMIC TRANSACTIONS

Correctness

DTT guarantees that any arbitrary history of committed transactions is *strictly serializable*, which is a correctness property that is the analogue of linearizability [32] for transactions. Our proof of correctness is based on the notion of commutativity isolation [26], which states that the history of committed transactions is *strictly serializable* for any transactional data structure that obeys the rules of linearizability, commutativity isolation, compensating actions, and disposable methods. We first define the rules of commutativity isolation, then we prove that DTT follows these rules and therefore provides the guarantee of strict serializability.

Definitions

We provide the definitions and correctness rules from Herlihy and Koskinen's work [26] that are necessary for our proof of strict serializability. A *history* of computation is a sequence of instantaneous events. Events associated with a method call include invocation I and response R. A single transaction running in isolation defines a *sequential history*. A *sequential specification* for a data structure defines a set of *legal histories* for that data structure.

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

Definition 2. *Two method calls I, R and I', R' commute if: for all histories h, if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same abstract state.*

Commutativity is a property on operations that have no dependencies on each other such that the execution of commutative operations in any order yields the same abstract state. Let O_A be

transactional container A and let O_B be transactional container B. The commutativity specification for set operations among multiple containers is as follows:

$$\begin{aligned}
\text{O}_A.\textsc{Insert}(x) &\leftrightarrow \text{O}_A.\textsc{Insert}(y),\ x \neq y \\
\text{O}_A.\textsc{Insert}(x) &\leftrightarrow \text{O}_B.\textsc{Insert}(x),\ A \neq B \\
\text{O}_A.\textsc{Delete}(x) &\leftrightarrow \text{O}_A.\textsc{Delete}(y),\ x \neq y \\
\text{O}_A.\textsc{Delete}(x) &\leftrightarrow \text{O}_B.\textsc{Delete}(x),\ A \neq B \\
\text{O}_A.\textsc{Insert}(x) &\leftrightarrow \text{O}_A.\textsc{Delete}(y),\ x \neq y \\
\text{O}_A.\textsc{Insert}(x) &\leftrightarrow \text{O}_B.\textsc{Delete}(x),\ A \neq B \\
\text{O}_A.\textsc{Find}(x) &\leftrightarrow \text{O}_A.\textsc{Insert}(x)/false \leftrightarrow \\
&\qquad \text{O}_A.\textsc{Delete}(x)/false \\
\text{O}_A.\textsc{Find}(x) &\leftrightarrow \text{O}_B.\textsc{Insert}(x),\ A \neq B \\
\text{O}_A.\textsc{Find}(x) &\leftrightarrow \text{O}_B.\textsc{Delete}(x),\ A \neq B
\end{aligned} \tag{B.1}$$

Definition 3. *For a history h and any given invocation I and response R, let I^{-1} and R^{-1} be the inverse invocation and response. That is, the invocation and response such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reached after history h.*

Definition 4. *For a history h, let G be the set of histories g such that $h \cdot g$ is legal. A method call denoted $I \cdot R$ is disposable if, $\forall g \in G$, if $h \cdot I \cdot R$ and $g \cdot I \cdot R$ are legal, then $h \cdot I \cdot R \cdot g$ and $h \cdot g \cdot I \cdot R$ are legal and both define the same state.*

The method call $I \cdot R$ is disposable if it can be delayed arbitrarily long, and the abstract state of the system is the same as the case in which $I \cdot R$ had occurred.

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

Rule 2. ***Commutativity Isolation:*** *Let T_1 and T_2 be transactions. For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, either T_1 commits or aborts before any additional method calls in T_2 are invoked, or vice-versa.*

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

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

Serializability and Recoverability

We now show that DTT meets the four above correctness requirements in order to show that any arbitrary history of transactions is strictly serializable. We denote the concrete state of a set as an node set N. At any time, the abstract state observed by transaction T_i is $S_i = \{n.key \mid n \in N \wedge \text{IsKeyPresent}(n.info,\ desc_i)\}$, where $desc_i$ is the descriptor of T_i.

Linearizability is the correctness property such that concurrent operations appear as if they took place instantaneously at some points between their invocations and responses. We reason about linearizability by identifying *linearization points* in which the transformed operations take effect. We use the notion of decision points and state-read points to facilitate our reasoning. The decision point of an operation is defined as the atomic statement that finitely decides the result of an operation, i.e. independent of the result of any subsequent instruction after that point. A state-read point is defined as the atomic statement where the state of the data structure, which determines the outcome of the decision point, is read.

Lemma 1. *The set operations* INSERT*,* DELETE*, and* FIND *are linearizable, satisfying Rule 1.*

Proof. The DELETE operation will check if a node exists in a transactional container, as shown on line 4.8.10. If the node does not exist, then DELETE will return $fail$ on line 4.8.13. If the node does exist, then the descriptor for the node is attempted to be updated to the current node descriptor by UPDATEINFO on line 4.8.11. The state-read point for this case is when $info.desc.status$ is read on line 2.3.13. The abstract states S' observed by all transactions immediately after the reads are unchanged, i.e., $\forall i, S'_i = S_i$. The decision point for a successful logical status update occurs when the CAS operation on line 15 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after the CAS is $i = d \implies S'_i = S_i - n.key$. For all other transactions $i \neq d \implies S'_i = S_i$. In all cases, the update of abstract states conforms to the sequential specification of the DELETE operation. After a successful logical status update, then the node to be deleted is stored in del on line 4.8.15, which will be inserted in the $delnodes$ set. If the transaction status is updated by CAS from $Active$ to $Committed$, then the $delnodes$ set is marked for deletion. Once the node descriptor is marked by CAS on line 4.8.28, the node will be physically deleted by DO_DELETE on line 4.8.29. The UPDATEINFO operation will physically delete a node on line 2.3.4 if the node descriptor is marked. The code path for physically deleting a node is linearizable because the corresponding DO_DELETE operation in the base data structure is linearizable.

The same reasoning process applies to the transformed INSERT and FIND operations because they share the same logical status update procedure with DELETE. □

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

Lemma 2. *Conflict detection in DTT satisfies the commutativity isolation rule as defined in Rule 2.*

Proof. As identified in Equation B.1, INSERT and DELETE commute if they access different keys

or operate on different transactional containers. Because of the one-to-one mapping from node to keys, we have $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. Therefore, INSERT and DELETE commute if they access two different nodes. Let T_1 denotes a transaction that currently accesses node n_1, i.e., $n_1.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 were to access n_1, it would invoke UPDATEINFO, and therefore perform EXECUTEOPS for T_1 on line 2.3.7. EXECUTEOPS always updates the transaction status by CAS because a failed CAS indicates that some other thread updated the transaction status. Therefore, it is guaranteed that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. Since it is guaranteed that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds, DTT satisfies the commutativity isolation rule. □

Lemma 3. *The logical rollback mechanism of DTT handles aborts equivalently to performing the inverses of completed transactions, satisfying Rule 3.*

Proof. Let T denote a transaction that executes the operations $I_0 \cdot R_0 \cdots I_i \cdot R_i$ on nodes $n_0 \cdots n_i$ and then aborts. Let S_0 denote the abstract state of the data structure immediately before I_0, and S_i denote the abstract state immediately after R_i. The requirement of Rule 3 is that after T aborts, it must execute the inverses of the successful method calls, namely $I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1}$. This requirement is equivalent to the abstract state being restored to S_0, the original state of the data structure.

We prove that for a data structure generated by DTT, when a transaction T aborts, then the following is true for each node n_x in $n_0 \cdots n_i$: the next operation (whether INSERT, DELETE, or FIND) that accesses n_x will logically interpret the current abstract state S_y to be equal to S_0.

INSERT method. For an INSERT($n_x.key$) call, there are two cases. In the first case where $n_x.key \notin S_0$, the INSERT method call either places a new node n_x into the data structure (line 4.6.15), or INSERT changes the existing node's transaction descriptor field (line 4.6.11). Either way, after

R_x, n_x has its transaction descriptor field pointing to T. Assume that T aborts after R_x, resulting in T's transaction descriptor status being set to $Aborted$. The next operation that accesses n_x will follow the transaction descriptor field of n_x, observe T's descriptor status as $Aborted$, and logically interpret that $n_x.key \notin S_y$. Therefore, $S_y = S_0$.

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

DELETE method. For a DELETE($n_x.key$) call, there are two cases. In the first case where $n_x.key \in S_0$, the DELETE method call changes the transaction descriptor field of the existing node n_x to point to T (line 4.8.11). Assume that the T aborts after R_x, resulting in T's transaction descriptor status being set to $Aborted$. The next operation that accesses n_x will follow the transaction descriptor field of n_x, observe T's descriptor status as $Aborted$, and logically interpret that $n_x.key \in S_y$. Therefore, $S_y = S_0$.

In the second case where $n_x.key \notin S_0$, the DELETE method does not perform any writes to the data structure, so this case is similar to the INSERT case in which $n_x.key \in S_0$.

FIND method. For a FIND($n_x.key$) call, there are two cases. In the first case where $n_x.key \in S_0$, the FIND method call changes the transaction descriptor field of the existing node n_x to point to T (line 4.8.11). Whether or not T aborts, the next operation that accesses n_x will always logically interpret that $n_x.key \in S_y$. Therefore, $S_y = S_0$.

In the second case where $n_x.key \notin S_0$, the FIND method does not perform any writes to the data structure, so this case is similar to the DELETE case in which $n_x.key \in S_0$. □

Lemma 4. *The* MARKDELETE *method is the only method call that can be invoked by a transaction*

T after T has committed or aborted. The MARKDELETE *method is disposable, so DTT satisfies Rule 4.*

Proof. We prove that the MARKDELETE method is disposable because it does not change the abstract state of the data structure. This implies that it is disposable since it can be postponed arbitrarily without anyone being able to tell that it did not occur.

We now prove that the MARKDELETE method does not change the data structure's abstract state. MARKDELETE can only be called on a node n if it has been invoked by a transaction T that called DELETE(n) and then committed. Then after T commits, $n.key \notin S$, where S is the abstract state of the data structure. The node n is still linked in the data structure, but its key is not present in the abstract state. MARKDELETE(n) causes n to no longer be linked in the data structure. This step is handled by the underlying DO_DELETE method of the base data structure. Because n is no longer linked in the data structure, $n.key \notin S$. Therefore, MARKDELETE does not change the abstract state, so it is disposable. □

Theorem 1. *The history of committed transactions is strictly serializable for a data structure constructed by DTT.*

Proof. Following Lemmas 1, 2, 3, 4, and the main theorem in Herlihy and Koskinen's work [26], the theorem holds. □

Theorem 2. *For a data structure generated by DTT, the complete history of transactions (including committed and aborted transactions) is strictly serializable.*

Proof. According to the Theorem of Aborted Transactions in Herlihy and Koskinen's work, if a system that obeys the four rules discussed in this section, then any history defines the same abstract state as a history with aborted transactions removed. Following Theorem 1, and the Theorem of

Aborted Transactions, then a data structure constructed by DTT guarantees that any history defines the same abstract state as a strictly serializable history. Therefore, any such history is strictly serializable. □

Progress Guarantees

The wait-free progress assurance scheme is based on the approach by Kogan et al. [38]. In order to provide wait-free progress for all threads in the system, we guarantee that each operation will be completed in a finite number of steps, and that the helping mechanism employed for the node-based conflict detection will detect a cyclic dependency in a finite number of steps. Lemma 5 gives an upper bound on the time to complete a single operation. Lemma 6 gives an upper bound on time to detect a cyclic dependency.

Lemma 5. *Let F be the number of steps to complete an operation in a lock-free manner. Let D be the delay to help a thread complete a pending operation. Let n be the total number of threads in the system. Let k be the maximum number of operations in a transaction. The number of steps to complete a single operation in a transaction is bounded by $O(F + (D + k) \cdot n^2 + k)$.*

Proof. We first show the time complexity for a delayed operation to be completed. An arbitrary thread t_i that fails to complete an operation will post its transaction information in the announcement table. A thread t_j will perform D operations prior to checking the announcement table and updating its HELPID to the next thread to be helped. When t_j begins to help another thread, t_j will start the thread's transaction at the current operation id. Thread t_j will perform at most k operations for each thread that it helps prior to reaching t_i. Since there are n threads, it will take $O((D + k) \cdot n)$ time for t_j to begin to help t_i. In a worst case, all threads will be required to help t_i. The time complexity for all threads to reach t_i is $O((D + k) \cdot n^2)$.

We now show the time complexity for a single operation. An arbitrary thread t_i that starts an operation will initially check the announcement table to determine if a thread t_j needs help with a pending operation. If a pending operation exists, then t_i will help t_j complete its entire transaction. In a worse case, it will take $O((D + k) \cdot n^2)$ time to complete t_j's operation. In this scenario, all threads will now be assigned to t_j's transaction. It will therefore take at most k steps to complete the transaction, yielding a time complexity of $O((D + k) \cdot n^2 + k)$. After t_i completes t_j's transaction, it may begin its own operation. Thread t_i will attempt to complete an operation in its transaction in a lock-free manner within F number of steps. If t_i fails to complete the operation, it will post the transaction information in the announcement table, which will take at most $O((D + k) \cdot n^2)$ steps to complete the operation. The total time complexity to complete a single operation is $O(F + (D + k) \cdot n^2 + k)$. □

Lemma 6. *Let n be the total number of threads in the system. Let k be the maximum number of operations in a transaction. The time complexity to detect a cyclic dependency is $O(n \cdot k)$.*

Proof. In the presence of conflicts, threads may set out to help each other during the execution of each of the operations. The number of recursive helping invocations is bound by the number of active transactions. For a system with n threads, the upper bound of the number of active transactions is n. Prior to helping a transaction, a thread will push the transaction descriptor onto the thread local help stack. Since the maximum transaction length is k, a cyclic dependency will be detected due to a duplicate descriptor in the help stack in at most $O(n \cdot k)$ steps. □

APPENDIX C: CORRECTNESS OF THE TRANSACTIONAL HASH MAP

Proof. As identified in Equation B.1, two set operations commute if they access different keys. Because of the one-to-one mapping from node to keys, we have $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. This means that two set operations commute if they access two different nodes. Let T_1 denote a transaction that currently accesses node n_1, i.e., $n_1.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 were to access n_1, it must perform EXECUTEOPS for T_1 on line 2.3.7 because EXECUTEOPS always updates the transaction status when it returns on line 4.3.21 or 4.3.24 (note that failed CAS also means the transaction status has been set, by another thread). We thus ensure that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. □

APPENDIX D: CORRECTNESS OF THE TRANSACTIONAL VECTOR

Lemma 7. *The vector operations* PUSH_BACK, POP_BACK, READ, *and* WRITE *are linearizable.*

Proof. The PUSH_BACK operation will add a new element to the end of a transactional container. The descriptor for the node is attempted to be updated to the current node descriptor by UPDATE-INFO. The state-read point for this case is when $info.desc.status$ is read on line 2.3.13. The abstract states S' observed by all transactions immediately after the reads are unchanged, i.e., $\forall i, S'_i = S_i$. The decision point for a successful logical status update occurs when the CAS operation on line 15 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after the CAS is $i = d \implies S'_i = S_i \cup n.key$. For all other transactions $i \neq d \implies S'_i = S_i$. In all cases, the update of abstract states conforms to the sequential specification of the PUSH_BACK operation. The code path for the physical insertion of a node is linearizable because the corresponding DO_PUSH_BACK operation in the base data structure is linearizable.

The same reasoning process applies to the transformed POP_BACK, READ, and WRITE operations because they share the same logical status update procedure with PUSH_BACK. □

LIST OF REFERENCES

[1] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.

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

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

[4] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.

[5] Cliff Click. A lock-free hash table (http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf). Retrieved 12/12/2012.

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

[7] D. Dechev. The ABA Problem in Multicore Data Structures with Collaborating Operations. In *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2011)*, 2011.

[8] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In M. Shvartsman, editor, *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer Berlin / Heidelberg, 2006.

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

[10] J. Dybnis. nbds. https://code.google.com/p/nbds/, October 2014.

[11] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

[12] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4:315–344, September 1979.

[13] S. Feldman, C. Valera-Leon, and D. Dechev. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667, March 2016.

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

[15] K. Fraser. Practical lock-freedom. In *Computer Laboratory, Cambridge Univ*, 2004.

[16] H. Gao, J. F. Groote, and W. H. Hesselink. Almost wait-free resizable hashtable. In *IPDPS*, 2004.

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

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

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

[20] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, 2001. Springer- Verlag.

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

[22] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

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

[24] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[25] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

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

[27] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

[30] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, New York, NY, USA, 2008.

[31] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

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

[33] S. V. Howley and J. Jones. A non-blocking internal binary search tree. *Spaa*, page 161, 2012.

[34] Intel Corporation. Reference for Intel Threading Building Blocks (http://threadingbuildingblocks.org/). Retrieved 08/25/2015.

[35] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages: C++*. American National Standards Institute, September 2011.

[36] M. Khiszinsky. Concurrent data structures. http://libcds.sourceforge.net/, May 2013.

[37] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[38] A. Kogan and E. Petrank. 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, pages 141–150, New York, NY, USA, 2012. ACM.

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

[40] P. LaBorde, S. Feldman, and D. Dechev. A wait-free hash map. *International Journal of Parallel Programming*, pages 1–28, 2015.

[41] P. LaBorde, L. Lebanoff, C. Peterson, D. Zhang, and D. Dechev. Wait-free dynamic transactions without rollbacks for linked containers. Technical report, University of Central Florida, 2018.

[42] P. Larson. Dynamic hashing. *BIT Numerical Mathematics*, 18:184–201, 1978. 10.1007/BF01931695.

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

[44] Lockless Inc. Technical specifications for the lockless inc. memory allocator. http://locklessinc.com/technical_allocator.shtml, December 2011.

[45] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on*

Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), 2006.

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

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

[48] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[49] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[50] Microsoft. System.collections.concurrent namespace. http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx, 2011. .NET Framework 4.

[51] M. Moir and N. Shavit. *Handbook of Data Structures and Applications*, chapter Concurrent Data Structures, pages 47–1–47–30. Chapman and Hall/CRC Press, 2007.

[52] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111, New York, NY, USA, 2003. ACM Press.

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

[54] A. Spiegelman, G. Golan-Gueta, and I. Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 682–696. ACM, 2016.

[55] H. Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 24b, april 2005.

[56] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68:1008–1020, July 2008.

[57] I. Walulya and P. Tsigas. Scalable lock-free vector with combining. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 917–926, May 2017.

[58] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2015.

[59] D. Zhang and D. Dechev. An efficient lock-free logarithmic search data structure based on multi-dimensional list. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 281–292, June 2016.

[60] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626, March 2016.

[61] D. Zhang and D. 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.

[62] D. Zhang, P. Laborde, L. Lebanoff, and D. Dechev. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC)*, 2018.