

Electronic Theses and Dissertations, 2004-2019

2016

High-Performance Composable Transactional Data Structures

Deli Zhang
University of Central Florida

 Part of the [Computer Sciences Commons](#)
Find similar works at: <https://stars.library.ucf.edu/etd>
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Zhang, Deli, "High-Performance Composable Transactional Data Structures" (2016). *Electronic Theses and Dissertations, 2004-2019*. 5069.
<https://stars.library.ucf.edu/etd/5069>

HIGH-PERFORMANCE COMPOSABLE TRANSACTIONAL DATA STRUCTURES

by

DELI ZHANG

M.S. University of Central Florida, 2016

B.S. Huazhong University of Science and Technology, 2008

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, Florida

Summer Term
2016

Major Professor: Damian Dechev

© 2016 Deli Zhang

ABSTRACT

Exploiting the parallelism in multiprocessor systems is a major challenge in the post “power wall” era. Programming for multicore demands a change in the way we design and use fundamental data structures. Concurrent data structures allow scalable and thread-safe accesses to shared data. They provide operations that appear to take effect atomically when invoked individually. A main obstacle to the practical use of concurrent data structures is their inability to support composable operations, i.e., to execute multiple operations atomically in a transactional manner. The problem stems from the inability of concurrent data structure to ensure atomicity of transactions composed from operations on a single or multiple data structures instances. This greatly hinders software reuse because users can only invoke data structure operations in a limited number of ways.

Existing solutions, such as software transactional memory (STM) and transactional boosting, manage transaction synchronization in an external layer separated from the data structure’s own thread-level concurrency control. Although this reduces programming effort, it leads to significant overhead associated with additional synchronization and the need to rollback aborted transactions. In this dissertation, I address the practicality and efficiency concerns by designing, implementing, and evaluating high-performance transactional data structures that facilitate the development of future highly concurrent software systems.

Firstly, I present two methodologies for implementing high-performance transactional data structures based on existing concurrent data structures using either lock-based or lock-free synchronizations. For lock-based data structures, the idea is to treat data accessed by multiple operations as resources. The challenge is for each thread to acquire exclusive access to desired resources while preventing deadlock or starvation. Existing locking strategies, like two-phase locking and resource hierarchy, suffer from performance degradation under heavy contention, while lacking a desirable

fairness guarantee. To overcome these issues, I introduce a scalable lock algorithm for shared-memory multiprocessors addressing the resource allocation problem. It is the first multi-resource lock algorithm that guarantees the strongest first-in, first-out (FIFO) fairness. For lock-free data structures, I present a methodology for transforming them into high-performance lock-free transactional data structures without revamping the data structures' original synchronization design. My approach leverages the semantic knowledge of the data structure to eliminate the overhead of false conflicts and rollbacks.

Secondly, I apply the proposed methodologies and present a suite of novel transactional search data structures in the form of an open source library. This is interesting not only because the fundamental importance of search data structures in computer science and their wide use in real world programs, but also because it demonstrates the implementation issues that arise when using the methodologies I have developed. This library is not only a compilation of a large number of fundamental data structures for multiprocessor applications, but also a framework for enabling composable transactions, and moreover, an infrastructure for continuous integration of new data structures. By taking such a top-down approach, I am able to identify and consider the interplay of data structure interface operations as a whole, which allows for scrutinizing their commutativity rules and hence opens up possibilities for design optimizations.

Lastly, I evaluate the throughput of the proposed data structures using transactions with randomly generated operations on two different hardware systems. To ensure the strongest possible competition, I chose the best performing alternatives from state-of-the-art locking protocols and transactional memory systems in the literature. The results show that it is straightforward to build efficient transactional data structures when using my multi-resource lock as a drop-in replacement for transactional boosted data structures. Furthermore, this work shows that it is possible to build efficient lock-free transactional data structures with all perceived benefits of lock-freedom and with performance far better than generic transactional memory systems.

To my parents and wife for their unwavering support over the years.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Damian Dechev for his guidance and support along the way.

I would like to also thank my dissertation committee members Dr. Gary Leavens, Dr. Cliff Zou, Dr. Mingjie Lin, and Dr. Kien Hua for their time and effort.

TABLE OF CONTENTS

LIST OF FIGURES	xiii
LIST OF TABLES	xv
LIST OF ALGORITHMS	xvi
CHAPTER 1: INTRODUCTION	1
Motivation	1
Contribution	5
Pseudo-Code Convention	6
Outline	7
CHAPTER 2: BACKGROUND	9
Terminology	9
Atomic Primitives	9
Correctness Property	10
Non-blocking Progress Assurance	11
Mutual Exclusion	11

Resource Allocation	12
Related Work	12
Lock-based Concurrency Control	13
Resource Allocation Solutions	13
Queue-Based Algorithms	14
Transaction Synchronization	15
Transactional Memory	15
Lock Inference	16
Semantic Conflict Detection	17
Search Data Structures	18
Search Trees	19
Skiplists	20
Tries	21
CHAPTER 3: METHODOLOGY	22
Multi-resource Lock	22
Motivation	24
A Naive Algorithm	25

Queue-based Algorithm	26
Acquiring Locks	29
Releasing Locks	31
Bitset Operations	31
Lock-free Transactional Transformation	32
Overview	33
Data Type Definition	35
Node-based Conflict Detection	37
Logical Status Interpretation	38
Logical Status Update	40
Transaction Execution	41
Multi-dimensional Linked List	43
Motivation	44
Overview	46
Definition	47
Data Types	48
Concurrent Find	49

Concurrent Insert	51
Concurrent Delete	54
CHAPTER 4: LIBRARY IMPLEMENTATION	57
Interface Design	57
Unified Transaction Descriptor	58
Set Interface	61
Applying MRLock	61
Applying LFTT	63
Code Transformation	63
Code Templates	64
CHAPTER 5: EXPERIMENTAL EVALUATION	67
Lock-based Transactions	67
Experiment Setup	67
Single-thread Overhead	69
Resource Scalability	71
Thread Scalability	75
Performance Consistency	77

Lock-free Transactions	78
Transactional List	79
Transactional Skiplist	83
Lock-free Dictionaries	85
CHAPTER 6: CONCLUSION	93
APPENDIX A: CORRECTNESS PROOF OF MRLOCK	95
Safety	96
Liveness	97
APPENDIX B: CORRECTNESS PROOF OF LFTT	99
Definitions	100
Serializability and Recoverability	102
Progress Guarantees	104
APPENDIX C: CORRECTNESS PROOF OF MDLIST	106
Invariants	107
Linearizability	109
Lock Freedom	111

REFERENCES 113

LIST OF FIGURES

1	False conflict in STM	4
2	Atomic lock acquisition process	27
3	Transaction Execution and Conflict	38
4	BSTs have various layouts for the same logical ordering (a and b). The linked list (c) has deterministic layout that is independent of execution histories.	45
5	FIND operation in a 3DList ($D = 3, N = 64$)	49
6	INSERT operation in a 3DList ($D = 3, N = 64$)	51
7	Contention scaling up to 64 resources	70
8	Contention scaling up to 1024 resources	72
9	Thread Scaling Trends for 64 Resources	74
10	Thread Scaling Trends for 1024 Resources	75
11	Standard deviation (11a, 11b) and relative error (11a, 11b) out of 10 runs	78
12	Transaction Lists (10K Key Range)	81
13	Transaction Skiplists (1M Key Range)	84
14	50% INSERT, 50% DELETE, 0% FIND on the NUMA System	86

15	20% INSERT, 10% DELETE, 70% FIND on the NUMA System	87
16	9% INSERT, 1% DELETE, 90% FIND on the NUMA System	89
17	Throughput on SMP system and dimension sweep on NUMA	91

LIST OF TABLES

1	Coding Convention	7
2	Lock overhead obtained without contention	70

LIST OF ALGORITHMS

1	TATAS lock for resource allocation	26
2	Multi-Resource Lock Data Structures	28
3	Multi-Resource Lock Acquire	29
4	Multi-Resource Lock Release	30
5	Bitset Operations	32
6	Type Definitions	36
7	Pointer Marking	37
8	Logical Status	39
9	Update NodeInfo	40
10	Transaction Execution	42
11	Lock-free Dictionary Data Structure	48
12	Pointer Marking	48
13	Concurrent Find	50
14	Predecessor Query	50
15	Concurrent Insert	53
16	Child Adoption	54
17	Concurrent Delete	55
18	Libtxd Transaction Descriptor	58
19	Libtxd Usage Example	59
20	Libtxd Set Interface	60
21	Libtxd MRLock Interface	62
22	Template for Transformed Insert Function	64
23	Template for Transformed Find Function	64
24	Template for Transformed Delete Function	65

25 Micro-Benchmark 68

CHAPTER 1: INTRODUCTION

This dissertation is concerned with the design and implementation of high-performance transactional data structures. By providing two designing strategies — one lock-based and one lock-free — I show that a wide range of useful transactional data structures can be built from their concurrent counterparts. My experimental results demonstrate that the performance of proposed transactional data structures surpass state-of-the-art generic constructions based on transactional memory.

In this chapter, I outline the core issues with exiting concurrent data structure which motivate this work, and highlight the contributions that are described in this dissertation. I then summarize the contents of each chapter and explain the pseudo code language convention used in this work.

Motivation

With the growing prevalence of multi-core systems numerous highly concurrent non-blocking data structures have emerged [65, 29, 97, 74]. Researchers and advanced users have been using libraries like LibCDS ¹, Tervel ² and Intel TBB ³, which are packed with efficient concurrent implementations of fundamental data structures. High level programming languages such as C#, Java and Scala also introduce concurrent libraries, which allow users who are unaware of the pitfalls of concurrent programming to safely take advantage of the performance benefits of increased concurrency. These libraries provide operations that appear to execute atomically when invoked individually. However, they fall short when users need to execute a sequence of operations atomically (i.e., compose operations in the manner of a transaction). Users thus cannot extend or compose the

¹<http://libcdfs.sourceforge.net/>

²<http://ucf-cs.github.io/Tervel/>

³<https://www.threadingbuildingblocks.org/>

provided operations safely without breaking atomicity. For example, given a concurrent map data structure, the following code snippet implementing a simple COMPUTEIFABSENT pattern [37] is error prone.

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

```

The intention of this code is to compute a value and store it in the map, if and only if the map does not already contain the given key. The code snippet fails to achieve this since another thread may have stored a value associated with the same key right after the execution of CONTAINSKEY and before the invocation of PUT. As a result, the thread will overwrite the value inserted by the other thread upon the completion of PUT. Programmers may experience unexpected behavior due to the violation of the intended semantics of COMPUTEIFABSENT. Many Java programs encounter bugs that are caused by such non-atomic composition of operations [90]. Because of such hazards, users are often forced to fall back to locking and even coarse-grained locking, which has a negative impact on performance and annihilates the non-blocking progress guarantees provided by some concurrent containers.

Moreover, the complexity of operation composition drastically increases in real world programs, where updates to multiple data structures need to be coordinated simultaneously. Consider a program that needs to concurrently maintain a group of items with different properties in separate sets, such an order processing program with a set of pending orders and a set of processed orders. The program needs to atomically extract one order from the pending set and put it in the processed set. If the sets are implemented using hash tables with fine-grained locks, then executing this composed operation while maintaining data structure consistency requires acquiring one lock in the pending set and one lock in the processed set. This would break the clean interface and encapsulation of the

concurrent sets by exposing their internal locks, because it is not sufficient to perform the two operations separately. This use case may also lead to deadlock if the locks are not acquired following a global order. As the number of data structures and operations involved in a transaction increases, the user's code will be exponentially more complicated. On the other hand, lock-free sets do not even have the option of exposing their internal synchronization mechanism to users.

The problem of implementing high-performance transactional data structures⁴ is important and has recently gained much attention [37, 12, 52, 40, 39, 48, 63]. Supporting composable transaction for concurrent data structures is of paramount importance for building reusable software systems. I refer to a transaction as sequence of linearizable operations on one or more concurrent data structures. This can be seen as a special case of memory transactions where the granularity of synchronization is on the data structure operation level instead of memory word level. I consider concurrent data structures “transactional” if they support executing transactions 1) atomically (i.e., if one operation fails, the entire transaction should abort), and 2) in isolation (i.e., concurrent executions of transactions appear to take effect in some sequential order). In this dissertation, I focus the discussion on the data structures that implement set and map *abstract data types*⁵. Concurrent sets and maps have emerged as one of the main abstractions of multi-core programming because their semantics allow maximum *disjoint access parallelism* compared with other data structures.

Software transactional memory (STM) [91, 55] can be used to conveniently construct transactional data structures from their sequential counterparts: operations executed within an STM transaction are guaranteed to be transactional. It can be used to atomically compose multiple operations on a single data structure, or across multiple data structures. Despite the appeal of straightforward

⁴Also referred as *atomic composite operations* [37]

⁵Also referred to as collection and dictionary interfaces in software engineering. In this dissertation, I use the term interchangeably

implementation, this approach has yet to gain practical acceptance due to its significant runtime overhead [14]. An STM instruments threads' memory accesses by recording the locations a thread reads in a *read set*, and the locations it writes in a *write set*. Conflicts are detected among the *read/write sets* of different threads. In the presence of conflicts, only one transaction is allowed to commit while the others are aborted and restarted. Apart from the overhead of metadata management, excessive transaction aborts in the presence of data structure “hot-spots” (memory locations that are constantly accessed by threads, e.g., the head node of a linked list) limit the overall concurrency [52]. Figure 1 illustrates such an example. It shows a set implemented as an ordered linked list, where each node has two fields, an integer value and a pointer to the next node. The initial state of the set is $\{0, 3, 6, 9, 10\}$. Thread 1 and Thread 2 intend to insert 4 and 1, respectively. Since these two operations commute, it is feasible to execute them concurrently [16]. In fact, 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. The inherent disadvantage of STM concurrency control is that *low-level memory access conflicts do not necessarily correspond to high-level semantic conflicts*.

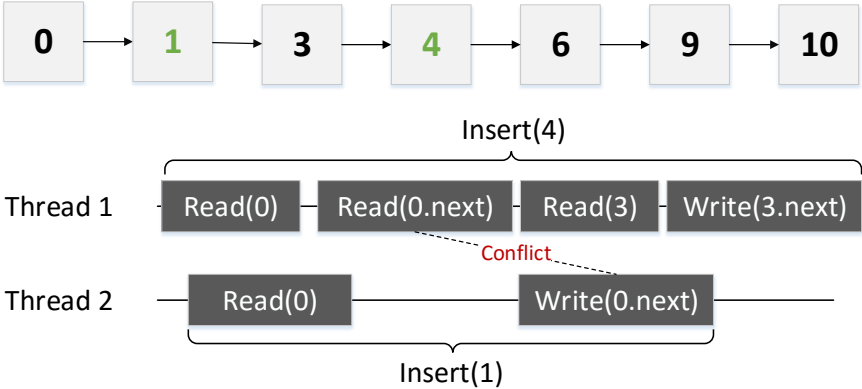


Figure 1: False conflict in STM

Contribution

It is my thesis that exiting concurrent data structures should support efficient transactional operation in order for them to gain practical use. The methodologies and tools to develop transactional data structures should be placed within the reach of mainstream programmers. Existing memory transaction programming paradigm has been too restrictive in performance to be of practical use. In this dissertation, I introduce the design and implementation of efficient transactional data structures that facilitate the development of future highly concurrent software systems.

My first contribution is two methodologies for implementing high-performance transactional data structures using lock-based and using non-blocking synchronization. The locking strategy employs *multi-resource lock*, or MRLock, which uses a lock-free FIFO queue to manage locking requests in batches. When combined with existing lock-based transaction synchronization techniques such as semantic locking [38] and transaction boosting [52], it can be used by programmers who prefer lock-based code to implement high-performance transactional data structures on familiar grounds. The lock-free strategy involves the use of my *lock-free transactional transformation*, or LFTT methodology, which employs transaction descriptor objects to announce the transaction globally so that delayed threads can be helped. It is applicable to a large class of linked data structures such as linked lists, binary trees, and skip lists. It encapsulates all operations, operands, and transaction status in a transaction descriptor, which is shared among the nodes accessed by the same transaction. It coordinates threads to help finish the remaining operations of delayed transactions based on their transaction descriptors. When transaction fails, it recovers the correct abstract state by reversely interpreting the logical status of a node. Write operations are invisible to operations outside the scope of the transaction until the transaction commits.

A further contribution is a suite of high-performance transactional data structure designs that implements the widely used set and map abstract data types. This includes a brand new search data

structure — *multi-dimensional linked list*, or MDList, which is designed from scratch to provide maximum disjoint access parallelism. It has a distributed memory layout which alleviates memory access contention. A write operation modifies at most two nodes so that the interference among concurrent operations are brought down to a minimum. This data structure implements the map abstract data type, which is ubiquitous in modern applications. When combined with the above mentioned transactional strategies, this could greatly benefit application developers who deal with data intensive scenarios such as in memory databases.

Lastly, by fully implementing each of the designs and integrate them in a open source library I provide the concurrent programming community with a readily available building block for high-performance concurrent applications as well as a framework to continuously incorporating additional transactional data structures. A library of transactional data structures serves as a dedicated framework for transaction descriptor management, and a unified data structure pool where different instances of data structures have mutual awareness of the existence of each other. All transactional data structures within this library will support the same transaction descriptor format, which enables co-operative transaction execution across multiple data structures.

Pseudo-Code Convention

In this dissertation, I use a C++ like pseudo-code as well as actual C++ code snippet to list algorithms. Generic algorithms are listed in pseudo-code so that they can be easily implemented in another language. Library related code are displayed in C++ to preserve the language features lacked by the pseudo-code. For clarity of presentation, I use the notion of **inline** functions, which have implicit access to the caller's local variables without explicit argument passing. I also denote line *b* from algorithm *a* by *a.b*. Table 1 list the operators and primitives that are used in this dissertation.

Table 1: Coding Convention

Operator	C++ representation	Pseudo-code representation
Assignment	=	←
Equality	==	=
Relational	<, >, <=, >=	<, >, ≤, ≥
Logical	, &&, !	or, and, !
Bit Operation	, &	, &
Member Access	., - >	.
Pointer Dereference	*	*
Array Index	[i]	[i]
Index Range	not supported	[i : j]

Outline

In this section, I describe the organization of the rest of this dissertation.

In Chapter 2, I explain the terminology for concurrent data structures and transaction synchronization. I then review previous work which relates to and motivates my dissertation.

In Chapter 3, I motivate and present efficient designs of both lock-based and lock-free transactional data structures based on two easy to use techniques: multi-resource lock and lock-free transactional transformation. I also introduce a brand new search structure, multi-dimension list, which optimize concurrent write accesses

In Chapter 4, I discuss the implementation details of my transactional data structure library — *libtxd*. I also demonstrate how to apply MRLock and LFTT to obtain transactional data structures using the utility code provided in *libtxd*.

In Chapter 5, I explain the testing environment for my transactional data structures. I then present experimental results that showcase the practicality and efficiency of my new transactional designs

compared with the best performing alternatives.

In Chapter 6, I conclude the dissertation and suggest directions for future research.

Lastly, I also included detailed correctness proofs for MRLock, LFTT, and MDList in the Appendices.

CHAPTER 2: BACKGROUND

I start this chapter by presenting in greater details the technical terms that are generally used when discussing transactional data structures. Some of the terms are originated from database concurrency control, and they may have slightly different implication when applied in the context of data structures operations.

Terminology

I consider a concurrent data structure to be a set of memory locations that shared among multiple *threads* and are accessed and updated only by a group of operations defined by the API specifications. A concurrent data structure is blocking if it uses any kind of mutual exclusion locks to protect its shared data. Note that a concurrent data structure that use no locks is not necessarily non-blocking. Non-blocking data structures refer to those that satisfies one of the progress assurances describe in Section 2. Although blocking and non-blocking data structures differ in terms of progress assurances, they conform to the same set of concurrent correctness criteria and employ the same set of atomic primitives.

Atomic Primitives

Atomic primitives are the cornerstones of any synchronization algorithm. `COMPAREANDSWAP(ADDR, EXPECTED, VAL)`¹, or `CAS` for short, always returns the original value at the specified `ADDR` but only writes `VAL` to `ADDR` if the original value matches `EXPECTED`. `CAS` is preferred for two reasons: first, it is a *universal* atomic operation (infinite consensus number) [50], thus

¹Also known as `COMPAREEXCHANGE`

can be used to construct any other atomic operations; second, it is now widely supported in most systems after first appearing in the IBM 370. In C++ memory model [9], the use of an atomic operation is usually accompanied by a memory order argument (`std::memory_order`), which specify how regular memory accesses made by different threads should be ordered around the atomic operation. More specifically, the use of `std::memory_order_acquire` along with `std::memory_order_release` requires that when a thread performs an atomic load operation with `acquire` order, prior writes made to other memory locations by the thread that did the `release` become visible to it. `std::memory_order_relaxed`, on the hand, poses no ordering constraints.

Correctness Property

Sequential data structures can be verified against their sequential specifications by checking their pre- and post-conditions as well as program invariants. Determining if a concurrent data structure behaves as expected is more intricate as threads' execution can interleave resulting almost infinite possible execution histories. One widely adopted concurrent correctness property is *linearizability* [58]. This property is defined in terms of method invocations and responses from a compliant operations: if a method is a synchronized procedure then a call to that procedure is a invocation and the eventual return from that procedure is a response. A concurrent operation is linearizable if and only if it appears to took place instantaneously at some points between their invocations and responses. One important and desirable advantage of linearizability is that it is composable in the sense that any combination of linearizable objects is still linearizable. However, as I stated in the introduction, even though individual methods calls are linearizable there is no guarantee that the composed sequence of method calls is linearizable as a whole. In such cases, it is desirable that the sequence of method calls satisfies strict serializability [80], which is the analogue of linearizability [58] for transactions. A execution history is strictly serializable if its subsequence consisting

of all events of committed transactions is equivalent to a legal history in which these transactions execute sequentially in the order they commit.

Non-blocking Progress Assurance

Blocking data structures have very weak progress guarantee: if a thread crashes or gets preempted while holding a lock then it may be impossible for any concurrent operations to finish. Non-blocking synchronization [36] provides stronger progress guarantees by completely eliminating the use of mutual exclusion locks. A concurrent data structure is lock-free if at least one thread makes forward progress in a finite number of steps [51]. It is wait-free if all threads make forward progress in a finite number of steps. Compared to their blocking counterparts, non-blocking data structures promise greater scalability and robustness. One common way to construct a non-blocking object is to use CAS: each contending thread speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy [21].

Mutual Exclusion

Mutual exclusion algorithms are widely used to construct synchronization primitives like locks, semaphores and monitors. Designing efficient and scalable mutual exclusion algorithms has been extensively studied (Raynal [85] and Anderson [2] provide excellent surveys on this topic). In the classic form of the problem, competing threads are required to enter the critical section one at a time. In the k -mutual exclusion problem [32], k units of an identical shared resource exist so that up to k threads are able to acquire the shared resource at once. Further generalization of k -mutual exclusion gives the h -out-of- k mutual exclusion problem [84], in which a set of k identical resources are shared among threads. Each thread may request any number $1 \leq h \leq k$ of the resources, and the thread remains blocked until all the required resources become available.

Resource Allocation

Data structures transaction can be viewed as a form of resource allocation problem, [67], where memory locations in the data structure are resources and threads need to acquire exclusive accesses to them before update the memory values. The resource allocation problem on shared-memory multiprocessors extends the above mentioned h -out-of- k mutual exclusion problem in the sense that the resources are not necessarily identical. The minimal safety and liveness properties for any solution include mutual exclusion and deadlock-freedom [2]. Mutual exclusion means a resource must not be accessed by more than one thread at the same time, while deadlock-freedom guarantees system wide progress. Starvation-freedom, a stronger liveness property than deadlock-freedom, ensures every thread eventually gets the requested resources. In the strongest FIFO ordering, the threads are served in the order they arrive.

Related Work

To the best of my knowledge, there is no existing concurrent data structure that provides native support for transactions. A transactional execution of data structure operations can be seen as a restricted form of *software transactions* [45], in which the memory layout and the semantics of the operations are well defined according to the specification of the data structure. Straightforward generic constructions can be implemented by executing all shared memory accesses in coarse-grained *atomic sections*, which can employ either optimistic (e.g., STM) or pessimistic (e.g., lock inference) concurrency control. More sophisticated approaches [12, 52, 39] exploit semantic conflict detection for transaction-level synchronization to reduce benign conflicts. I draw inspirations from previous semantic based conflict detection approaches. However, with the specific knowledge on linked data structure, I further optimize the transaction execution by performing in-place conflict detection and contention management on existing nodes.

Lock-based Concurrency Control

In this section, I summarize the solutions to the resource allocation problem and related queue-based algorithms. I skip the approaches targeting distributed environments [7, 84]. These solutions do not transfer to shared-memory systems because of the drastically different communication characteristics. In distributed environments processes communicate with each other by message passing, while in shared-memory systems communication is done through shared memory objects. I also omit early mutual exclusion algorithms that use more primitive atomic read and write registers [85, 2]. As I show in section 2, the powerful CAS operation on modern multiprocessors greatly reduces the complexity of mutual exclusion algorithms.

Resource Allocation Solutions

Assuming each resource is guarded by a mutual exclusion lock, lock acquiring protocols can effectively prevent deadlocks. Resource hierarchy is one protocol given by Dijkstra [26] based on total ordering of the resources. Every thread locks resources in an increasing order of enumeration; if a needed resource is not available the thread holds the acquired locks and waits. Deadlock is not possible because there is no cycle in the resource dependency graph. Lynch [67] proposes a similar solution based on a partial ordering of the resources. Resource hierarchy is simple to implement, and when combined with queue mutex it is the most efficient existing approach. However, total ordering requires prior knowledge of all system resources, and dynamically incorporating new resources is difficult. Besides, FIFO fairness is not guaranteed because the final acquisition of the resources is always determined by the acquisition last lock in this hold-and-wait protocol. Two-phase locking [31] was originally proposed to address concurrency control in databases. At first, threads are allowed to acquire locks but not release them, and in the second phase threads are allowed to release locks without acquisition. For example, a thread tries to lock all needed

resources one at a time; if any one is not available the thread releases all the acquired locks and start over again. When applied to shared-memory systems, it requires a TRYLOCK method that returns immediately instead of blocking the thread when the lock is not available. Two-phase locking is flexible requiring no prior knowledge on resources other than the desired ones, but its performance degrades drastically under contention, because the release-and-wait protocol is vulnerable to failure and retry. Time stamp ordering [8] prevents deadlock by selecting an ordering among the threads. Usually a unique time stamp is assigned to the thread before it starts to lock the resources. Whenever there is a conflict the thread with smaller time stamp wins.

Queue-Based Algorithms

Fischer et al. [33] describes a simple FIFO queue algorithm for the k -mutual exclusion problem. Awerbuch and Saks [6] proposed the first queuing solution to the resource allocation problem. They treat it as a dynamic job scheduling problem, where each job encapsulates all the resources requested by one process. Newly enqueued jobs progress through the queue if no conflict is detected. Their solution is based on a distributed environment in which the enqueue and dequeue operation are done via message communication. Due to this limitation, they need to assume no two jobs are submitted concurrently. Spin locks such as the TATAS lock shown in Section 3.3 induce significant contention on large machines, leading to irregular timings. Queue-based spin locks eliminate these problems by making sure that each thread spins on a different memory location [89]. Anderson [3] embeds the queue in a Boolean array, the size of which equals the number of threads. Each thread determines its unique spin position by drawing a ticket. When relinquishing the lock, the thread resets the Boolean flag on the next slot to notify the waiting thread. The MCS lock [72] designed by Scott et al., employs a linked list with pointers from each thread to its successor. The CLH lock by Craig et al. [17] also employs a linked list but with pointers from each thread to its predecessor. A Recent queue lock based on *flat-combining* synchroniza-

tion [23] exhibits superior scalability on NUMA architecture than the above classic methods. The flat-combining technique reduce contention by aggregating lock acquisitions in batch and processing them with a combiner thread. A key difference between this technique and my multi-resource lock is that my method aggregates lock acquisition requests for multiple resources from one thread, while the flat-combining lock gathers requests from multiple threads for one resource. Although the above queue-based locks could not solve the resource allocation problem on their own, they share the same inspiration with my method: using a queue to reduce contention and provide FIFO fairness.

Transaction Synchronization

In this work, I combine both fine-grained thread-level synchronization found in existing lock-free data structures and semantic conflict detection to implement transactional linked data structures without the overhead of atomic section synchronizations.

Transactional Memory

Initially proposed as a set of hardware extensions by Herlihy and Moss [56], transactional memory was intended to facilitate the development of lock-free data structures. However, on current commodity chips hardware transaction memory (HTM) relies on cache-coherency based conflict detection scheme, so transactions are subject to spurious failures during page faults and context switches [22]. This makes HTM less desirable for data structure implementations. Considerable amount of work and ingenuity has instead gone into designing lock-free data structures using low-level synchronization primitives such as COMPAREANDSWAP, which empowers researchers to devise algorithm-specific fine-grained concurrency control protocol.

The first software transactional memory was proposed by Shavit and Touitou [91], which is lock-free but only supports a static set of data items. Herlihy, et al., later presented DSTM [55] that supports dynamic data sets on the condition that the progress guarantee is relaxed to obstruction-freedom. Over the years, a large number of STM implementations have been proposed [88, 35, 69, 24, 20]. I omit complete reviews because it is out of the scope of this paper. As more design choices were explored [68], emerging discussions on the issues of STM regarding usability [86], performance [14], and expressiveness [42] have been seen. There is also an increasing realization that the read/write conflicts inherently provide insufficient support for concurrency when shared objects are subject to contention [63]. It has been suggested that “STM may not deliver the promised efficiency and simplicity for *all* scenario, and multitude of approaches should be explored catering to different needs” [5].

Lock Inference

STM implementations are typically *optimistic*, which means they execute under the assumption that interferences are unlikely to occur. They maintain computationally expensive redo or undo logs to allow replay or rollback in case a transaction experiences interference. In light of this shortcoming, pessimistic alternatives based on lock inference have been proposed [71]. These algorithms synthesize enough locks through static analysis to prevent data races in atomic sections. The choice of locking granularity has an impact on the trade-off between concurrency and overhead. Some approaches require programmers’ annotation [37] to specify the granularity, others automatically infer locks at a fixed granularity [30] or even multiple granularities [15]. Nevertheless, most approaches associate locks with memory locations, which may lead to reduced parallelism due to false conflicts as seen in STM. Applying these approaches to real-world programs also faces scalability changes in the presence of large libraries [41] because of the high complexity involved in the static analysis process. Applying these approaches to real-world programs also faces scala-

bility changes in the presence of large libraries [41] because of the high cyclomatic complexity² involved in the static analysis process. Moreover, the use of locks degrades any non-blocking progress guarantee one might expect from using a non-blocking library.

Semantic Conflict Detection

Considering the imprecise nature of data-based conflict detection, semantic-based approaches have been proposed to identify conflicts at a high-level (e.g., two commutative operations would not raise conflict even though they may access and modify the same memory data) which enables greater parallelism. Because semantically independent transactions may have low-level memory access conflicts, some other concurrency control protocol must be used to protect accesses to the underlying data structure. This results in a two-layer concurrency control. Transactional boosting proposed by Herlihy [52] is the first dedicated treatment on building highly concurrent transactional data structures using a semantic layer of abstract locks. Transactional boosting [52] is a semantic-based methodology for transforming linearizable concurrent data structures into transactional data structures. The idea behind boosting is intuitive: if two operations commute they are allowed to proceed without interference (i.e., thread-level synchronization happens within the operations); otherwise they need to be synchronized at the transaction level. It treats the base data structure as a black box and uses *abstract locking* to ensure that non-commutative method calls do not occur concurrently. For each operation in a transaction, the boosted data structure calls the corresponding method of the underlying linearizable data structure after acquiring the abstract lock associated with that call. A transaction aborts when it fails to acquire an abstract lock, and it recovers from failure by invoking the inverses of already executed calls. Its semantic-based conflict detection approach eliminates excessive false conflicts associated with STM-based transactional

²A measure of the number of linearly independent execution paths [70].

data structures, but it still suffers from performance penalties due to the rollbacks of partially executed transactions. Moreover, when applied to non-blocking data structures, the progress guarantee of the boosted data structure is degraded because of the locks used for transactional-level synchronization. Transactional boosting is pessimistic in that it acquires locks eagerly before the method calls, but it still requires operation rollback because not all locks are acquired at once. Koskinen et al. [63] later generalized this work and introduce a formal framework called coarse-grained transactions. Bronson et al. proposed transaction prediction, which maps the abstract state of a set into memory blocks called predicate and relies on STM to synchronize transactional accesses [12]. Hassan et al. [49] proposed an optimistic version of boosting, which employs a white box design and provides throughput and usability benefits over the original boosting approach. Other STM variants, such open nested transactions [77] support a more relaxed transaction model that leverages some semantic knowledge based on programmers' input. The relaxation of STM systems and its implication on composability have been studied by Gramoli et al. [40]. The work by Golan-Gueta et al. [39] applies commutativity specification obtained from programmers' input to inferring locks for abstract data types.

Search Data Structures

Concurrent key-value store data structures that implement abstract dictionaries have been extensively studied in the literature. Unordered dictionaries can be built upon non-blocking hash tables [74], which achieve $\mathcal{O}(1)$ amortized cost. Range queries are not attainable on such data structures because keys are not totally ordered. I thus focus my discussion on BSTs and skiplists, which provide totally ordered key-value store and logarithmic search time by retaining balance either explicitly or probabilistically.

Search Trees

Early concurrent BSTs [44, 64, 78] are mostly lock-based adaptations of the sequential data structures, which focused on decoupling update and rebalancing to explore coarse-grained parallelism. The strict balance invariants were relaxed in the sense that the balancing condition can be temporarily violated by updates and eventually restored by subsequent rebalancing operations. Recent fine-grained locking implementations by Bronson et al. [11] and Afek et al. [1] take advantage of optimistic concurrently control and hand over hand validation to reduce synchronization overhead. The lock-based relaxed AVL tree by Crain et al. [18] further reduces contention by delaying tree rotation and employing a dedicated maintenance thread to remove logically deleted nodes. Drachsler et al. [27] proposed a relaxed balanced and an unbalanced lock-based BSTs in which the nodes store additional logical ordering data by pointing to its logical predecessor and successor. The search operation would traverse the logical chain to locate the target if it does not find the target after reaching a physical leaf node.

Fraser [35] presented a non-blocking BST using multi-word CAS, which is not available on existing multi-core chips and expensive to implement using CAS. The first practical lock-free linearizable BST design was given by Ellen et al. [29]. Their implementation was based on a leaf-oriented BST, where values are stored externally in leaf nodes and internal nodes were only used for routing. Howley and Jones [59] presented a lock-free node-oriented BST based on the same co-operative technique. Their algorithm has faster search operations than those in Ellen et al.'s algorithm because the search path is generally shorter in a node-oriented BST than in a leaf-oriented one. On the other hand, Ellen et al.'s DELETE operations, which avoid removing nodes with two children, are simpler and faster at the price of extra memory for internal nodes. Natarajan and Mittal [76] proposed a lock-free leaf-oriented BST, which marks edges instead of nodes. Their algorithm is more efficient than previous approaches because the mutating operations work on a smaller portion

of the tree and execute fewer atomic instructions. Due to the complexity of rebalancing, all of the above non-blocking trees are not balanced, thus they subject to potential linear runtime for certain inputs. Brown et al. [13] proposed a general template for implementing lock-free linearizable down-trees. In order to orchestrate rebalancing among threads, their tree update routine atomically replaces a subgraph with a new connected subgraph using a set of multi-word synchronization primitives. As recognized by the above researches, non-blocking rebalancing presents the biggest challenge to practical non-blocking BSTs.

Skiplists

Pugh [82] designed a concurrent skiplist with per-pointer locks, where an update to a pointer must be protected by a lock. Shavit [92] discovered that the highly decentralized skiplist is suitable for shared-memory systems and presents a concurrent priority queue based on Pugh's algorithm. In practice, probabilistic balancing is easier to implement and as efficient as deterministic balancing. This is why the concurrent map class of the Java standard library uses skiplists rather than search trees. Fraser [35] proposed a lock-free skiplist with an epoch based memory reclamation technique. It also employs the logical deletion technique, which was originally proposed by Harris [46], to mark pointers and delay physical removal. Fomitchev and Ruppert [34] sketched a lock-free skiplist design but did not offer any implementation. Sundell and Tsigas [94] presented a provably correct linearizable lock-free skiplist, which uses exponential back-off to alleviate contention. They guarantee linearizability by forcing threads to help physically remove a node before moving past it. Herlihy et al. [53] proposed an optimistic concurrent skiplist that simplifies previous work and allows for easy proof of correctness while maintaining comparable performance. They later presented a optimistic skiplist [54] that uses hand-over-hand locking to lock two nodes at a localized positions. Crain et al. [19] proposed a no hot spot skiplist that alleviates contention by localizing synchronization at the least contended part of the structure. Dick et al. [25] recently

presented an improvement over existing skiplists algorithms. Their lock-free skiplist uses rotating wheels instead of the usual towers to improve locality of reference and speedup traversals. Skiplists have also been used to build a number of hybrid data structures. Spiegel et al. [93] combined a skiplist and a B-tree to produce a lock-free multi-way search tree, improving spatial locality of reference of skiplists by storing several elements in a single node.

Tries

Prokopec et al. [81] proposed a lock-free hash array mapped trie using both single-word and double-word CAS. They introduce intermediate nodes to solve the synchronization issue of updating the branching nodes. However, the intermediate nodes also become hot spots for contention because every expansion of a branching node requires a new branching node to be linked to the corresponding intermediate node through CAS. Oshman and Shavit [79] proposed the lock-free skiptrie using double-word CAS, which combines a shallow skiplist with a x-fast trie to store high level nodes. The MDList also bears some similarity to above mentioned tries in that the keys are ordered by their prefixes: a node always shares the same key prefix with its parent nodes. The major difference lies in the partition strategies: in a trie a node shares the same prefix with all of its children, but in an MDList a node shares prefixes of different lengths with each of its child. This leads to a constant branch factor for nodes in tries and reducing branching factors for bottom levels nodes in MDLists. Besides, retrieving the minimal key in a trie requires repetitive search, while MDList behaves like a heap where the root node always holds the smallest key. Memory wise, the values are stored in a trie's leaf nodes, whereas they are stored in an MDList's internal nodes.

CHAPTER 3: METHODOLOGY

In Section 1.1, I described how existing generic constructions of transactional data structures suffer from performance bottlenecks. In this chapter, I introduce two methodologies for constructing efficient transactional data structures from their concurrent counterparts, which are referred as the *base data structures*. The locking strategy employs multi-resource lock, or MRLock, which uses a lock-free FIFO queue to manage locking requests in batches. When combined with existing lock-based transaction synchronization techniques such as semantic locking [38] and transaction boosting [52], it can be used by programmers who prefer lock-based code to implement high-performance transactional data structures in a familiar way. The lock-free strategy involves the use of the *lock-free transactional transformation* or LFTT, methodology, which employs transaction descriptor object to announce the transaction globally so that delayed threads can be helped. It is applicable to a large class of linked data structures such as linked lists and skip lists. Because both strategies introduces additional code paths for transactional synchronization, the performance of the obtained transactional data structures largely depends on and cannot exceed the performance of the base data structures. To obtain even better performing transactional sets and maps than the ones built from existing concurrent data structures such as linked lists and skiplists, I also introduce a novel lock-free search data structure — multi-dimensional linked list, or MDList. An MDList readily maintain its ordering property without rebalancing nor randomization, and it reduces write access conflict to maximize disjoint access parallelism.

Multi-resource Lock

In this section, I propose the first FIFO (first-in, first-out) multi-resource lock algorithm for solving the resource allocation problem on shared-memory multiprocessors. As mentioned in Section 2.2,

this lends support to implementing lock-based transactional data structures. Given k resources, instead of having k separate locks for each one, I employ a non-blocking queue as the centralized manager. Each element in the queue is a resource request bitset¹ of length k with each bit representing the state of one resource. The manager accepts the resource requests in a first-come, first-served fashion: new requests are enqueued to the tail, and then they progress through the queue in a way that no two conflicting requests can reach the head of the queue at the same time. Using the bitset, it detects resource conflict by matching the correspondent bits. I also introduce an adaptive scheme that is composed of two lock algorithms of user's choice, for example the multi-resource lock which has scalable performance under high levels of contention and a two phase lock which excels at low levels of contention. The adaptive scheme then alternates the use the locks depending on the system wide contention level. The key algorithmic advantages of my approaches include:

1. The FIFO nature of the manager guarantees fair acquisition of locks, while implying starvation-freedom and deadlock-freedom
2. The lock manager has low access overhead and is scalable with the cost of enqueue and dequeue being only a single COMPAREANDSWAP operation
3. The maximum concurrency is preserved as a thread is blocked only when there are outstanding conflicting resource requests
4. Using a bitset allows an arbitrary number of resources to be tracked with low memory overhead, and does not require atomic access
5. The adaptive scheme combines the strength of two lock algorithms and provides overall better performance

¹A bitset is a data structure that contains an array of bits.

Motivation

Improving the scalability of resource allocation algorithms on shared-memory multiprocessors is of practical importance due to the trend of developing many-core chips [10]. The performance of parallel applications on a shared-memory multiprocessor is often limited by contention for shared resources, creating the need for efficient synchronization methods. In particular, the limitations of the synchronization techniques used in existing database systems leads to poor scalability and reduced throughput on modern multicore machines [60]. For example, when running on a machine with 32 hardware threads, Berkeley DB was reported to spend over 80% of the execution time in its Test-and-Test-and-Set lock [60].

Mutual exclusion locks eliminate race conditions by limiting concurrency and enforcing sequential access to shared resources. Comparing to more intricate approaches like lock-free synchronization [57] and transactional memory [56], mutual exclusion locks introduce sequential ordering that eases the reasoning about correctness. Despite the popular use of mutual exclusion locks, one requires extreme caution when using multiple mutual exclusion locks together. In a system with several shared resources, threads often need more than just one resource to complete certain tasks, and assigning one mutual exclusion lock to one resource is common practice. Without coordination between locks this can produce undesirable effects such as deadlock, livelock and decrease in performance.

Consider two clerks, *Joe* and *Doe*, transferring money between two bank accounts C_1 and C_2 , where the accounts are exclusive shared resources and the clerks are two contending threads. To prevent conflicting access, a lock is associated with each bank account. The clerks need to acquire both locks before transferring the money. The problem is that mutual exclusion locks cannot be composed, meaning that acquiring multiple locks inappropriately may lead to deadlock. For example, when *Joe* locks the account C_1 then he attempts to lock C_2 . In the meantime, *Doe* has

acquired the lock on C_2 and waits for the lock on C_1 . In general, one seeks to allocate multiple resources among contending threads that guarantees forward system progress, which is known as the resource allocation problem [32]. Two pervasive solutions, namely resource hierarchy [26] and two-phase locking [31], prevent the occurrence of deadlocks but do not respect the fairness among threads and their performance degrades as the level of contention increases. Nevertheless, both the GNU C++ library [9] and the Boost library [61] adopt the two-phase locking mechanism as a means to avoid deadlocks.

A Naive Algorithm

Given the atomic CAS instruction, it is straightforward to develop simple spin locks. In Algorithm 1 I present an extended TATAS lock that solves the resource allocation problem for a small number of resources. The basic TATAS lock [87] is a spin lock that allows threads to busy-wait on the initial `test` instruction to reduce bus traffic. The key change I made is to treat the lock integer value as a bit array instead of a Boolean flag. A thread needs to specify the resource requests through a bitset mask when acquiring and releasing the lock. With each bit representing a resource, the bits associated with the desired resources are set to 1 while others remain 0. The request updates the relevant bits in the lock bitset if there is no conflict, otherwise the thread spins. One drawback of this extension is that the total number of resources is limited by the size of integer type because a bitset capable of representing arbitrary number of resources may span across multiple memory words. Updating multiple words atomically is not possible without resorting to multi-word CAS [47], which is not readily available on all platforms.

Algorithm 1 TATAS lock for resource allocation

```
1 typedef uint64 bitset; //use 64bit integer as bitset
2
3 //input l: the address of the lock
4 //input r: the resource request bit mask
5 void tatas_lock(bitset* l, bitset r){
6     bitset b;
7     do{
8         b = *l; //read bits value
9         if(b & r) //check for conflict
10            continue; //spin with reads
11    }while(!compare_and_set(l, b, b | r));
12 }
13
14 void tatas_unlock(bitset* l, bitset r){
15     bitset b;
16     do{
17         b = *l;
18    }while(!compare_and_set(l, b, b & ~r));
19 }
```

Queue-based Algorithm

I implement a queue-based multi-resource lock that manages an arbitrary number of exclusive resources on shared-memory architectures. My highly scalable algorithm controls resource request conflicts by holding all requests in a FIFO queue and allocating resources to the threads that reach the top of the queue. It achieves scalable behavior by representing resource requests as a bitset and by employing a non-blocking queue that grants fair acquisition of the locks.

My conflict management approach is built on an array-based bounded lock-free FIFO queue [57]. The lock-free property is desirable as my lock manager must guarantee deadlock freedom. The FIFO property of the data structure allows for serving threads in their arriving order, implying starvation-freedom for all enqueued requests. I favor an array-based queue over other high performance non-blocking queues because it does not require dynamic memory management. Link-list based queues involve dynamic memory allocation for new nodes, which could lead to significant

performance overhead and the ABA² problem [73]. With a pre-allocated continuous buffer, my lock algorithm is not prone to the ABA problem and has low runtime overhead by using a single CAS for both enqueue and dequeue operations.

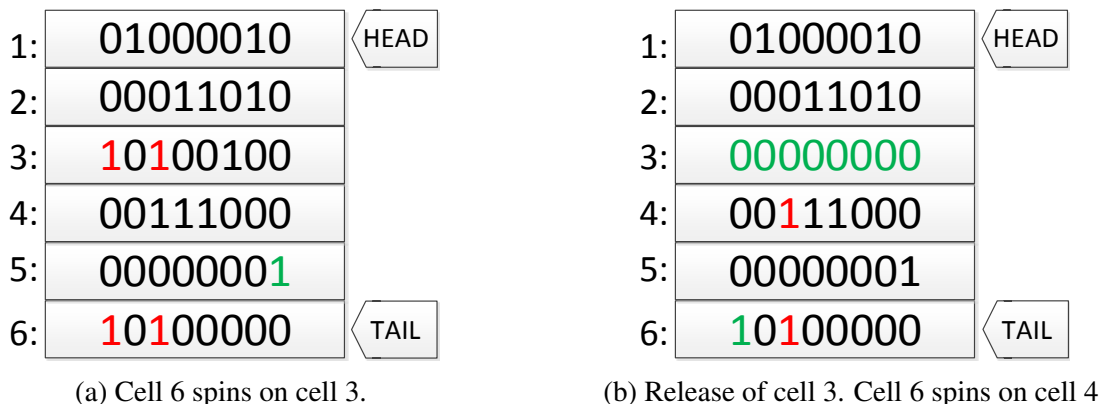


Figure 2: Atomic lock acquisition process

Given a set of resources, each bit in a request bitset is uniquely mapped to one resource. A thread encapsulates a request of multiple resources in one bitset with the correspondent bit of the requested resources set to 1. The multi-resource lock handles requests atomically meaning that a request is fulfilled only if all requested resources are made available, otherwise the thread waits in the queue. This all-or-nothing atomic acquisition allows the maximum number of threads, without conflicting requests, to use the shared resources. The length of the bitset is unlimited and can be set either at runtime as in `boost::dynamic_bitset`, or at compile time as in `std::bitset`. Using variable length bitset is also possible to accommodate growing number of total resources at runtime, as long as the resource mapping is maintained.

Figure 2a demonstrates this approach. A newly enqueued request is placed at the tail. Starting

²Note that ABA is not an acronym. It refers situations where a thread reads value A at some address and later attempts CAS operation expecting value A [57]. However, between the read and the CAS another thread has changed the value from A to B and back to A, thus the CAS operation succeed when it should not.

from the queue head, it compares the bitset value with each request. In the absence of conflict, it moves on to the next one until it reaches itself. Here, the thread on 5th cell successfully acquires all needed resources. The thread on the tail (6th cell) spins on the 3rd request due to conflict. In Figure 2b, the thread on the tail proceeds to spin on the 4th cell when the 3rd request was released.

Algorithm 2 Multi-Resource Lock Data Structures

```

1  #include<bitset.h>
2  #include<atomic>
3  using namespace std;
4
5  struct cell{
6      atomic<uint32> seq; //sequence number
7      bitset bits;      //resource request bits
8  }
9  struct mrlock{
10     cell* buffer;      //ring buffer of cells
11     uint32 mask;       //mask for fast modulation
12     atomic<uint32> head; //head position
13     atomic<uint32> tail; //tail position
14 }
15
16 //input l: reference to the lock instance
17 //input siz: required buffer size (power of 2)
18 void init(mrlock& l, uint32 siz){
19     l.buffer = new cell[siz];
20     l.mask = siz - 1;
21     l.head.store(0, memory_order_relaxed);
22     l.tail.store(0, memory_order_relaxed);
23     //initialize bits to all 1s and seq to cell index
24     for (uint32 i = 0; i < siz; i++) {
25         l.buffer[i].bits.set();
26         l.buffer[i].seq.store(i, memory_order_relaxed);
27     }
28 }
29
30 void uninit(mrlock& l){
31     delete[] l.buffer;
32 }

```

Algorithm 2 defines the lock manager’s class. The `cell` structure defines one element in the queue, it consists of a bitset that represents a resource request and an atomic *sequence number* that coordinates concurrent access. The `mrlock` structure contains a cell buffer pointer, the size mask,

and the queue head and tail. I use the size mask to apply fast index modulus. In my implementation, the head and tail increase monotonically; I use an index modulus to map them to the correct array position. Expensive modulo operations can be replaced by bitwise and if the buffer size is chosen to be a power of two. I discuss the choice of buffer size in Section A, and explain the initialization of the sequence number and the bitset in following section.

Algorithm 3 Multi-Resource Lock Acquire

```

1  //input l: reference to mrlock structure
2  //input r: resource request
3  //output : the lock handle
4  uint32 acquire_lock(mrlock& l, bitset r){
5      cell* c;
6      uint32 pos;
7      for(;;){          //cas loop, try to increase tail
8          pos = l.tail.load(memory_order_relaxed);
9          c = &l.buffer[pos & l.mask];
10         uint32 seq = c->seq.load(memory_order_acquire);
11         int32 dif = (int32)seq - (int32)pos;
12         if(dif == 0){
13             if(l.tail.compare_exchange_weak(pos, pos + 1, memory_order_relaxed))
14                 break;
15         }
16     }
17     c->bits = r; // update the cell content
18     c->seq.store(pos + 1, memory_order_release);
19     uint32 spin = l.head;
20     while(spin != pos){
21         if(pos - l.buffer[spin & l.mask].seq > l.mask || !(l.buffer[spin & l.
22             mask].bits & r))
23             spin++;
24     }
25     return pos;
}

```

Acquiring Locks

In Algorithm 3, the code from line 3.7 to line 3.16 outlines a CAS-based loop, with threads competing to update the queue tail on line 3.13. If the CAS attempt succeeds the thread is granted access to the `cell` at the tail position, and the tail is advanced by one. The thread then stores its

resource request, which is passed to `acquire_lock` as the variable `r`, in the cell along with a sequence number. The sequence number serves as a sentinel in my implementation. During the enqueue operation the thread assigns a sequence number to its `cell` as it enters the queue as seen on line 3.18. The nature of a bounded queue allows the head and tail pointers to move through a circular array. Dequeue attempts to increment the head pointer towards the current tail, while a successful call to enqueue will increment the tail pointer pulling it away from head. The sequence numbers are initialized on line 2.26 in Algorithm 2. It is also used by the `release_lock` function in Algorithm 4.

Algorithm 4 Multi-Resource Lock Release

```

1 //input l: reference to mrlock instance
2 //input h: the lock handle
3 void release_lock(mrlock& l, uint32 h){
4     l.buffer[h & l.mask].bits.reset();
5     uint32 pos = l.head.load(memory_order_relaxed);
6     while(l.buffer[pos & l.mask].bits == 0){
7         cell* c = &l.buffer[pos & l.mask];
8         uint32 seq = c->seq.load(memory_order_acquire);
9         int32 dif = (int32)seq - (int32)(pos + 1);
10        if(dif == 0){
11            if(l.head.compare_exchange_weak(pos, pos + 1, memory_order_relaxed))
12                {
13                    c->bits.set();
14                    c->seq.store(pos + l.mask + 1, memory_order_release);
15                }
16            pos = l.head.load(memory_order_relaxed);
17        }
18    }

```

Once a thread successfully enqueues its request, it spins in the while loop on line 3.20 to 3.23. It traverses the queue beginning at the head. When there is a conflict of resources indicated by the bitset, the thread will spin locally on the conflicting request. Line 3.21 displays two conditions that allow the thread to advance: 1) the cell the thread is spinning on is free and recycled, meaning the cell is no longer in front of this thread. This condition is detected by the use of sequence numbers; 2) The request in the cell has no conflict, which is tested by bitwise and of the two requests. Once

the thread reaches its position in the queue, it is safe to assume the thread has acquired the requested resources. The position of the enqueued request is returned as a handle, which is required when releasing the locks.

Releasing Locks

The `release_lock` function releases the locks on the requested resources by setting the bitset fields to zero using the lock handle, on line 4.4 of Algorithm 4. This allows threads waiting for this position to continue traversing the queue. The removal of the request from the queue is delayed until the request in the head cell is cleared (line 4.6). If a thread is releasing the lock on the head cell, the releasing operation will perform dequeue and recycle the cell. The thread will also examine and dequeue the cells at the top of the queue until a nonzero bitset is found. The code between lines 4.6 and 4.17 outlines a CAS loop that is similar to the enqueue function. The difference is that here threads assist each other with the work of advancing the head pointer. With this release mechanism, threads which finish before becoming the head of the queue do not block the other threads.

Bitset Operations

I represent the array of resources status in the form of bitset because its compact memory layout facilitates efficiency access for both lock acquire and release functions. In Algorithm 3 line 3.21 the two bitsets are compared using bitwise AND operation, and in Algorithm 4 line 4.4 and 4.12 the bitset store in a cell is cleared and re-initialized. In case of 64 or less resources, the bitset can be implemented as a single 64-bit integer. Otherwise, the bits array can be composed by an array of integers. I show an excerpt of the bitset class in Algorithm 5. The bitwise AND operation simply walk through the integer array and perform AND on individual pair of integers. It is easy to deduce

that most bitset operations requires linear time with respect to the total number of resources divided by a constant factor N , where N is the word size of the processor. The length of the integer array grows reasonably slow on a 64-bit system; my approach consistently outperform the alternatives up to 1024 resources. Moreover, modern CPUs like the Haswell chips by Intel support vector instructions that computes AND between two 256-bit integers in a single step [66], which further improves the efficiency of bitset operations.

Algorithm 5 Bitset Operations

```
1  class Bitset {
2      int64 m_words, *m_bits;
3      bool operator & (const Bitset& rhs) const {
4          for (int64 i = 0; i < m_words; i++) {
5              if(m_bits[i] & rhs.m_bits[i]) {
6                  return true;
7              }
8          }
9          return false;
10     }
11     void Set() {
12         memset(m_bits, ~0, m_words * sizeof(int64));
13     }
14     void Reset() {
15         memset(m_bits, 0, m_words * sizeof(int64));
16     }
17 };
```

Lock-free Transactional Transformation

In this section, I present *lock-free transactional transformation*: a methodology for transforming high-performance lock-free *base data structures* into high-performance lock-free transactional data structures. My approach is applicable to a large class of linked data structures—ones that comprise a set of data nodes organized by references. I focus my discussion here on the data structures that implement the set *abstract data type* with three canonical operations INSERT, DELETE, and FIND. Linked data structures are desirable for concurrent applications because their distributed

memory layout alleviates contention [92]. The specification for the base data structures thus defines an *abstract state*, which is the set of integer keys, and a *concrete state*, which consists of all accessible nodes. Lock-free transactional transformation treats the base data structure as a *white box*, and introduces a new code path for transaction-level synchronization using only the single-word COMPAREANDSWAP (CAS) synchronization primitive.

Overview

The two key challenges for high-performance data structure transaction executions are: 1) to efficiently buffer write operations so that their modifications are invisible to operations outside the transaction scope; and 2) to minimize the penalty of rollbacks when aborting partially executed transactions.

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

To overcome the second challenge, I introduce *logical rollback*—a process integrated into the transformed data structure to interpret the *logical status* of the nodes. This process interprets the logical status of the nodes left behind by an aborted transaction in such a way that concurrent operations observe a consistent abstract state as if the aborted transaction has been revoked. The

logical status defines how the concrete state of a data structure (i.e., set of nodes) should be mapped to its abstract state (i.e., set of keys). Usually the mapping is simple—every node in the concrete state corresponds to a key in the abstract state. Previous works on lock-free data structures have been using *logical deletion* [46], in which a key is considered removed from the abstract state if the corresponding node is bit-marked. Logical deletion encodes a binary logical status so that a node maps to a key only if its reference has not been bit-marked. I generalize this technique by interpreting a node’s logical status based on the combination of transaction status and operation type. The intuition behind my approach is that operations can recover the correct abstract state by *inversely interpreting* the logical status of the nodes with a descriptor indicating an aborted transaction. For example, if a transaction with two operations INSERT(3) and DELETE(4) fails because key 4 does not exist, the logical status of the newly inserted node with key 3 will be interpreted as *not inserted*.

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

I make the following contributions with the introduction of this methodology:

- To the best of my knowledge, lock-free transactional transformation is the first methodology that provides both lock-free progress and semantic conflict detection for data structure

transactions.

- I introduce a node-based conflict detection scheme that does not rely on STM nor require the use of an additional data structure. This enables us to augment linked data structures with native transaction support.
- I propose an efficient recovery strategy based on interpreting the logical status of the nodes instead of explicitly revoking executed operations in an aborted transaction.
- Data structures transformed by my approach gain substantial speedup over alternatives based on transactional boosting and the best-of-breed STMs; Due to cooperative execution in my approach, the number of aborts caused by node access conflict is brought down to a minimum.
- Because my transaction-level synchronization is compatible with the base data structure's thread-level synchronization, I am able to exploit the considerable amount of effort devoted to the development of lock-free data structures.

Data Type Definition

Any transactional data structure must cope with two tasks: conflict detection and recovery. In previous works [52, 39], locks were commonly used to prevent access conflict, and undo logs were often used to discard speculative changes when a transaction aborts. I introduce *lock-free transactional transformation*: a methodology that streamlines the execution of a transaction by abolishing locks and undo logs. My lock-free transactional transformation combines three key ideas: 1) node-based semantic conflict detection; 2) interpretation-based logical rollback; and 3) cooperative transaction execution. In this section, I explain these ideas and introduce the core procedures that will be used by transformed data structures: a) the procedure to interpret the logical

status of a node; b) the procedure to update an existing node with a new transaction descriptor; and c) the transaction execution procedure that orchestrates concurrent executions.

For clarity, I list the constants and data type definitions in Algorithm 6. In addition to the fields used by the base lock-free data structure, I introduce a new field *info* in NODE. NODEINFO stores *desc*, a reference to the shared transaction descriptor, and an index *opid*, which provides a history record on the last access (i.e., given a node *n*, *n.info.desc.ops[n.desc.opid]* is the most recent operation that accessed it). A node is considered *active* when the last transaction that accessed the node had an active status (i.e., *n.info.desc.status = Active*). A descriptor [57] is a shared object commonly used in lock-free programming to announce steps that cannot be done by a single atomic primitive. The functionalities of my transaction descriptor is twofold: 1) it stores all the necessary context for helping finish a delayed transaction; and 2) it shares the transaction status among all nodes participating in the same transaction. For set operations, I pack the high-level instructions including the operation type and the operand using only 8 bytes per operation. I also employ the pointer marking technique described by Harris [46] to designate logically deleted nodes. The macros for pointer marking are defined in Algorithm 12. The *Mark* flag is co-located with the *info* pointers.

Algorithm 6 Type Definitions

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

Algorithm 7 Pointer Marking

```
1: int Mark  $\leftarrow 0 \times 1$ 
2: define SetMark(p) (p | Mark)
3: define ClearMark(p) (p &  $\sim$  Mark)
4: define IsMarked(p) (p & Mark)
```

Node-based Conflict Detection

In my approach, conflicts are detected at the granularity of a node. If two transactions access different nodes (i.e. the method calls in them commute [52]), they are allowed to proceed concurrently. In this case, shared-memory accesses are coordinated by the concurrency control protocol in the base data structure. Otherwise, threads proceed in a cooperative manner as detailed in Section 3. For set data types, each node is associated with a unique key, thus my conflict detection operates at the same granularity as the abstract locking used by transactional boosting.

I illustrate an example of node access conflict in Figure 3. At the beginning, Thread 1 committed a transaction t_1 inserting keys 1 and 3. Thread 2 attempted to insert keys 4 and 2 in transaction t_2 , while Thread 3 was concurrently executing transaction t_3 to delete keys 3 and 4. Thread 3 was able to perform its first operation by updating the *info* pointer on node 3 with a new `NODEINFO`. However, it encounters a conflict when attempting to update node 4 because Thread 2 has yet to finish its second operation. To enforce serialization, operations must not modify an active node. In order to guarantee lock-free progress, the later transaction must help carry out the remaining operations in the other active transaction.

My synchronization protocol is *pessimistic* in that it assigns a node to a transaction as soon as an operation requires it, for the duration of the transaction. Moreover, node-based conflict detection effectively compartmentalizes the execution of transactions. Completed operations will not be affected should a later operation in the transaction experience contention and need to retry

(most lock-free data structures use CAS-based retry loops). Each node acts as a checkpoint; once an operation successfully updates a node, the transaction advances one step towards completion. Data-based conflict detection, due to the lack of such algorithm-specific knowledge, has to restart the whole transaction upon conflict.

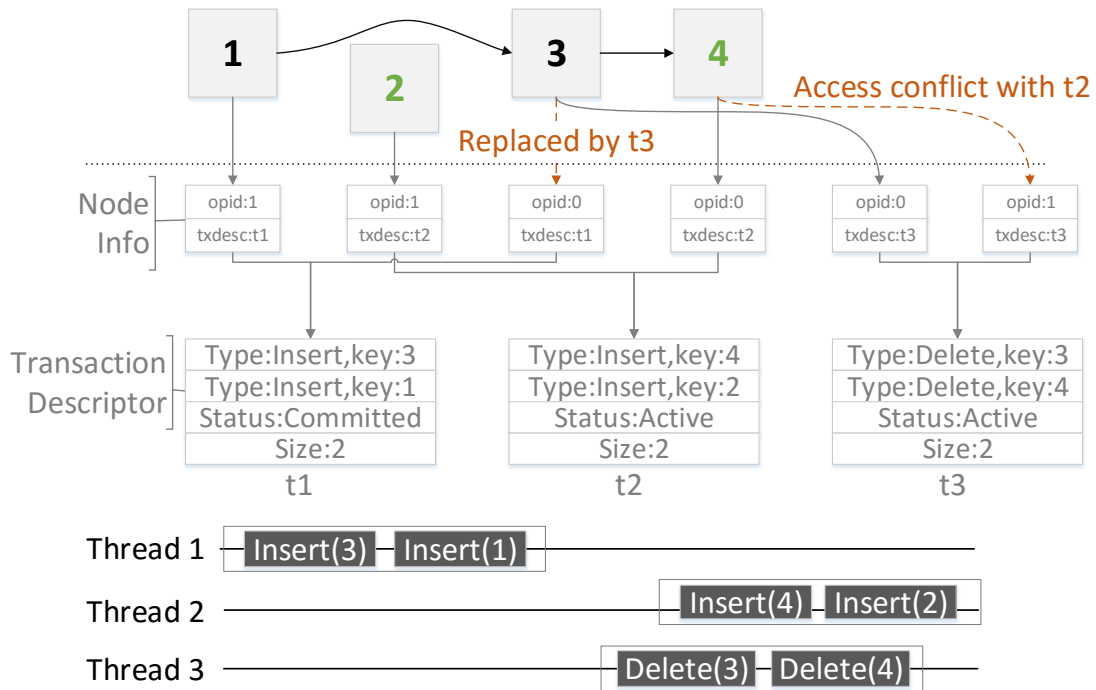


Figure 3: Transaction Execution and Conflict

Logical Status Interpretation

To achieve isolation in transaction executions, a write operation needs to buffer or “hide” its update until the transaction commits; and to achieve atomicity it needs to revoke its modifications upon transaction abort. In the context of data structure transactions, existing strategies undo the operations by invoking their inverse operations [52]. This would incur a significant penalty because the compute cycles spent on the inverse operations do not contribute to the overall throughput and

introduce additional contention. I approach the recovery task from another angle: an aborted transaction does not need to physically invoke the inverse methods; executed operations in an aborted transaction just need to *appear to have been undone*. I achieve this by having operations *inversely interpret* the logical status of nodes accessed by operations in an aborted transaction. Both physical undo and my logical undo reach the same goal of restoring the abstract state of the data structure.

Algorithm 8 Logical Status

```

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

```

In Algorithm 8, I list the function to interpret the logical status of a node according to the value of the transaction descriptor. Function ISNODEPRESENT, verifies that a node associated with a specific key is present. This is a common test found in existing linked data structures. Given a node's presence, function ISKEYPRESENT verifies if the key should be logically included in the abstract state, and returns a boolean value based on the combination of operation type and transaction status. For a node most recently accessed by an INSERT operation, its key is considered present if the transaction has successfully *committed*. On the contrary, according to the semantics of DELETE, a successful operation must remove the key from the set. Thus for a node most recently accessed by a DELETE operation, its key is considered present if the transaction has *aborted*. These two opposite interpretations also match previous observations that INSERT and DELETE are a pair

of inverse operations [52]. Since the FIND operation is read-only, no rollback is needed. The node's key is always present regardless the status of the transaction. A special case is the operations in an active transaction, which I treat as committed but are visible only to subsequent operations within the same transaction scope.

Algorithm 9 Update NodeInfo

```

1: function UPDATEINFO(Node* n, NodeInfo* info, bool, wantkey)
2:   NodeInfo* oldinfo ← n.info
3:   if ISMARKED(oldinfo) then
4:     DO_DELETE(n)
5:     return retry
6:   if oldinfo.desc ≠ info.desc then
7:     EXECUTEOPS(oldinfo.desc, oldinfo.opid + 1)
8:   else if oldinfo.opid ≥ info.opid then
9:     return success
10:  bool haskey ← ISKEYPRESENT(oldinfo)
11:  if (!haskey and wantkey) or (haskey and !wantkey) then
12:    return fail
13:  if info.desc.status ≠ Active then
14:    return fail
15:  if CAS(&n.info, oldinfo, info) then
16:    return success
17:  else
18:    return retry

```

Logical Status Update

As mentioned above, the logical status of a node depends on the interpretation of its transaction descriptor. In a transformed data structure, an operation needs to change a node's logical status before performing any necessary low-level node manipulations. This is done by updating the node's NODEINFO pointer as shown in Algorithm 9. Given a node *n*, the function UPDATEINFO reads its current *info* field (line 9.2, verifies its sanity, and attempts to update *n.info* through the use of CAS (line 9.15). It returns a tri-state value indicating whether the operation succeeded,

failed, or should be retried. I make sure that any other active transaction accessing n is completed by helping execute its remaining operations (line 9.7). However, it has to avoid helping the same transaction because of the hazard of infinite recursions. This is prevented by the condition check on line 9.6. It also skips the update and reports success if the operation has already been performed by other threads (line 9.8). Due to the use of the helping mechanism, the same operation may be executed multiple times by different threads. The condition check on line 9.8 allows us to identify the node accessed by threads that execute the same transaction, and ensures consistent results. It validates the presence of the key on line 9.10 and test if the key's presence (as required by deletions and finds), or lack of presence (as required by insertions), is desired on line 9.11. The boolean flag *wantkey* is passed on by the caller function to indicate if the presence of key is desired. The operation reports failure when *wantkey* contradicts *haskey*. Finally, it validates that the transaction is still active (line 9.13) to prevent a terminated transaction from erroneously overwriting *n.info*.

Transaction Execution

I consider the transaction execution model in which a transaction explicitly aborts upon the first operation failure. The EXECUTETRANSACTION in Algorithm 10 is the entry point of transaction execution, which is invoked by the thread that initiates a transaction. The EXECUTEOPS function executes operations in sequence starting from *opid*. For threads that help to execute a delayed transaction, the *opid* could be in the range of $[1, desc.size]$. In each step of the while loop (line 10.13), the return value of the previous operation is verified. It requires the operations to return a boolean value indicating if the executions are successful. A false value indicates the precondition required by the operation is unsatisfied and the transaction will abort. Once all operations complete successfully it atomically updates the transaction status with a *Committed* flag (line 10.26). It is not necessary to retry this CAS operation as a failed CAS indicates that some other thread must have

successfully updated the transaction status. The thread that successfully executed the CAS will be responsible for performing physical node deletion (line 10.27).

Algorithm 10 Transaction Execution

```

1: thread local Stack helpstack
2: function EXECUTETRANSACTION(Desc* desc)
3:   helpstack.INIT()
4:   EXECUTEOPS(desc, 0)
5:   return desc.status = Committed

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

```

By adopting cooperative transaction execution, my approach is able to eliminate the majority of aborts caused by access conflicts. Although rare, potential livelock is possible if two threads were to access two of the same nodes in opposite order. In such cases, both threads will be trapped

in infinite recursions helping execute each other's transaction. It detects and recovers from this hazard by using a per-thread *help stack*, which is a simple stack containing *Desc* pointers. This is similar to a function call stack, except it records the invocation of EXECUTEOPS. A thread initializes its help stack before initiating a transaction. Each time a thread begins to help another transaction, it pushes the transaction descriptor onto its help stack. A thread pops its help stack once the help completes. Cyclic dependencies among transactions can be detected by checking for duplicate entries in the help stack (line 10.9). It recovers by aborting one of the transactions as shown on line 10.10.

Multi-dimensional Linked List

The performance of the transactional data structures obtained by the above mentioned two strategies will only be as good as the performance of the base data structure that one starts with. Although many concurrent sets and maps based on binary search trees (BST) and skiplists have been proposed [35, 29], their performance is limited by the inherent bottleneck of disjoint access parallelism in BSTs and skiplists. In this section, I present a high-performance linearizable lock-free dictionary design based on a multi-dimensional list (MDList). A node in an MDList arranges its child nodes by their dimensionality and order them by coordinate prefixes. The search operation works by first generating a one-to-one mapping from the scalar keys to a high-dimensional vectors space, then uniquely locating the target position by using the vector as coordinates. My algorithm guarantees worst-case search time of $\mathcal{O}(\log N)$ where N is the size of key space. Moreover, the ordering property of the data structure is readily maintained during mutations without rebalancing nor randomization.

Motivation

Binary search trees are among the most ubiquitous sequential data structures. Despite recent research efforts, designing a self-balancing non-blocking BST is challenging and remains an active topic [13, 76, 28]. One difficulty is to devise a correct and scalable design for predecessor query, which serves as the subroutine for all three operations to locate the physical node containing the target key. When executing the predecessor query concurrently with write operations, the physical location of the target key in the tree might have changed before the search finishes. This is especially troublesome when a predecessor query fails to reach the target node, under which circumstance it has to decide whether the target element is absent, or the target element's physical location has been changed by some concurrent updates [27]. The problem stems from the lack of one-to-one mapping between the *logical ordering* of keys and the physical layout of a BST. For example, the BST in Figure 4a and 4b differ in layout but represent the same ordering for integers $\{1, 6, 7\}$. If a predecessor query looking for node 6 in tree (a) gets suspended when examining node 7 and another thread transforms the tree into (b) by deleting node 4, the resumed operation would falsely conclude that node 6 does not exist. Well-orchestrated synchronization techniques, such using a leaf-oriented BST [11], and embedding logical ordering [27], have been proposed to address the issue, but they pose additional space or time overhead. Another difficulty is to cope with sequential bottleneck of rebalancing. BSTs provide logarithmic access time complexity when they are height balanced. Rebalancing is triggered to maintain this invariant immediately after the height difference exceeds a certain threshold. For concurrent accesses by a large number of threads, frequent restructuring induces contention. Mutating operations need to acquire not only exclusive access to a number of nodes to guarantee the atomicity of their change, but also locks to ensure that no other mutation affects the balance condition before the proper balance is restored [11]. Relaxed balance [11] and lazy rebalancing [18] have been suggested to alleviate contention in lock-based BSTs, but designing efficient lock-free rebalancing operations remains an open topic.

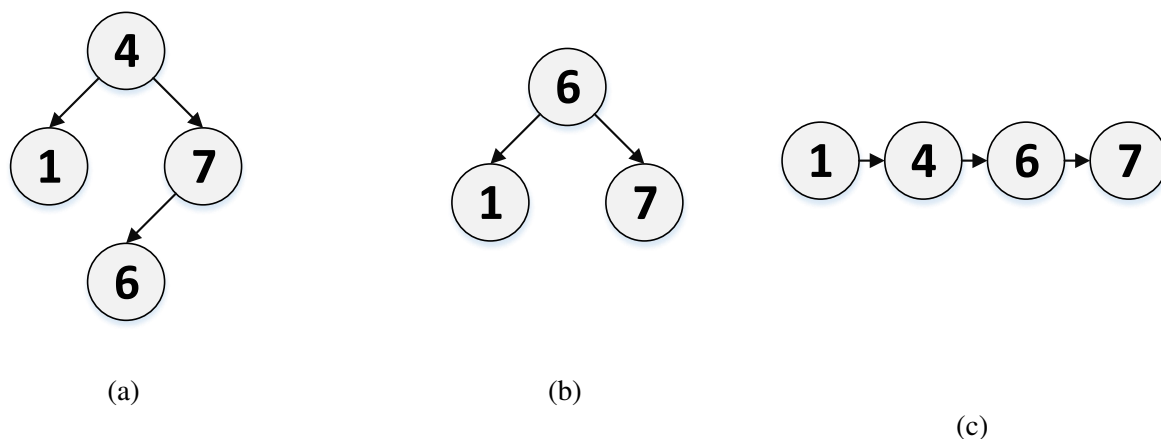


Figure 4: BSTs have various layouts for the same logical ordering (a and b). The linked list (c) has deterministic layout that is independent of execution histories.

In recent research studies [65, 94, 35], non-blocking dictionaries based on skiplists are gaining momentum. A skiplist [83] is a linked list that provides a probabilistic alternative to search trees with logarithmic sequential search time on average. It eliminates the need of rebalancing by using several linked lists, organized into multiple levels, where each list skips a few elements. Links in the upper levels are created with exponentially smaller probability. Skiplist-based concurrent dictionaries have a distributed memory structure that allows concurrent accesses to different parts of the data structure efficiently with low contention. However, INSERT and DELETE operations on skiplists involve updating shortcut links in distant nodes, which incurs unnecessary data dependencies among concurrent operations and limits the overall throughput. Besides, due to the nature of randomization, skiplists may exhibit less than ideal linear worst-case search time.

Overview

I present a linearizable lock-free dictionary implementation based on a multi-dimensional list (MDList) using single-word COMPAREANDSWAP (CAS) primitives. An MDList [97, 96] stores ordered key-value pairs in nodes and guarantees worst-case sequential search time of $\mathcal{O}(\log N)$ where N is the size of the key universe. The search works by injectively mapping a scalar key into a high dimensional vector space, then uniquely locating the target node using the vector as coordinates. The dimensionality D of an MDList is defined as the dimensionality of its vector coordinates. A node in an MDList shares a coordinate prefix with its parent node. The search is done through prefix matching rather than key comparison. Unlike previous prefix-based search data structures [81, 79], an MDList partitions the key universe in a way such that 1) the nodes sharing a common coordinate prefix form a sub-list; and 2) a node store links to at most D sub-lists arranged by the length of their shared coordinate prefixes. As a result, an MDList provides efficient concurrent accesses to different partition of the data structure, and its ordering invariant is readily maintained during insertions and deletions without rebalancing nor randomization. The proposed dictionary has the following algorithmic characteristics that aim to further improve the throughput over existing approaches by exploiting a greater level of parallelism and reducing contention among concurrent operations.

- Nodes are ordered by coordinate prefix, which eliminates the need of rebalancing and randomization.
- Physical layout is deterministic and independent of execution histories, which provides a unique location for each key, and simplifies the FIND algorithm.
- Each INSERT and DELETE operation modifies at most two consecutive nodes, which allows concurrent updates to be executed with minimal interference.

Definition

The core idea of a multi-dimensional list is to partition a linked list into shorter lists and rearrange them in a multi-dimensional space to facilitate search. Just like a point in a D -dimensional space, a node in a D -dimensional list can be located by a D -dimensional coordinate vector. The search operation examines one coordinate at a time and locates correspondent partitions by traversing nodes that belong to each dimension. The search time is bounded by the dimensionality of the data structure and logarithmic time is achieved by choosing D to be a logarithm of the key range.

Definition 1. *A D -dimensional list is a rooted tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, and each child is assigned a unique dimension of $d' \in [d, D)$.*

In an ordered multi-dimensional list, I associate every node with a coordinate vector \mathbf{k} , and determine the order among nodes lexicographically based on \mathbf{k} . A dimension d node share a coordinate prefix of length d with its parent. The following requirement prescribes the exact arrangement of nodes according to their coordinates.

Definition 2. *Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, \dots, k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, \dots, k'_{D-1})$ in an ordered D -dimensional list: $k_i = k'_i, \forall i \in [0, d) \wedge k_d > k'_d$.*

The search operation examines one coordinate at a time and locates correspondent partitions by traversing nodes that belong to each dimension. The search time is bounded by the dimensionality of the data structure and logarithmic search time is achieved by choosing D to be a logarithm of the key range. To map a scalar key to a high-dimensional vector, one can choose any injective and monotonic function. In this paper, I employ KEYTOCOORD, a simple mapping function based on numeric base conversion [97]. This function maps keys uniformly to the vector space, which opti-

mizes the average search path length for random inputs. For a key in range $[0, N)$, KEYTOCOORD converts it to the base- $\lceil \sqrt[D]{N} \rceil$ representation, and treats each digit as one coordinate. For example, given key 1234, when $N = 2^{32}$ and $D = 8$ we have $(1234)_{10} = (4D2)_{16}$. Thus the key 1234 will be mapped to vector $(0,0,0,0,0,4,D,2)$.

Algorithm 11 Lock-free Dictionary Data Structure

1: class Dictionary 2: const int D 3: const int N 4: Node* $head$ 5: struct Node 6: int $key, k[D]$ 7: void* val	8: Node* $child[D]$ 9: AdoptDesc* $adesc$ 10: struct AdoptDesc 11: Node* $curr$ 12: int dp 13: int dc
---	--

Algorithm 12 Pointer Marking

1: int $F_{adp} \leftarrow 0 \times 1, F_{del} \leftarrow 0 \times 2, F_{all} \leftarrow F_{adp} F_{del}$ 2: define SetMark $p, m (p m)$ 3: define ClearMark $p, m (p \& \sim m)$ 4: define IsMarked $p, m (p \& m)$

Data Types

I define the structure of the concurrent dictionary and the auxiliary descriptor object in Algorithm 11. The dimension of the dictionary is denoted by a constant integer D and the range of the keys by N . A node contains a key-value pair, an array $k[D]$ of integers that caches the coordinate vector to avoid repetitive computation, a descriptor for the child adoption process (detailed in Section 3), and an array of child pointers in which the d th pointer links to a dimension d child node. A descriptor [57] is an object that stores operation context used by helping threads to finish a delayed operation. For a dimension d node, only the indices in $[d, D)$ of its child array are valid and the rest

are unused³. The dictionary is initialized with a dummy head node, which has the minimal key 0. I employ the pointer marking technique [46] to mark adopted child nodes as well as logically deleted nodes. The macros for pointer marking are defined in Algorithm 12. F_{adp} and F_{del} flags are co-located with the *child* pointers.

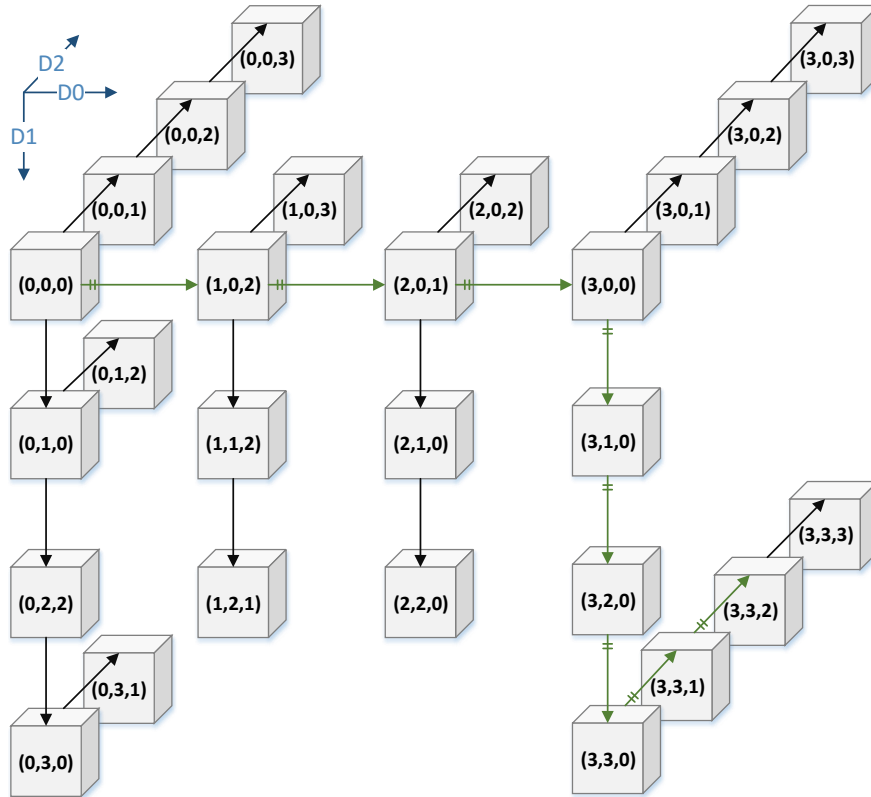


Figure 5: FIND operation in a 3DList ($D = 3, N = 64$)

Concurrent Find

I illustrate the FIND operation on an example 3DList in Figure 5. To locate the node with key 62 (coordinates (3,3,2)), the FIND operation traverse 3 sub-lists following the path highlighted by the

³Since nodes of higher dimensions have less children, for a d dimension node it is possible to allocate a child array of size d to reduce memory consumption. In this paper, I demonstrate the use of constant size child array for simplicity.

green arrows. The worst-case time complexity of the search algorithm is $\mathcal{O}(D \cdot M)$ where M is the maximum number of nodes in each dimension. If we use previously described KEYTOCOORD to uniformly map the keys into the D -dimensional vectors, M is bounded by $\sqrt[D]{N}$. This gives $\mathcal{O}(D \cdot \sqrt[D]{N})$, which is equivalent to $\mathcal{O}(\log N)$, if we choose $D \propto \log N$ (Note that $\log \sqrt[D]{N} = 2$).

Algorithm 13 Concurrent Find

```

1: function FIND(int key)
2:   Node* curr, pred
3:   int dp, dc
4:   pred  $\leftarrow$  NIL, curr  $\leftarrow$  head, dp  $\leftarrow$  0, dc  $\leftarrow$  0
5:   LOCATEPRED(KEYTOCOORD(key))
6:   if dc =  $D$  then
7:     return curr.val
8:   else
9:     return NIL

```

Algorithm 14 Predecessor Query

```

1: inline function LOCATEPRED(int k[ ])
2:   while dc <  $D$  do
3:     while curr  $\neq$  NIL and k[dc] > curr.k[dc] do
4:       pred  $\leftarrow$  curr, dp  $\leftarrow$  dc
5:       ad  $\leftarrow$  curr.adesc
6:       if ad  $\neq$  NIL and dp  $\in$  [ad.dp, ad.dc] then
7:         FINISHINSERTING(curr, ad)
8:         curr  $\leftarrow$  CLEARMARK(curr.child[dc],  $F_{all}$ )
9:       if curr = NIL or k[dc] < curr.k[dc] then
10:        break
11:      else
12:        dc  $\leftarrow$  dc + 1

```

I list the concurrent FIND function in Algorithm 13. The search begins from the head (line 13.4). It then invokes LOCATEPRED listed in Algorithm 14 to perform the predecessor query. The LOCATEPRED function is an extension of the sequential MDList search function [97]. Given a coordinate vector k , it tries to determine its immediate parent $pred$ and child $curr$. In the case that the node with the target coordinates already exists, $curr$ points to the node itself. Together with the dimension variables dp and dc , they amount to the context for inserting or deleting nodes. On

line 14.7, prior to reading *curr*'s child LOCATEPRED helps finish any child adoption tasks in order to acquire the up-to-date values. Since a child adoption process updates the children indexed from *dp* to *dc*, the function must help *curr* node if it intends to read the child node in this range (line 14.6). The helping task does not alter already traversed nodes, so the search process can continue without restart.

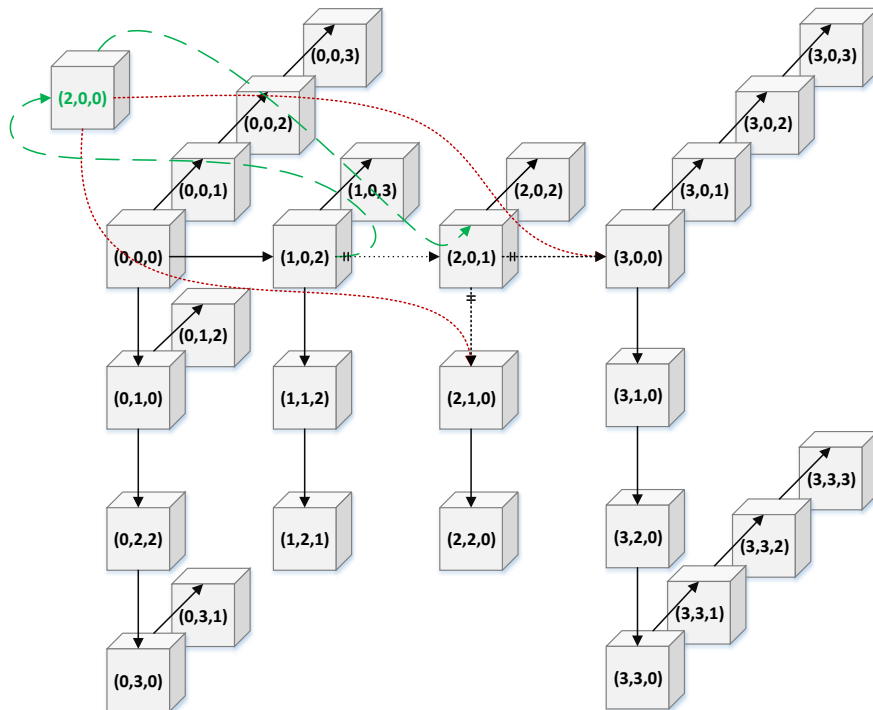


Figure 6: INSERT operation in a 3DList ($D = 3, N = 64$)

Concurrent Insert

The lock-free INSERT operation takes one or two steps and updates no more than two consecutive nodes. Figure 6 illustrates the insertion of key 32 (coordinates (2,0,0)). In the first step, the process updates the predecessor node: the new node is spliced with its predecessor on dimension 0 (as marked the by green arrows), and the old child of the predecessor becomes a dimension 2 child of the new node. In the second step, the process updates the successor node if its dimensionality

has changed during the first step. In this case, the new node adopts child nodes of its successor between dimension $[0, 2)$ (as marked by the red arrows). I guarantee lock-free progress in the node splicing step by atomically updating predecessor's child pointer using CAS [46]. To provide for lock-free progress in the child adoption step, it needs to announce the operation globally using descriptor objects [57]. This allows the interrupting threads to help finish the operation in case the insertion thread is preempted.

I list the concurrent INSERT function in Algorithm 15. After locating the target position (line 15.12), the function updates the child pointer of the predecessor node (line 15.21). The dimension of the node being inserted is kept in dp and the dimension of the child in dc (line 14.4 and 14.12). The new node should be inserted at the dimension dp child of the $pred$ node, while a non-empty $curr$ node will become the dimension dc child of the new node. The code between lines 15.13 and 15.15 reads the $adesc$ fields from $curr$ and tests if helping is needed. Like in LOCATEPRED, the insertion need to help $curr$ node if it is going to adopt children from $curr$ node.

Prior to atomically updating the link to the new node, it fills the remaining fields of the new node (line 15.27 and 15.35). If the new node needs to adopt children from $curr$ node, I use an adopt descriptor to store the necessary context (line 15.30). The pointers within the range $[0, dp)$ of the new node's child array are marked with F_{adp} . This effectively invalidates these positions for future insertions. The pointers within the range $[dp, D]$ are set to NIL meaning they are available for attaching child nodes. $curr$ node is conditionally linked to the new node on line 15.34. dc can be set to D on either line 15.19 or line 14.12. In the first case, $curr$ must be logically deleted, and the new node is immediately in front of $curr$. By not linking it with the new node, it effectively discards $curr$. In the second case, the new node has the same key as $curr$ and it essentially updates the associated value by replacing $curr$ with the new node. On line 15.21, the single-word CAS atomic synchronization primitive is used to update the $pred$'s child pointer. The CAS would fail under three circumstances: 1) the desired child slot has been updated by another competing

insertion; 2) the desired child slot has been invalidated by a child adoption process; and 3) the desired child slot has been marked for logical deletion. If any of the above cases is true, the loop restarts. Otherwise, the insertion proceeds to finish its own child adoption process.

Algorithm 15 Concurrent Insert

```

1: function INSERT(int key, void* val)
2:   Node* node ▷ the new node
3:   Node* pred, curr ▷ new node's parent and child
4:   int dp, dc ▷ dimension of node in pred and curr
5:   AdoptDesc* ad ▷ descriptor for child adoption task
6:   node ← new Node
7:   node.key ← key, node.val ← val
8:   node.k[0 : D] ← KEYTOCOORD(key)[0 : D]
9:   node.child[0 : D] ← NIL
10:  while true do
11:    pred ← NIL, curr ← head, dp ← 0, dc ← 0
12:    LOCATEPRED(node.k)
13:    ad ← curr ≠ NIL ? curr.adesc : NIL
14:    if ad ≠ NIL and dp ≠ dc then
15:      FINISHINSERTING(curr, ad)
16:    if ISMARKED(pred.child[dp],  $F_{del}$ ) then
17:      curr ← SETMARK(curr,  $F_{del}$ )
18:      if dc = D - 1 then
19:        dc ← D
20:      FILLNEWNODE( )
21:      if CAS(&pred.child[dp], curr, node) then
22:        if ad ≠ NIL then
23:          FINISHINSERTING(node, ad)
24:        break
25:
26:  inline function FILLNEWNODE( )
27:    ad ← NIL
28:    if dp ≠ dc then
29:      ad ← new AdoptDesc
30:      ad.curr ← curr, ad.dp ← dp, ad.dc ← dc
31:      node.child[0 : dp] ←  $F_{adp}$ 
32:      node.child[dp : D] ← NIL
33:      if dc < D then
34:        node.child[dc] ← curr
35:      node.adesc ← ad

```

The FINISHINSERTING function in Algorithm 16 performs child adoption on a given node n with the descriptor ad . This is a helping procedure that must correctly handle duplicate and simultaneous executions. The function first reads the adoption context from the descriptor into its local variables. It then transfers $curr$ node's children within the range of $[dp, dc)$ to n . Before a child pointer can be copied, it must be safeguarded so that the value cannot be changed while the copy is in progress. This is done by setting the F_{adp} flag in the child pointers (line 16.5). Once the flag is set, the function proceeds to copy the pointer to n (line 16.8). Finally, the descriptor field in n is cleared to designate the operation's completion.

Algorithm 16 Child Adoption

```

1: function FINISHINSERTING(Node*  $n$ , AdoptDesc*  $ad$ )
2:   Node*  $child$ ,  $curr \leftarrow ad.curr$ 
3:   int  $dp \leftarrow ad.dp$ ,  $dc \leftarrow ad.dc$ 
4:   for  $i \in [dp, dc)$  do
5:      $child \leftarrow \text{FETCHANDOR}(\&curr.child[i], F_{adp})$ 
6:      $child \leftarrow \text{CLEARMARK}(child, F_{adp})$ 
7:     if  $n.child[i] = \text{NIL}$  then
8:        $\text{CAS}(\&n.child[i], \text{NIL}, child)$ 
9:    $\text{CAS}(\&n.adesc, ad, \text{NIL})$ 

```

Concurrent Delete

The sequential DELETE and INSERT operations on an MDList [97] works reciprocally: the former may promote a node to a lower dimension while the latter may demote a node to a higher dimension. This works well for sequential algorithms, but in concurrent execution where threads help each other, bidirectional change of nodes' dimension incurs contention and synchronization issues. Consider a node n with an active child adoption descriptor (i.e., $n.adesc \neq \text{NIL}$). When concurrency level is high, several threads may read this descriptor and proceed to help finish the adoption by marking some children on node $n.adesc.curr$ as invalid. One of them will eventually succeeds in finish the helping (as observed by setting $n.adesc \leftarrow \text{NIL}$), but I have no way to know

if all threads have finished the helping. If a DELETE operation promotes the node $n.adesc.curr$ by re-enabling the invalid child pointers, an unfinished helping process may erroneously disable them again. Additional synchronization is required to prevent threads from interfering with each other when they perform helping task on the same node. I found the simplest solution is to *keep dimensionality change unidirectional*.

Algorithm 17 Concurrent Delete

```

1: function DELETE(int  $key$ )
2:   Node*  $curr, pred, child, marked$ 
3:   int  $dp, dc$ 
4:   while true do
5:      $pred \leftarrow \text{NIL}, curr \leftarrow head, dp \leftarrow 0, dc \leftarrow 0$ 
6:     LOCATEPRED(KEYTOCOORD( $key$ ))
7:     if  $dc \neq D$  then
8:       return NIL
9:      $marked \leftarrow \text{SETPREMARK}(curr, F_{del})$ 
10:     $child \leftarrow \text{CAS}_{val}(\&pred.child[dp], curr, marked)$ 
11:    if CLEARMARK( $child, F_{del}|F_{adp}$ ) =  $curr$  then
12:      if !ISMARKED( $child, F_{del}|F_{adp}$ ) then
13:        return  $curr.val$ 
14:      else if ISMARKED( $child, F_{del}$ ) then
15:        return NIL

```

My lock-free DELETE operation is thus asymmetrical in the sense that it does not remove any node from the data structure nor alter the nodes' dimensionality. It only marks a node for logical deletion [46], and leaves the execution of physical removal to subsequent INSERT operations. When a new node (n_n) is inserted immediately before a logically deleted node (n_d), n_n expunge n_d from the data structure by skipping n_n and linking directly into all of the child nodes of n_d . Since the physical deletion is embedded in the insertion operation, I reduce the interaction between insertion and deletion operations to a minimal and achieved an overall simple design of lock-free dictionary. This may sound counterintuitive, but the list-like partition strategy of MDList allow us to efficiently discard nodes by simply skipping links. Since a logically deleted node only gets purged when an insertion take place immediately in front of it, there will be a number of zombie nodes. I

thus trade memory consumption for scalability.

In Algorithm 17, the concurrent DELETE operation traverses the dictionary starting from the head looking for target node. It shares the same CAS-based loop as the INSERT function. The process terminates on line 17.8 if it fails to find the target node. Otherwise, it marks the target node for logical deletion using CAS 17.10. A node is considered logically deleted once the pointer in its parent's child array is marked with F_{del} . The CAS_{val} returns the value store on the address before the update. It is considered successful if the return value $child$ is equal to the expected value $curr$, which is detected by the conditional statement on line 17.11 and 17.12. Otherwise, the function checks if the target node has already been marked for deletion by examine the F_{del} flag on $child$ (line 17.14). If so, the function returns. Finally, the child pointer in $pred$ must have been updated by concurrent insertions, DELETE would start anew from the head.

CHAPTER 4: LIBRARY IMPLEMENTATION

Exploring opportunities for future multicore scalability requires fully harnessing potential concurrency that is latent in software interfaces. Recent researches theorize that software systems *can be implemented* in a way that scales when their interface operations commute [16, 52]. This posits an baseline for software scalability. Nevertheless, *how to implement* software systems that exceeds the baseline remains an open research challenge. I adopt a holistic approach for developing transactional data structures — building a reusable open source library named *libtxd*. In this library, I integrated four data structures — a linked list, a skiplist, a hash table, and an MDList each supporting two interfaces (set and map). I also developed a software framework for supporting composable transaction across multiple data structures, and necessary code templates for continuous integration of new data structures.

Interface Design

The lock-free transactional transformation described in Section 3.3 supports transaction execution of operations for a single data structure. The transaction execution and descriptor management is a built-in module of the data structure. This design is shaped by the focus on standalone application — granting users flexibility to obtain self-contained transaction data structure that has minimal external dependencies. To allow for executing transactions that involves operations across multiple data structures, I externalized the transaction execution and create a dedicated framework for transaction descriptor management. Libtxd serves as a unified data structure pool where different instances of data structures have mutual awareness of the existence of each other. I also developed an extended transaction descriptor format that encodes the data structure instances besides operation and operands. All transactional data structures within the library support the same trans-

action descriptor format, which enables co-operative transaction execution across multiple data structures.

Algorithm 18 Libtxd Transaction Descriptor

```
1  enum TxStatus{
2      ACTIVE,
3      COMMITTED,
4      ABORTED
5  };
6
7  struct TxInfo{
8      TxInfo(TxDesc* _desc, uint8_t _opid);
9      TxDesc* desc;
10     uint8_t opid;
11 };
12
13 typedef std::function<void()> TxCall;
14
15 struct TxOp{
16     virtual bool Execute(TxInfo* info) = 0;
17     TxCall onCommit;
18     TxCall onAbort;
19 };
20
21 struct TxDesc{
22     TxDesc(std::initializer_list<TxOp*> _ops)
23     bool Execute();
24     bool Execute(uint8_t opid);
25     std::atomic<TxStatus> status;
26     std::vector<TxOp*> ops;
27 };
```

Unified Transaction Descriptor

I list the extended transaction descriptor in Algorithm 18. It is mostly C++ translation of the pseudo-code listed in Algorithm 6. The major improvement is that I designed an inheritance interface for TXOP using C++ polymorphisms. Instead of giving each operation an type number as in Algorithm 6, I allow users to implement their own operations as long as they fully override the pure virtual EXECUTE method. The advantage an inheritance based design is the ability to

extend the functionality of transaction operations modularly. As I will demonstrate in the SET interface design in Algorithm 20, the new TXOP interface allows me to write as many set operation as needed without them interfering with each other. Whereas in Algorithm 8 and 10, with introduction of each new operation type, care must be take to extend the switch cases. Furthermore, this essentially gives user freedom to implement any operation besides data structure operation. For example, numeric computation can be implement as a valid TXOP so that computation can be done between data structure operations.

Algorithm 19 Libtxd Usage Example

```
1 TxList set1;
2 TxSkiplist set2;
3 TxMDList set3;
4
5 TxDesc* desc = new TxDesc({
6     new Set<int>::InsertOp(3, set1),
7     new Set<int>::DeleteOp(6, set2),
8     new Set<int>::FindOp(5, set1),
9     new Set<int>::InsertOp(7, set3)});
10 desc->Execute();
```

Another improvement is that I extracted the transaction execution function and made it a member function of TXDESC class. The EXECUTE function follows the same flow as Algorithm 10, but now user can execute transaction independently from any instance of data structure. This is important as TXDESC must be logically decouple from data structure instances so that cross-container transactions are possible. Algorithm 19 shows a code snippet to create and execute a transaction across three containers. The construction of the transaction descriptor involves instantiating multiple (in this case, four) operations in one batch. The operations can be in any form as long as they are subclasses of TXOP. Here I show set operation on three different types of containers. The whole transaction will successfully commit if and only if every operation in it succeeds.

Algorithm 20 Libtxd Set Interface

```
1  template<typename T> class Set{
2  public:
3      struct KeyPredicate{
4          virtual bool IsKeyExist(const TxDesc* desc, const TxDesc* obsr)=0;
5      };
6      struct InsertOp : public TxOp, public KeyPredicate{
7          InsertOp(const T& _key, Set& _s);
8          virtual bool Execute(TxInfo* info){
9              return s.Insert(key, info, onAbort);
10         }
11         virtual bool IsKeyExist(const TxDesc* desc, const TxDesc* obsr){
12             return (desc->status == COMMITTED) || (desc->status == ACTIVE
13                 && desc == obsr);
14         }
15         Set& s;
16         T key;
17     };
18     struct DeleteOp : public TxOp, public KeyPredicate
19     {
20         DeleteOp(const T& _key, Set& _s);
21         virtual bool Execute(TxInfo* info){
22             return s.Delete(key, info, onCommit);
23         }
24         virtual bool IsKeyExist(const TxDesc* desc, const TxDesc* obsr){
25             return desc->status == ABORTED;
26         }
27         Set& s;
28         T key;
29     };
30     struct FindOp : public TxOp, public KeyPredicate{
31         FindOp(const T& _key, Set& _s);
32         virtual bool Execute(TxInfo* info){
33             return s.Find(key, info);
34         }
35         virtual bool IsKeyExist(const TxDesc* desc, const TxDesc* obsr){
36             return true;
37         }
38         Set& s;
39         T key;
40     };
41 protected:
42     virtual bool Insert(const T& key, TxInfo* info, TxCall& onAbort) = 0;
43     virtual bool Delete(const T& key, TxInfo* info, TxCall& onCommit) = 0;
44     virtual bool Find(const T& key, TxInfo* info) = 0;
45 };
```

Set Interface

I demonstrate how to design data structure interfaces that is compliant with the LIBTXD descriptor structure using set abstract data type as an example. In Algorithm 20, I list the C++ code for the set data type defined in LIBTXD. SET is the base class for all data structures that implement the set abstract data type. It is advantages to utilize C++ inheritance as it allows me to distill common functionality from all linked data structure and put them in the SET class to reduce code duplication. In practice, subclasses such as a linked list only needs to override the three pure virtual member functions. I cover more details on how to implement these methods using provide code template in the next section. The SET class also contains the definition for three operations that it supports: INSERTOP, DELETEOP, and FINDOP. As mentioned in the previous section, there operations are implemented as subclasses of TXOP and thus provide the actual method body for the EXECUTE method. For example, when the INSERTOP is executed, it invokes the INSERT method on the stored SET instance. Storing reference to the underlying container in the operation structures allows the users to bind different container instances at runtime. This also simplifies transaction execution in TXDESC because there is no need to identify container instances associated with each operation since they are bound together.

Applying MRLock

I listed the auxiliary classes that allow users to synchronize transaction execution of multiple method calls on a single concurrent data structure in Algorithm 21. In order to apply MRLock as the transaction manager for lock-based concurrent data structures, I define a resource to be a key in a set, and each key is dynamically mapped to a bit in a bitset. Prior to the execution of any data structures operations, users determines all the keys that will be access during the transaction and store them in a vector. The vector is passed to RESOURCEALLOCATOR class to be converted

into a lockable object. A lockable object is a class that implements a LOCK and UNLOCK method. The invocation of the LOCK method will be blocked until all required resources have been granted exclusive access. Once the users obtains a lockable object, they invoke LOCK method and proceed with any intended data structure operation as normal. The users then invoke UNLOCK after all operations has finished. The operations executed between the response of LOCK and the invocation of UNLOCK are guaranteed to satisfy atomicity and isolation as MRlock provide mutual exclusion for resources being access during that period.

Algorithm 21 Libtxd MRlock Interface

```

1  class MRlockable
2  {
3  public:
4      MRlockable(const Bitset& _mask, MRlock* _mrlock)
5          : mask(_mask), lockHandle(-1), mrlock(_mrlock){}
6      void Lock(){
7          lockHandle = mrlock->Lock(mask);
8      }
9      void Unlock(){
10         mrlock->Unlock(lockHandle);
11     }
12 private:
13     BitsetType m_resourceMask;
14     int m_lockHandle;
15     MRlock<BitsetType>* mrlock;
16 };
17
18 class ResourceAllocator
19 {
20 public:
21     ResourceAllocator (int numResources){
22         mrlock = new MRlock(numResources);
23     }
24     MRlockable* CreateLockable(const vector<int>& resources){
25         Bitset mask;
26         for (unsigned i = 0; i < resources.size(); i++) {
27             mask.Set(resources[i]);
28         }
29         return new MRlockable(mask, mrlock);
30     }
31 private:
32     MRlock* mrlock;
33 };

```

Applying LFTT

In this section, I demonstrate how to apply LFTT on linked data structures. The process involves two steps: 1) identify and encapsulate the base data structure's methods for locating, inserting, and deleting nodes; and 2) integrate the `UPDATEINFO` function (Algorithm 9) in each operation using the templates provided in this section.

Code Transformation

The first step is necessary because I still rely on the base algorithm and its concurrency control to add, update, and remove linkage among nodes. This is a refactoring process, as I do not alter the functionality of the base implementations. Although implementation details such as argument types and return values may vary, I need to extract the following three functions: `DO_LOCATEPRED`, `DO_INSERT`, and `DO_DELETE`. I add a prefix `DO_` to indicate these are the methods provided by the base data structures. For brevity, I omit detailed code listings, but express the general functionality specifications. Given a key, `DO_LOCATEPRED` returns the target node (and any necessary variables for linking and unlinking a node, e.g., its predecessor). `DO_INSERT` creates the necessary linkage to correctly place the new node in the data structure. `DO_DELETE` removes any references to the node. Note that some lock-free data structures [46, 35] employ a two-phased deletion algorithm, where the actual node removal is delayed or even separated from the first phase of logical deletion. In this case, I only expect `DO_DELETE` to perform the logical deletion as nodes will be physically removed during the next traversal. I also assume there will be sentinel head and tail nodes so that the `DO_LOCATEPRED` function will not return null pointers.

Algorithm 22 Template for Transformed Insert Function

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

Algorithm 23 Template for Transformed Find Function

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

Code Templates

Algorithm 22 lists the template for the transformed INSERT function. The function resembles the base node insertion algorithm with a CAS-based while loop (line 22.4). The only addition is the code path for invoking UPDATEINFO on line 22.7. The logic is simple: on line 22.6 I check if the data structure already contains a node with the target key. If so, I try to update the node's logical

status, otherwise I fall back to the base code path to insert a new node. Should any of the two code paths indicate a retry due to contention, I start the traversal over again.

Algorithm 24 Template for Transformed Delete Function

```

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

16: function MARKDELETE(set delnodes, Desc* desc)
17:   for del  $\in$  delnodes do
18:     if del = NIL then
19:       continue
20:     NodeInfo* info  $\leftarrow$  del.info
21:     if info.desc  $\neq$  desc then
22:       continue
23:     if CAS(del.info, info, SETMARK(info)) then
24:       DO_DELETE(del)

```

The DELETE operation listed in Algorithm 24 is identical to INSERT except it terminates with failure when the target node does not exist (line 24.13). I also adopt a two-phase process for unlinking nodes from the data structure: deleted nodes will firstly be buffered in a local set in EXECUTEOPS, and when the transaction commits, the *info* field of buffered nodes will be marked (line 24.23) and consequently unlinked from the data structure by invoking the base data structures' DO_DELETE. One advantage of my logical status interpretation is that unlinking nodes from the data structure is optional, whereas in transactional boosting nodes must be physically unlinked to restore the

abstract state. This opens up opportunities to optimize performance based on application scenarios. Timely unlinking deleted nodes is important for linked lists because the number of “zombie” nodes has linear impact on sequential search time. Leaving delete nodes in the data structure may be beneficial for skiplists because the overhead and contention introduced by unlinking nodes may outweigh the slight increase in sequential search time ($\mathcal{O}(\log n)$).

The FIND operation listed in Algorithm 23 also needs to update the node’s *info* pointer. Without this, concurrent deletions may remove the node after FIND has returned and before the transaction commits. Since I have extracted the core functionality of interpreting and updating logical status into a common subroutine, the transformation process is generic and straightforward. To use a transformed data structure, the application should first initialize and fill a DESC structure, then invoke EXECUTETRANSACTION (Algorithm 10) with it as an argument. The allocation of the transaction descriptor contributes to most of the transaction execution overhead.

CHAPTER 5: EXPERIMENTAL EVALUATION

All tests are conducted on a 64-core ThinkMate RAX QS5-4410 server running Ubuntu 12.04 LTS. It is a NUMA system with four AMD Opteron 6272 CPUs (16 cores per chip @2.1 GHz) and 64 GB of shared memory (16 × 4GB PC3-12800 DIMM). Both the micro-benchmark and the lock implementations are compiled with GCC 4.7 (with the options `-O1 -std=c++0x` to enable level 1 optimization and C++ 11 support).

Lock-based Transactions

In this section, I assess the overhead, scalability and performance consistency of my multi-resource lock (MRLock) and compare it with the `std::lock` function from GCC 4.7 (STDLock), the `boost::lock` function from Boost library 1.49 (BSTLock), the resource hierarchy protocol combined with both `std::mutex` (RHSTD) and `tbb::queue_mutex` from Intel TBB library 4.1 (RHQueue), and the extended TATAS lock (ETATAS) introduced in Algorithm 1. TBB's queue-based mutex is chosen as a representative queue lock implementation, and when combined with the resource hierarchy protocol it is the state-of-the-art resource allocation solution, while the Standard library and Boost library are two of the most widely used C++ libraries in practice.

Experiment Setup

Both `std::lock` and `boost::lock` implement a two-phase locking protocol to acquire multiple locks, but they have slightly different requirements for the input parameters: Boost's version accepts a range of lock iterators; the standard library's version, which takes advantage of the C++ 11 variadic templates, accepts a variable number of locks as function arguments. I im-

plement an interface adapter that encapsulates these differences. I use `std::mutex` as the underlying lockable object for `std::lock`, and `boost::mutex` for `boost::lock`. For the resource hierarchy protocol, I use two underlying mutex implementations: `std::mutex` and `tbb::queue_mutex`. ETATAS is implemented based on the standard `std::atomic_compare_exchange` provided by GCC.

I employ the same micro-benchmark suggested by Scott et al. [89] to evaluate the performance of these approaches for multiple resource allocation. It consists of a tight loop that acquires and releases a predetermined number of locks. The loop increments a set of integer counters, where each counter represents a resource. The counters are not atomic, so without the use locks their value will be incorrect due to data races. When the micro-benchmark's execution is complete, I check each counter's value to verify the absence of data races and validate the correctness of my lock implementations. Algorithm 25 lists the benchmark function and related parameters.

Algorithm 25 Micro-Benchmark

```
1: function MAIN()
2:   threads = CreateThreads(TestLock, n)
3:   WaitForBarrier()
4:   BeginTimer()
5:   WaitForThreads(threads)
6:   EndTimer()

7: function TESTLOCK(lock, resource, contention, iteration)
8:   requested = Random(resources, contention)
9:   WaitForBarrier()
10:  for 1  $\rightarrow$  iteration do
11:    lock.Acquire(requested)
12:    requested.IncreaseCount()
13:    lock.Release(requested)
```

When evaluating classic mutual exclusion locks, one may increase the number of concurrent threads to investigate their scalability. Since all threads contend for a single critical section, the contention level scales linearly with the number of threads. However, the amount of contention in

the resource allocation problem can be raised by either increasing the number of threads or the size of the resource request per thread. Given k total resources with each thread requesting h of them, I denote the *resource contention* by the fraction h/k or its quotient in percentage. This notation reveals that *resource contention* may be comparable even though the total number of resources is different. For example, $8/64$ or 12.5% means each request needs 8 resources out of 64, which produces about the same amount of contention as $4/32$. I show benchmark timing results in Section 5 that verifies this hypothesis. The product of the thread number p and *resource contention* level roughly represents the overall contention level.

To fully understand the efficiency and scalability in these two dimensions, I test the locks in a wide range of parameter combinations: for thread number $2 \leq p \leq 64$ and for resource number $4 \leq k \leq 1024$ each thread requests the same number of resources $2 \leq h \leq k$. I set the loop iteration in the micro-benchmark to 10,000 and get the average time out of 10 runs for each configuration.

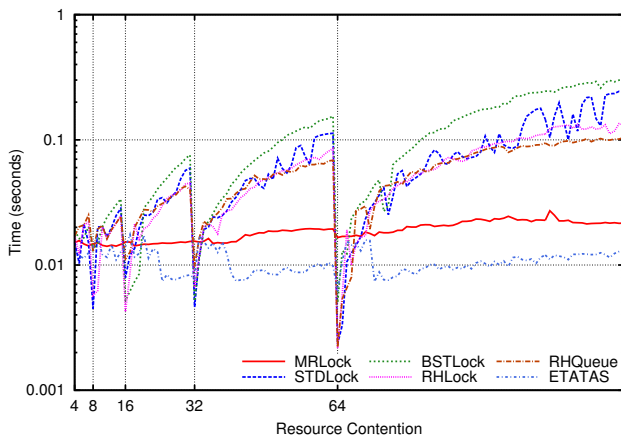
Single-thread Overhead

To measure the lock overhead in the absence of contention, I run the micro-benchmark with a single thread requesting two resources and subtract the loop overhead from the results. Table 2 shows the total timing for the completion of a pair of lock and unlock operations. In this scenario MRLock is twice as fast as the two phase locks and the resource hierarchy locks. MRLock process the two lock requests in one batch, and without obstruction the request handling process only takes one CAS operation to complete, which brings the overhead of MRLock down to the same level as a Standard or Boost mutex. The alternatives take about twice the time of MRLock because each of them need to process two mutex locking operations in order to solve this non-trivial resource allocation problem. MRLock is slightly slower than ETATAS because of the extra queue traversing operation. Although `std::mutex` and `boost::mutex` does not solve the resource allocation

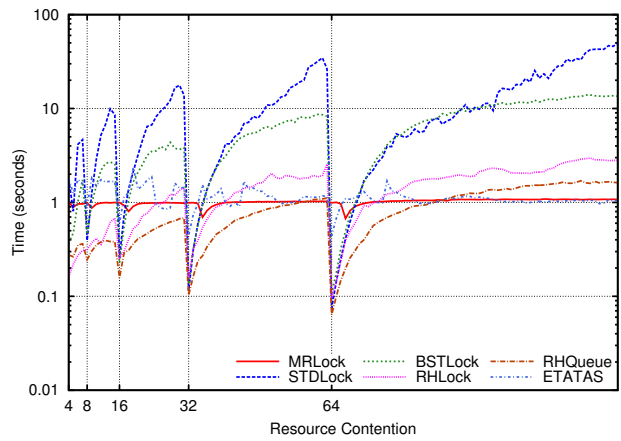
problem, I present them as a baseline performance metric.

Table 2: Lock overhead obtained without contention

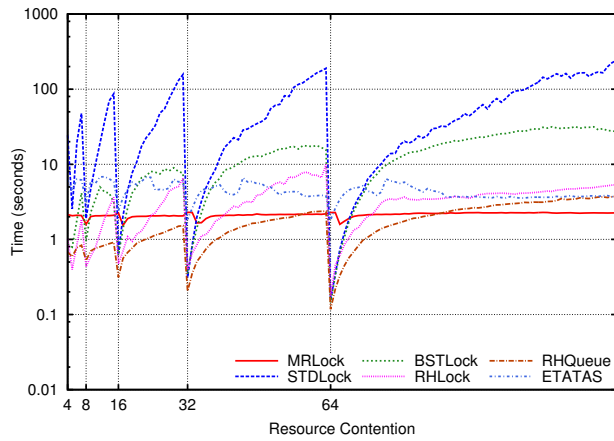
MRLock	STDLock	BSTLock	RHLock	RHQueue	ETATAS	std::mutex boost::mutex
42ns	95ns	105ns	88ns	90ns	34ns	35ns



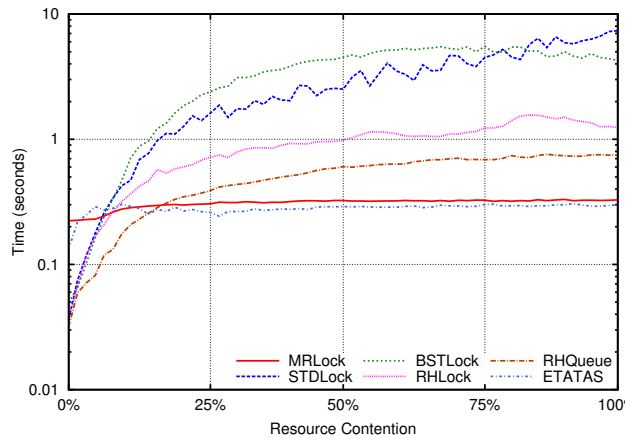
(a) 2 threads



(b) 32 threads



(c) 64 threads



(d) 16 threads with 64 resources

Figure 7: Contention scaling up to 64 resources

Resource Scalability

My performance evaluation exploring the scalability of the tested approaches when increasing the level of *resource contention* is shown in Figures 7a, 7b, and 7c. The *y*-axis represents the total time needed to complete the micro-benchmark in a logarithmic scale, where a smaller value indicates better performance. The *x*-axis represents the level of resource contention. Each tick mark on the *x*-axis represents the beginning of the section to its right, and the tick mark label denotes the total number of resources in that section. For example, the section between Ticks 32 and 64 represents the executions with 32 resources, while the section to the right of Tick 64 represents executions with 64 resources. Within each section, the level of contention increases from 1% to 100%. We observe a saw pattern because the resource contention level alternates regularly as we move along the *x* axis. In addition, we observe that the timing pattern is similar among different sections, supporting my argument that the level of resource contention is proportional to the quotient of the request size divided by total number of resources. I also show a zoomed-in view of a single section in Figure 7d, which illustrates the timings of 16 threads contending for 64 resources. In this graph, I mark the *x*-axis by contention level.

When increasing the number of requested resources per thread, the probability of threads requesting the same resources increases. This poses scalability challenges for both two-phase locks and the resource hierarchy implementations because they rely on a certain protocol to acquire the requested locks one by one. As the request size scales up, the acquiring protocol is prolonged thus prone to failure and retry. At high levels of contention, such as the case with 64 threads (Figure 7c) when the level contention exceeds 75%, `STDLock` is more than 50 times slower when compared to `MRLock`. `BSTLock` exhibits the same problem, and its observed performance closely resembles that of `STDLock`'s. Unlike the above two methods, `RHLock` acquires locks in a fixed order, and it does not release current locks if a required resource is not available. This hold-and-wait paradigm

helps stabilize the timings and reduce the overall contention. RHLock resembles the performance of STDLock in the two thread scenario (Figure 7a), but it outperforms both BSTLock and STDLock by about three times under 50% resource contention on 16 threads (Figure 7d). RHQueue achieves the best performance among the alternative approaches, due relieved contention introduced by the queue-based mutex.

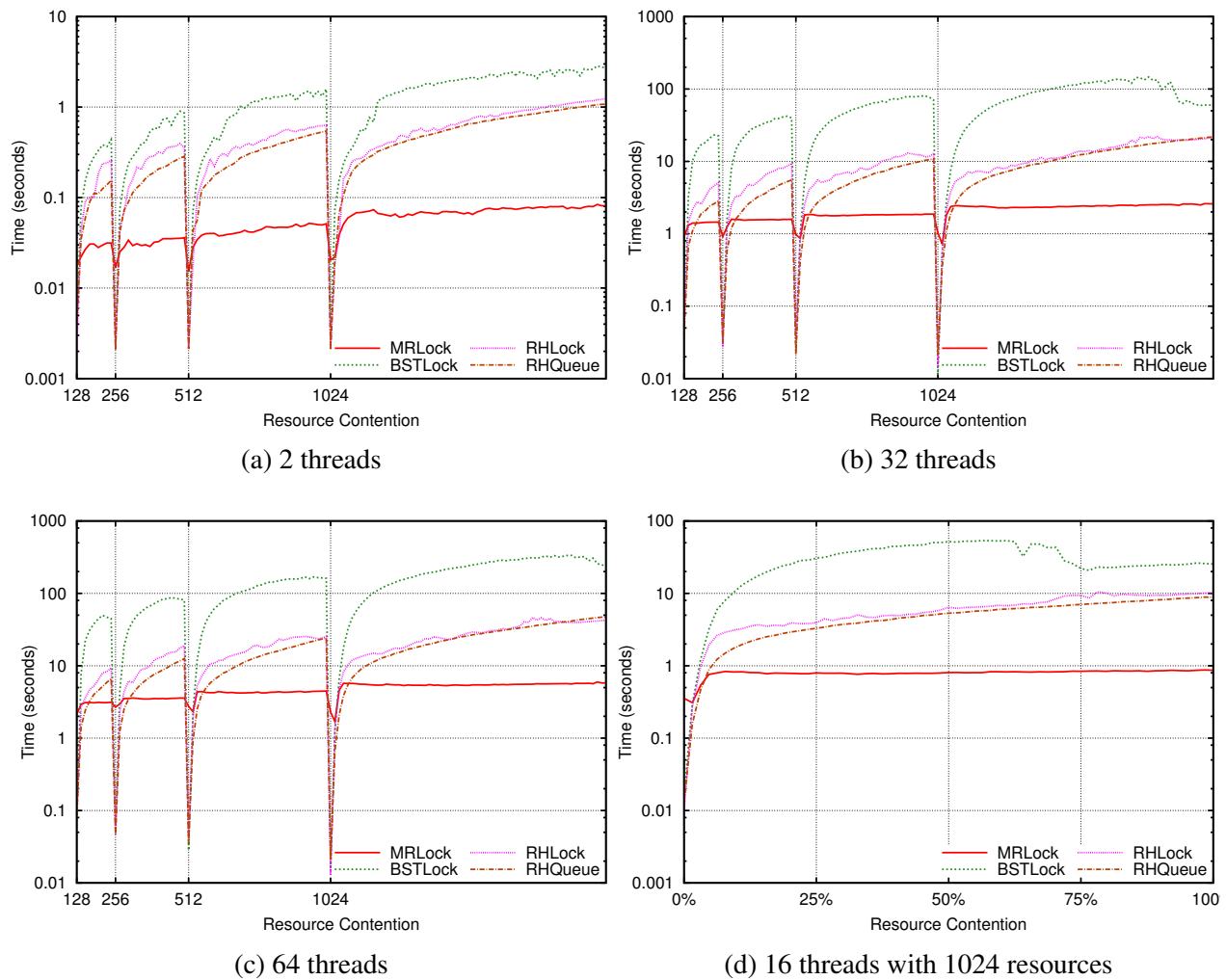


Figure 8: Contention scaling up to 1024 resources

While the time of all alternative methods show linear growth with respect to resource contention,

MRLock remains constant throughout all scenarios. In the case of 64 threads and contention level 32/64, MRLock achieves a 20 times speed-up over STDLock, 10 times performance gain over BSTLock, 2.5 times performance increase over RHLock, and is 30% faster than RHQueue. The fact that MRLock provides a centralized manager to respond the lock requests from threads in one batch contributes to this high degree of scalability. ETATAS also adopts the same all-or-nothing protocol, thus it could be seen as an MRLock algorithm with a buffer size of one. It outperforms MRLock on two threads by about 40% (Figure 7a), and almost ties with MRLock on 32 threads. However, MRLock is 1.7 time faster on 64 threads than ETATAS, because the queuing mechanism relieves the contention of the CAS loop. In Figure 7d, we see that under low levels of contention (less than 10% or 7/64), MRLock and ETATAS no longer hold an advantage over the resource hierarchy locks, and the two-phase locks. For example, STDLock takes only 0.003s under contention 2/64 on 64 threads, while MRLock takes 0.016s. When the resource contention level is low, the lock requests are likely to spread out among the pool of resources. The locking acquiring process using resource hierarchy or two phase locking are not likely to encounter any conflicts, and may proceed optimistically, but for MRLock every thread has to enqueue its requests even though they might not conflict with any outstanding request.

Figures 8a, 8b, 8c, and 8d show the time to complete the same benchmark with up to 1024 resources. I exclude ETATAS from the comparison because it dose not support more than 64 resources as explained in Section 2. I also exclude STDLock from this test because of the compiler constraint on the total number of arguments allowed in the variadic template. We observe that MR-Lock exhibits clear performance advantage over the alternatives. In 2 treads scenario (Figure 8a) for 1024 resources, MRLock obtains over ten times speed-up against the RHQueue which is the best alternative, and it maintains an average speed-up of eight times for 32 threads (Figure 8b) and 64 threads (Figure 8c) scenarios. As shown in Figure 8d, MRLock still maintains a roughly constant execution time on all levels of contention, but I also notice a slight increase in time when

increasing the total number of resources in Figures 8b, and 8c. The execution time of MRLock forms a series of steps in these two graphs. This is mainly because of the extra time spent on reading and writing the resource request bitsets with increased length. Also notice that as the total number of resources increases, the advantage of RHQueue over RHLock diminishes. For example, in Figure 8c RHQueue almost has the identical performance as RHLock in the region of 1024 resources. This is an indication that at larger scale the contention among the resources outweighs the benefit of applying queuing strategy on a single resource.

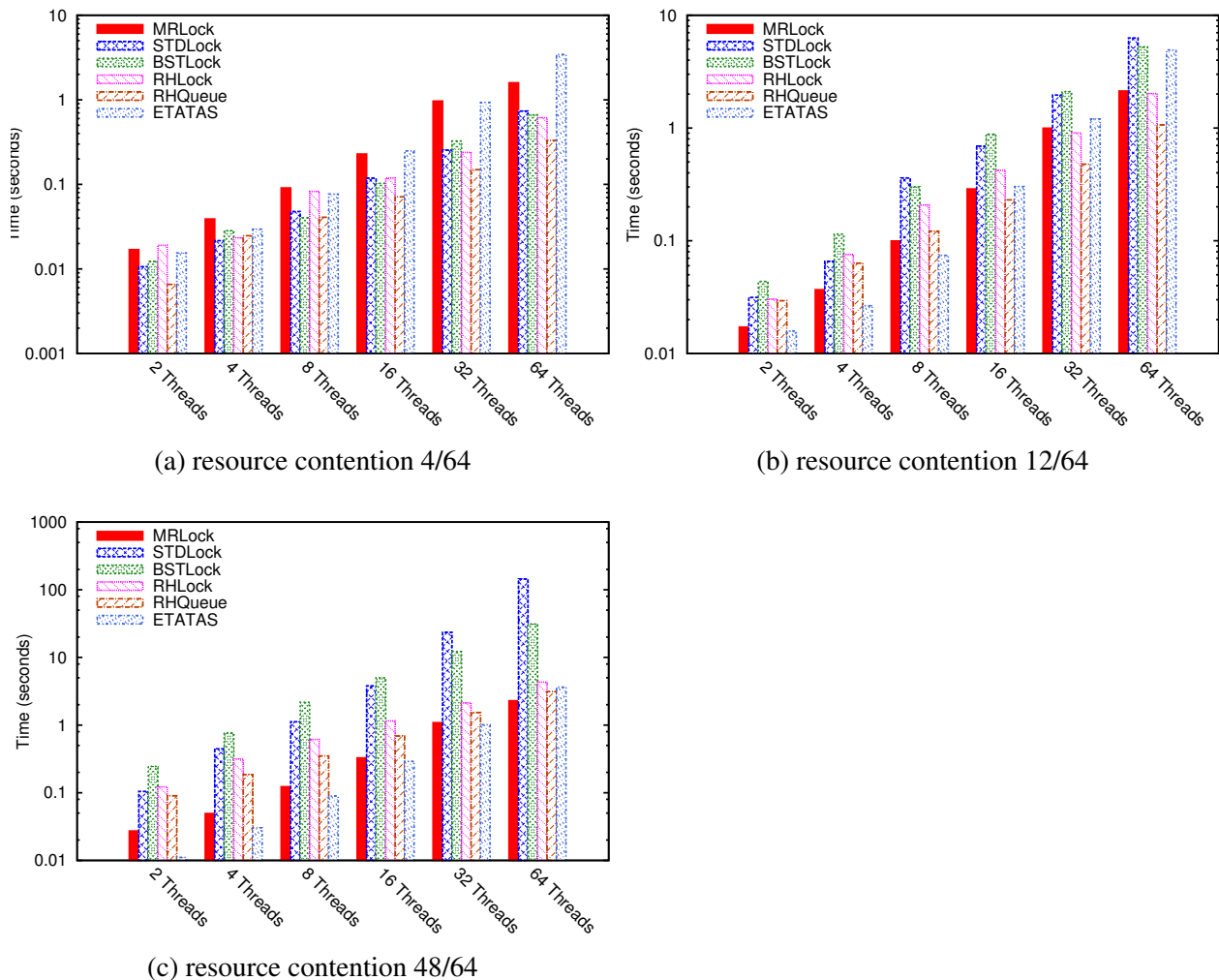


Figure 9: Thread Scaling Trends for 64 Resources

Thread Scalability

Figures 9a, 9b, and 9c show the execution time for my benchmark in the scenarios where the threads experience contention levels of 4/64, 12/64, and 48/64 respectively. In these graphs, the contention level is fixed and I investigate the performance scaling characteristics by increasing the number of threads. I cluster five approaches on the x -axis by the number of threads, while the y -axis represents the total time needed to complete the benchmark in logarithm scale.

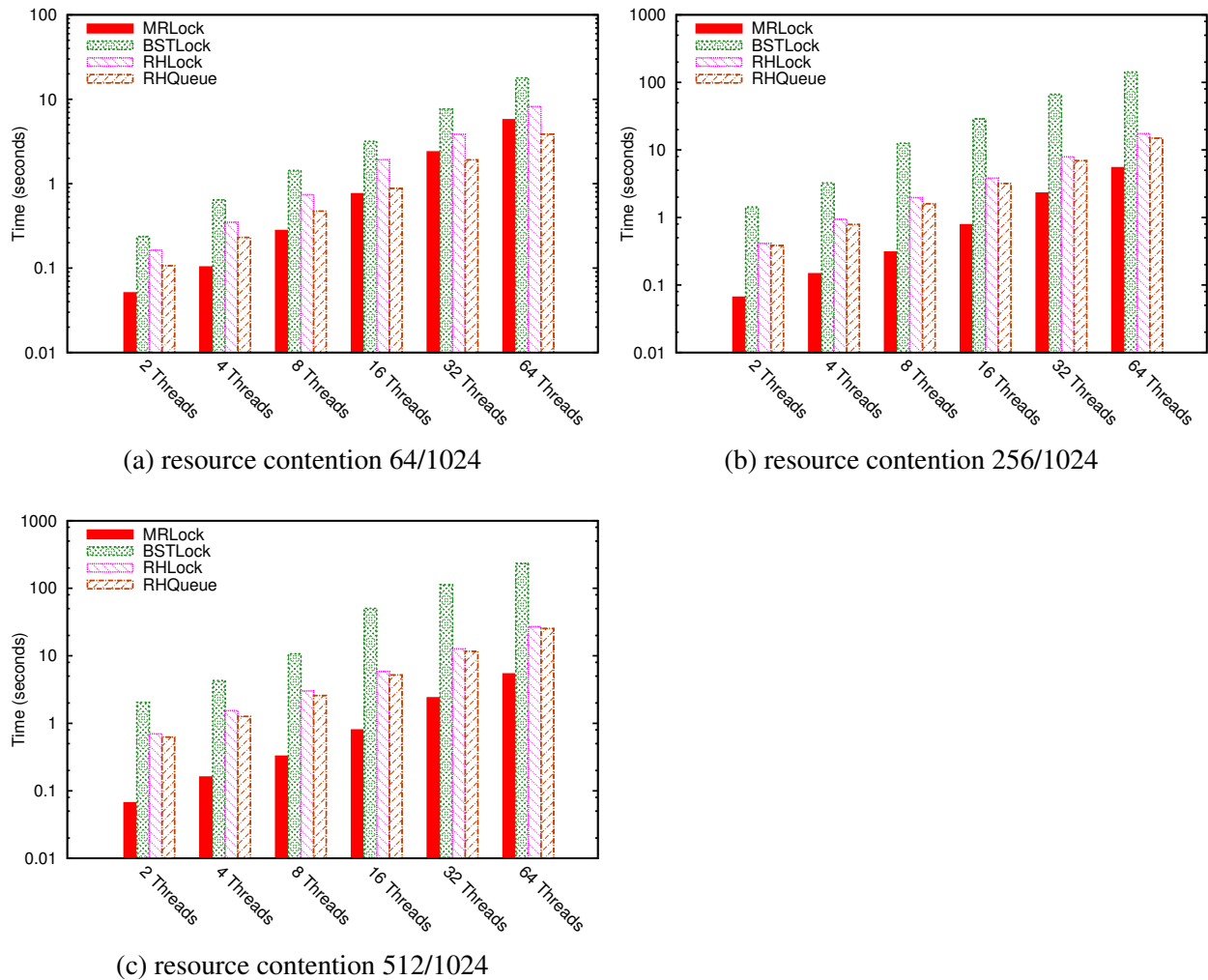


Figure 10: Thread Scaling Trends for 1024 Resources

When the level of resource contention is low, MRLock and ETATAS do not exhibit performance advantages over the other approaches. This is shown in Figure 9a. In this scenario we observe that when using 32 threads, MRLock is 3.7 times slower than STDLock. The difference in performance decreases to about 2 times on 64 threads, which implies that my approach has a smaller scaling factor. I also observe better scalability of the MRLock approach against ETATAS; when moving from 32 threads to 64 threads the performance of ETATAS degrades threefold resulting in a 2 times slowdown compared to MRLock.

For 64 resources, the resource contention level that is a pivot point for my algorithm's performance is 12/64 or 18% as shown in Figure 9b. MRLock ties with RHQueue up to 16 threads and outperforms the other algorithms. MRLock is 4 times faster than STDLock and twice as fast as ETATAS on 64 threads. In addition, MRLock exhibits better scalability compared to its alternatives. The time needed to complete the benchmark for ETATAS, BSTLock and STDLock almost tripled when the number of threads is increased from 32 to 64, while the time of MRLock only increases by 100%. However, we do observe good performance for RHQueue when the number of threads increases. Using the resource hierarchy protocol as oppose to a centralized manager, RHQueue is more efficient handling the contention among threads, because memory access is spread among a number of mutexes assigned to each resource. However, as the level of resource contention increases the benefit of handling resource requests in batches outweighs the slowdown introduced by contending threads. This is shown in Figure 9c, and further illustrated in Figures 10b and 10c where I test the algorithms up with a pool of 1024 resource. In Figure 9c STDLock takes more than 20 times longer than MRLock while RHQueue is 40% slower than MRLock on average. MRLock also outperforms the other approaches on all scales except for ETATAS. ETATAS exhibits poor scalability when increasing the number of threads. As a light weight algorithm it dose not provide any fairness guarantee, but it is surprisingly the fastest when there are less than 16 threads. In Figures 10b, and 10c I exclude ETATAS and STDLock due their inapplicability to 1024 resources.

MRLock outperforms the alternatives on all scales by an average of 8 times with slightly larger performance advantage towards scenarios with fewer threads (to the left of x -axis). This is, again, due to the centralized manager handling the contention among competing threads. Note that the pivot contention level becomes smaller when increasing the total number of resources. As shown in In Figures 10a and 8d, MRLock outperforms the alternatives starting from resource contention level 64/1024 or 6% as oppose to 18% in the 64 resources scenario.

Performance Consistency

It is often desirable that an algorithm produces predicible execution time. I demonstrated in Section 5 that my multi-resource lock exhibits reliable execution time regardless the level of resource contention. Here, I further illustrate that my lock implementation achieves more consistent timings among different runs when compared to the competing implementations.

Figures 11a and 11b display the standard deviation of execution times from 10 different runs. I generate randomized resource requests at the beginning of each test run (Algorithm 25), so the actual resource conflicts might be different for each run. I show the absolute value of the standard deviation on the y axis, and the number of threads on the x axis (both are in logarithm scale). Overall, the deviation of all approaches grows slightly as the number of threads increases. This is expected because in regions of high parallelism, the operating system itself contributes in large part to this overhead. By looking at the absolute values, MRLock achieves the lowest variation, which means it also outperforms TATAS in terms of consistency. This indicates that the incorporation of a FIFO queue stabilized my lock algorithm. Notably, the deviation of STDLock grows linearly. I also include the percentage deviation in Figures 11d and 11c. The y axis gives the percentage error normalized by the average time, the variation of MRLock is within 2% or its executing time.

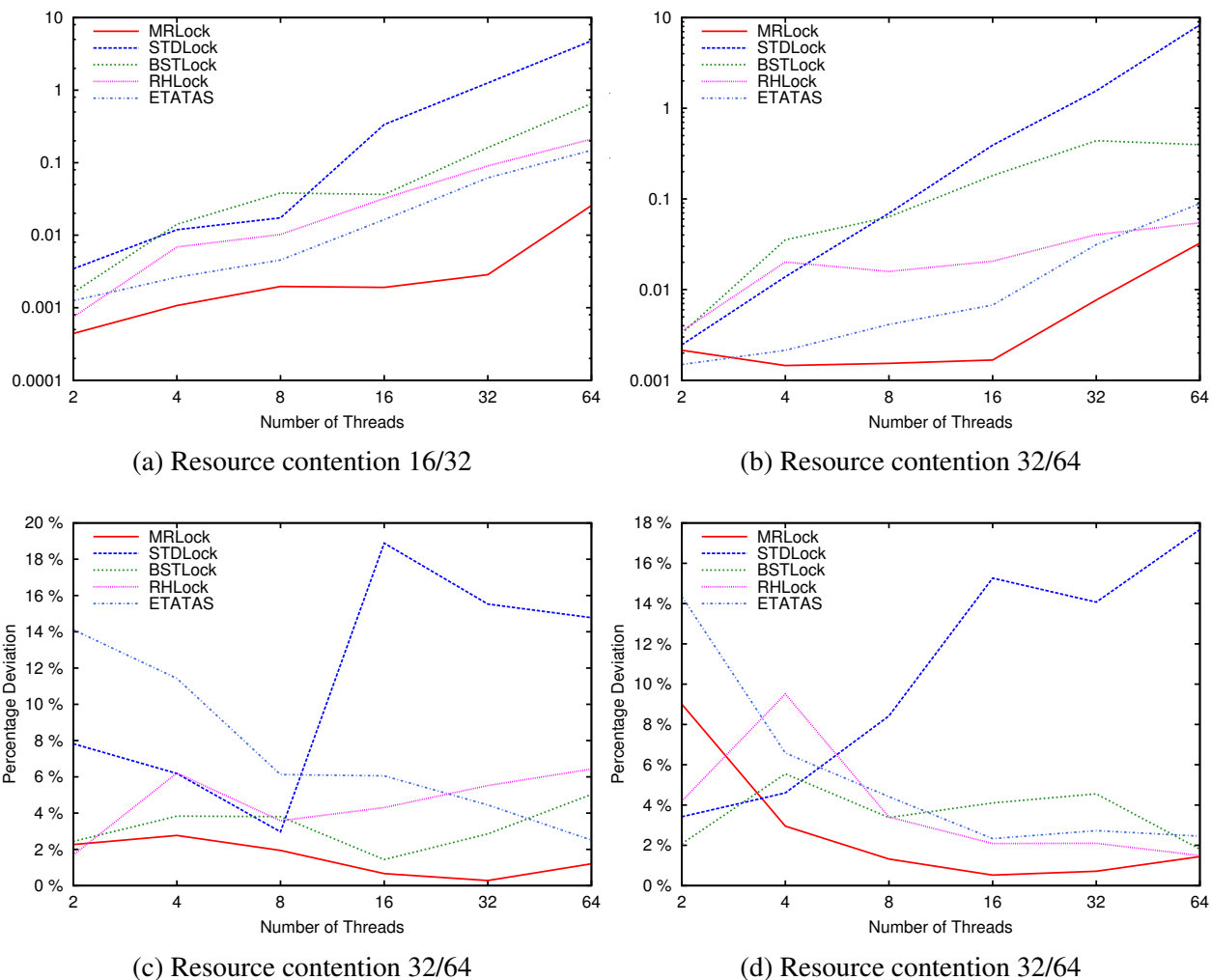


Figure 11: Standard deviation (11a, 11b) and relative error (11a, 11b) out of 10 runs

Lock-free Transactions

In this section, I compare the overhead and scalability of my lock-free transactional list and skiplist against the implementations based on transaction boosting, NOrec STM from Rochester Software Transactional Memory package [69] and Fraser’s lock-free object-based STM [35]. RSTM is the best available comprehensive suite of prevailing STM implementations. Most of the algorithms

distributed with RSTM support building transactional data structures with a few exception such as single lock algorithms and in-place write algorithms. Due to their lack of support for explicit self-abort, transactions with failed operations cannot be revoked leading to potentially erroneous behavior. In my test, TML [20] and its extension NOrec [20] are among the fastest on my platform. They have extreme low overhead and good scalability due to elimination of ownership records. I choose NOrec as the representative implementation because its value-based validation allows for more concurrency for readers with no actual conflict.

For transaction boosting, I implement the lookup of abstract lock using Intel TBB's concurrent hash map. Although the transaction boosting is designed to be used in tandem with STMs for replaying undo logs, it is not necessary in my test case as the data structures are tested in isolation. To reduce the runtime overhead, I scrap the STM environment and implement a lightweight per-thread undo log for the boosted data structures. I employ a micro-benchmark to evaluate performance in three types of workloads: write dominated, read dominated, and mixed. This canonical evaluation method [20, 46] 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. I 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.

Transactional List

I show the throughput and the number of spurious aborts in Figure 12 for all three types of lists. The throughput is measured in terms of number of completed operations per second, which is the product of the number of committed transactions and transaction size. The number of spurious

aborts takes into account the number of aborted transactions except self-aborted ones (i.e., those that abort due to failed operations). This is an indicator for the effectiveness of the contention management strategy. Each thread performs 10^5 transactions and the key range is set up to 10^4 . My lock-free transactional list is denoted by LFT, the boosted list by BST, and the NOrec STM list by NTM. The underlying list implementations for both LFT and BST were based on Harris' lock-free design [46]. The linked list for NTM is taken directly from the benchmark implementation in RSTM suite. Since the lock-free list and the RSTM list use different memory management scheme, I disable node reclamation for fair comparison of the synchronization protocols. For each list legend annotation, I append a numeric postfix to denote the transaction size (e.g., BST-4 means the boosted list tested with 4 operations in one transaction).

In Figure 12a, threads perform solely write operations. The upper half of the graph shows the throughput with both y - and x -axes in logarithm scale. Starting with transaction of size 1, the throughput curve of BST-1 and LFT-1 essentially expose the overhead difference between the two transaction synchronization protocols. Because each transaction contains only one operation, the code paths for transaction rollback in BST and transaction helping in LFT will not be taken. For each node operation, BST-1 needs to acquire and release a mutex lock, while LFT-1 needs to allocate a transaction descriptor. For executions within one CPU chip (no more than 16 threads), LFT-1 maintains a moderate performance advantage to BST-1, averaging more than 15% speedup. As the execution spawns across multiple chips, LFT-1's performance is setback by the use of descriptor, which incurs more remote memory accesses. This trend can be observed for all scenarios with different transaction sizes. Another noticeable trend is that LFT lists gain better performance as the transaction size grows. For example, on 64 threads the throughput of LFT-2 slightly falls short behind that of BST-2, then the performance of LFT-4 is on par with BST-4, and finally LFT-8 and LFT-16 outperforms their BST counterpart by as much as 50%. Two factors contribute to the great scalability of LFT lists in handling large transactions: 1) its helping mechanism manages

conflict and greatly reduces spurious aborts whereas in BST such aborts cause a significant amount of rollbacks; 2) the number of allocated transaction descriptors decreases as the transaction size grows whereas in BST the number of required lock acquisitions stays the same.

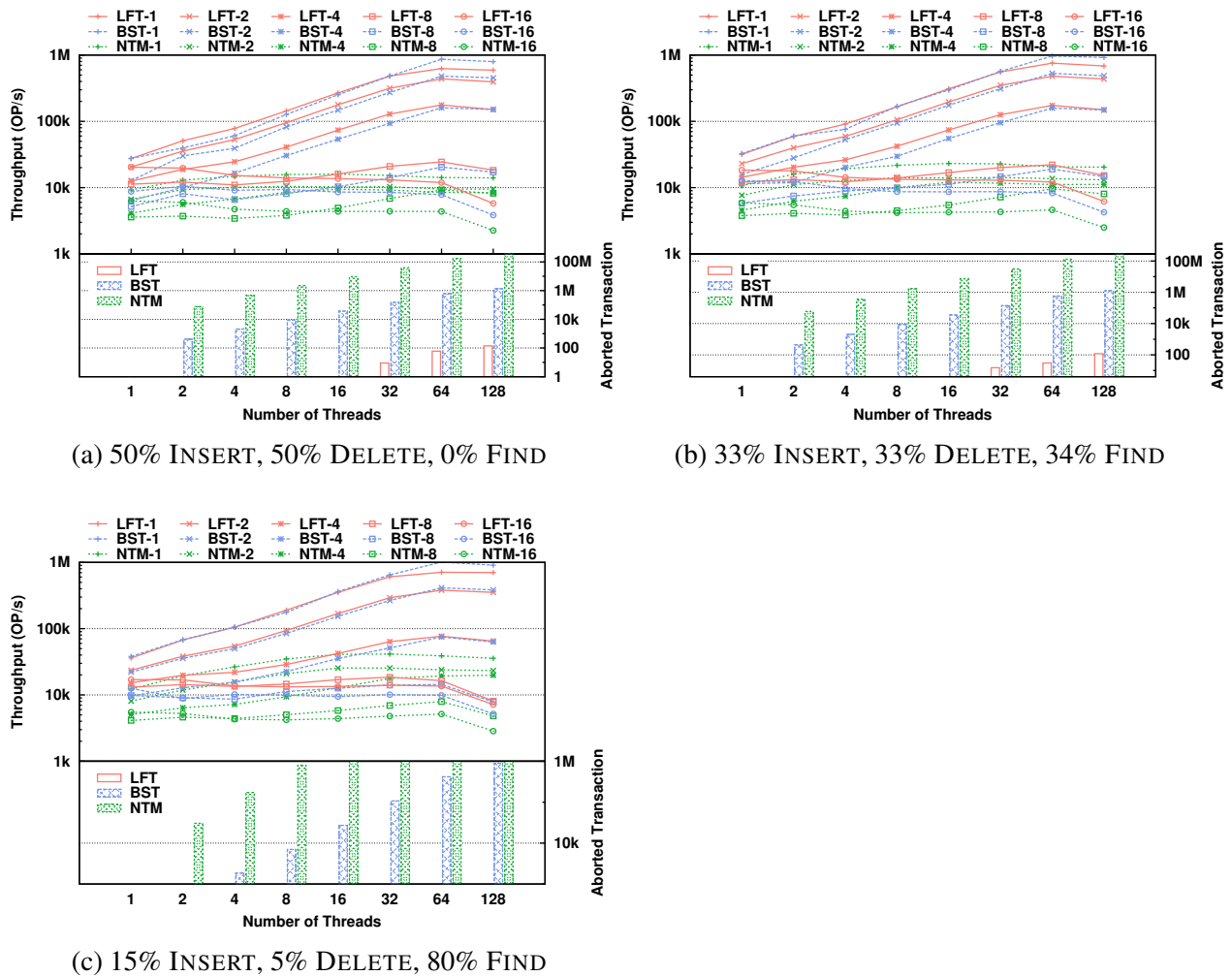


Figure 12: Transaction Lists (10K Key Range)

Generally, we observe that for small transaction sizes (no more than 4 operations), BST and LFT lists explore fine-grained parallelism and exhibit similar scalability trends. The throughput increases linearly until 16 threads, and continues to increase at a slower pace until 64 threads. Be-

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

On the bottom half of Figure 12a, I illustrate the histogram of spurious aborts across all transaction sizes and cluster them by thread counts. The y -axis is in logarithmic scale. For BST lists and NTM lists the number of spurious aborts grows linearly with the increase of threads. BST lists have about 100 times less aborts than NTM lists, which matches my intuition that semantic conflict detection can remarkably reduce the number of false conflicts. Also as expected no approach incurs spurious aborts in single thread scenario. Remarkably, LFT lists do not introduce spurious aborts until 32 threads, and the number of aborts is 4 orders of magnitude smaller than that of BST lists. The helping mechanism of LFT is able to resolve majority of the conflicts in a cooperative manner, and aborts only when cyclic dependencies exist among transactions.

I show the results from mixed and read-dominated workloads in Figure 12b and 12c. The throughputs follow the same pattern as in Figure 12a with LFT lists' performance advantage slightly diminishes in read-dominated workload. This is because the FIND operations in LFT lists also update descriptors in nodes, which requires extra cycles compared with read-only BST FIND operations.

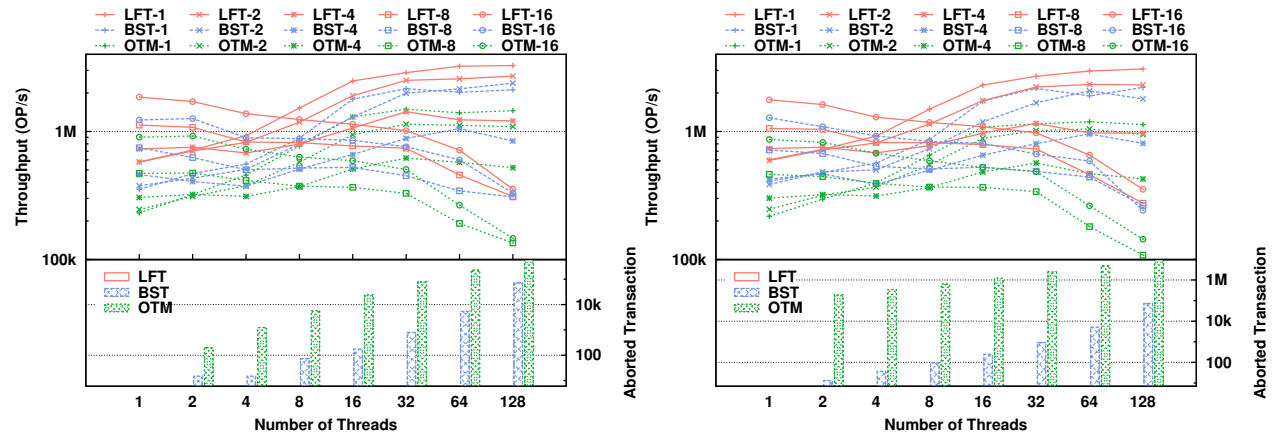
Nevertheless, LFT lists still maintain an average of 20% speedup over BST lists in these scenarios and achieves as much as 40% throughput gain for large transactions. Because of allowing reader concurrency, NTM lists also exhibit some degree of scalability in read-dominated scenarios.

Transactional Skiplist

In Figure 13, I show the throughput and the number of spurious aborts for three types of transactional skiplists. All of them are based on Fraser's open source lock-free skiplist [35], and the epoch-based garbage collection is enabled in all three. The naming convention for BST and LFT skiplists remain the same as in Figure 12. I denote the STM based skiplist as OTM because it uses Fraser's object-based STM. Compared with word-based STMs, an object-based STM groups memory into blocks thus reducing metadata overhead. Since skiplists have logarithmic search time, I am able to stress the algorithms with heavier workloads: each threads now performs 1 million transactions and the key range is also boosted to 1 million.

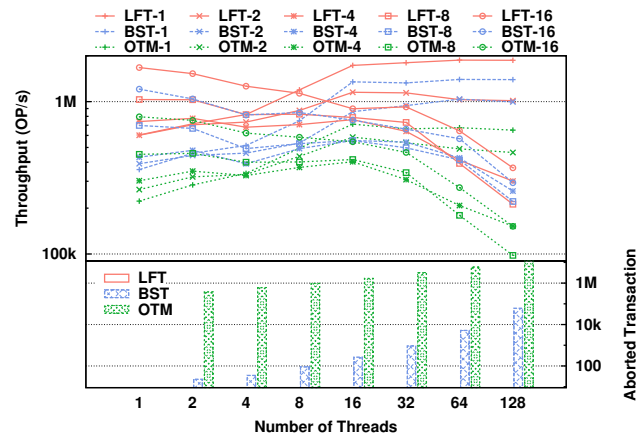
Overall, with a peak throughput of more than 3 million (OP/s), transaction execution on skiplists is considerably more efficient than on linked lists. Also both OTM and BST skiplists generates 2 orders or magnitude less spurious aborts than their list counterparts. Because skiplist algorithms traverse exponentially less nodes than list algorithms, a single operation can finish much sooner, which greatly reduces the probability of memory access conflicts in STM and lock acquisition time out in transaction boosting. Another noteworthy difference is that the divergent scalability trends of large and small transactions. As we can see in Figure 13a, large transaction such as LFT-8 and LFT-16 achieves maximum throughputs on a single thread, then their throughputs steadily fall as the number of threads increases. On the contrary, the throughputs of small transactions such as LFT-2 and LFT-4 start low, but gain momentum as more threads are added. Large transactions have lower synchronization overhead but are vulnerable to conflict. As the number of threads increases,

a failed large transaction could easily forfeit a considerable amount of operations. On the flip side, small transactions incur greater synchronization overhead, but are less likely to encounter conflicts when more threads contend with each other.



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

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



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

Figure 13: Transaction Skiplists (1M Key Range)

Despite the differences, we still observe performance results generally similar to that of transaction lists. LFT and BST skiplists outperform OTM skiplists by as much as 3 times across all scenarios, while LFT skiplists maintain an average of 60% speedup over BST for large transactions. For

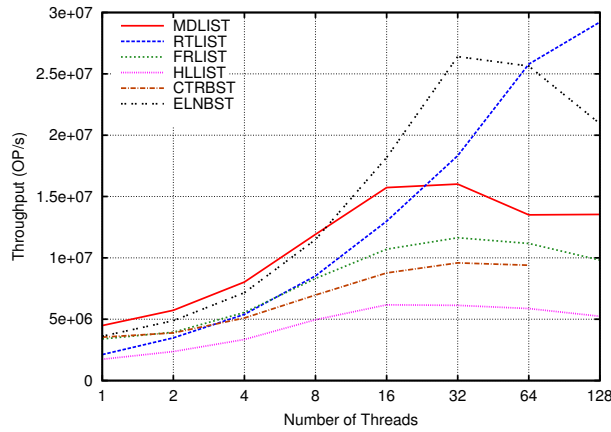
example, in Figure 13a on 32 threads LFT-8 outperforms BST-8 by 125%. Even for small transactions, LFT skiplists begin to set the throughput apart from BST skiplists further than what is in Figure 12. For example, in Figure 13b on 32 threads LFT-2 achieves an 30% speedup over BST-2.

Lock-free Dictionaries

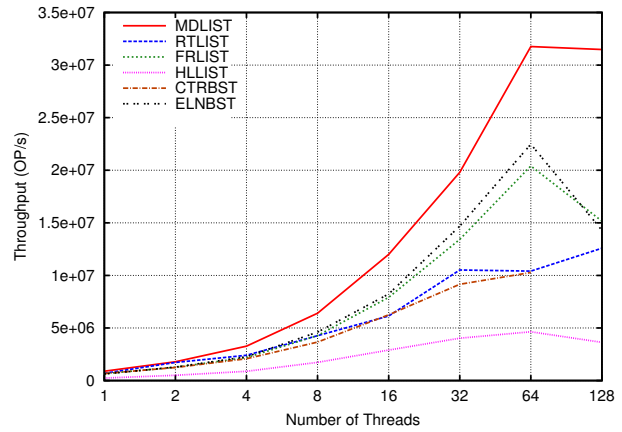
I compare the performance of MDList to the following concurrent skiplists and BSTs that are available to me in C/C++ implementations.

1. The rotating skiplist (RTLIS) by Dick et al. [25]. Their implementation is build upon the C port of Crain's no hot-spot skiplist [19], which employs a background thread to maintain balance and handle physical deletions.
2. Herlihy's lock-free skiplist [57] (HLLIST) with lazy deletions. The tested version is derived from Wicht's C++ implementation [95].
3. Fraser's publicly available lock-free skiplist [35] (FRLIST), which includes several optimizations not found in HLLIST. It is often considered as the most efficient skiplist implementation.
4. The lock-based implementation of Bronson et al.'s relaxed AVL tree [11] (BRNBST) derived from Wicht's C++ implementation [95].
5. The lock-free unbalanced BST (ELNBST) based on Ellen et al's algorithm [29]. The implementation is derived from Wicht's C++ implementation [95].
6. The Citrus tree [4] (CTRBST) by Arbel et al, which employs the novel synchronization mechanism *read copy update* (RCU). RCU allows updates to produce a copy of the data

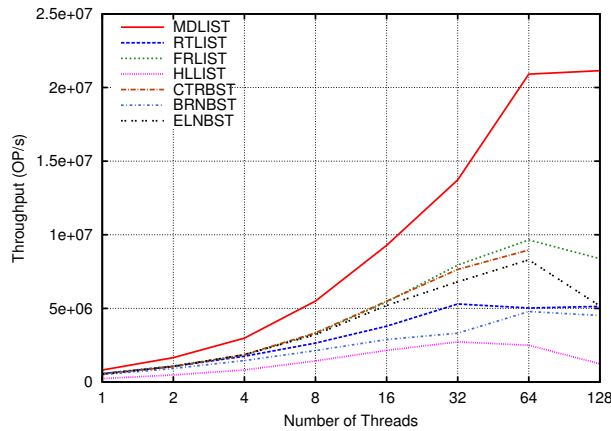
they write so the read-only accesses can proceed without acquiring locks. Code is available directly from the authors.



(a) 1K keys



(b) 1M keys



(c) 1G keys

Figure 14: 50% INSERT, 50% DELETE, 0% FIND on the NUMA System

The original implementation of FRLIST, RTLIST and my algorithm used the epoch based garbage collection proposed by Fraser [35], while BRNBST, ELNBST and HLLIST used hazard pointers [75]. CTRBST, on the other hand, did not employ any memory management scheme. For fair comparison of the algorithms themselves, I disabled memory reclamation for all approaches but

use thread-caching malloc ¹ as a scalable alternative to the standard library malloc. I employ a micro-benchmark to evaluate the throughput of these approaches for uniformly distributed keys. This canonical evaluation method [46, 25, 76] consists of a tight loop that randomly chooses to perform an INSERT, a DELETE or a FIND operation in each iteration. Each thread performs one million operations and I take the average from four runs. As done in [76, 13] I pre-populate the data structures to half capacity to ensure consistent result.

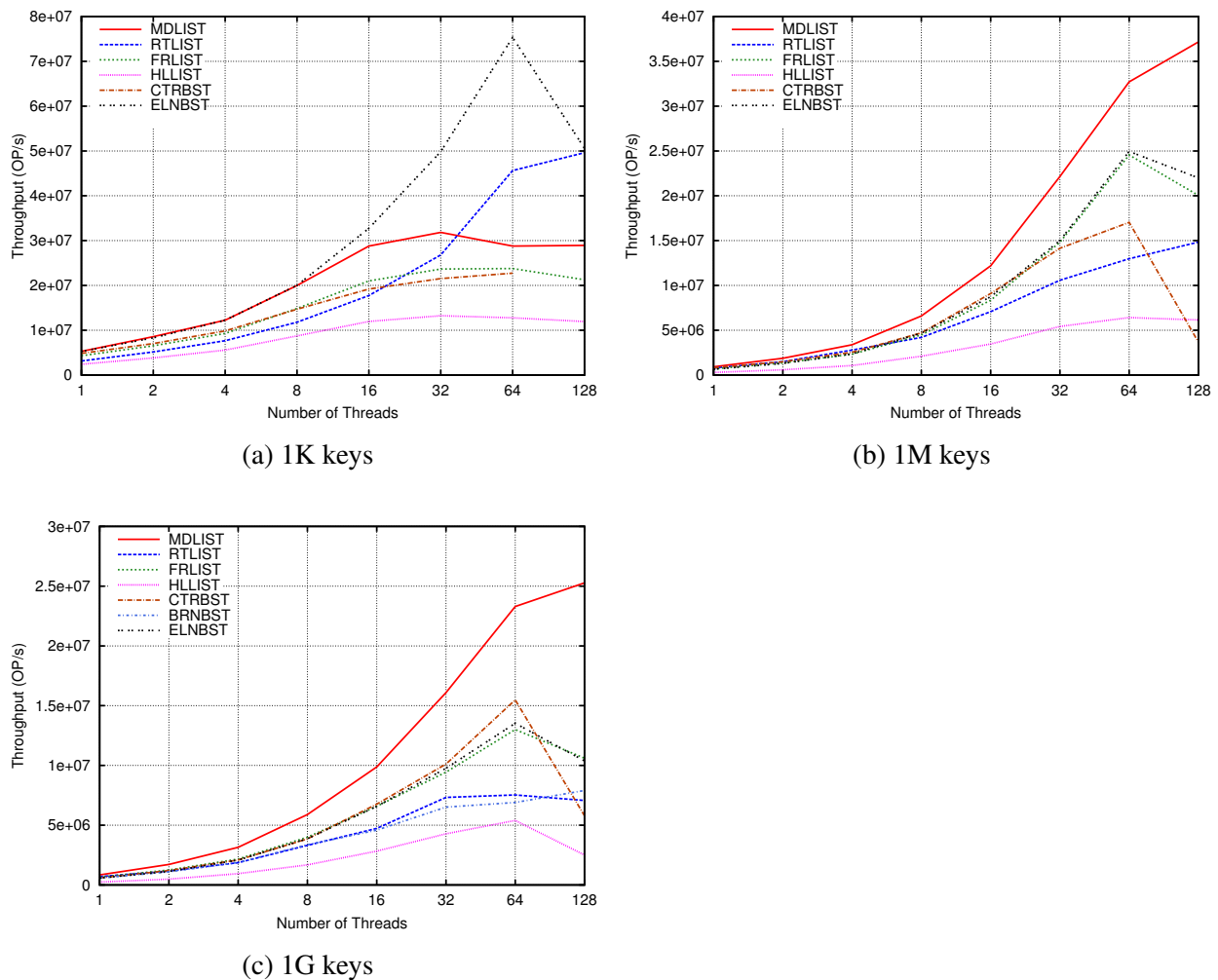


Figure 15: 20% INSERT, 10% DELETE, 70% FIND on the NUMA System

¹<http://goog-perftools.sourceforge.net/doc/tcmalloc.html>

Figures 14, 15 and 16 illustrate the algorithms' throughput on the NUMA system. The y -axis represents the throughput measured by *operation per second*, and the x -axis represents the number of threads in logarithmic scale. Like the evaluations in [76, 59], I consider three different workload distributions: a) write-dominated with 50% insertion, 50% deletion; b) mixed workload with 20% insertion, 10% deletion and 70% find; c) read-dominated with 9% insertion, 1% deletion and 90% find. I also consider three ranges of keys, 1000 (1K), 1 million (1M) and 1 billion (1G). The size of key space affects the height of search trees, but it does not affect the tower height of a skiplist nor the dimension of an MDList as those are chosen by users prior to execution.

Figures 14a, 14b and 14c depicted the write-dominated situation. We observe that the both skiplist-based and BST-based dictionaries were able to explore fine-grained parallelism and exhibit similar scalability trends. The overall throughput increases almost linearly until 16 threads, and continues to increase at a slower pace until 64 threads. Because executions beyond 16 threads span across multiple chips, the performance growth is slightly reduced due to the cost of remote memory accesses. The executions are no longer fully concurrent beyond 64 threads, thus the overall throughput is capped and may even reduce due to context switching overhead. For small key space of 1K keys (Figure 14a, RCU-based Citrus trees stand out against the rest. This is because most insertions would not update the data structures due to the existence duplicated keys, and they essentially become read-only operations for which RCU is optimized. MDLIST has slightly larger overhead for traversing operations if it is under-populated. To locate an existing node, the prefix matching algorithm always needs to perform D comparison operations, whereas in a BST, the search algorithm could terminate much earlier because of shallow depth. For 1M key space (Figure 14b, most approaches achieve similar throughput except for HLLIST. MDLIST outperform the other approaches by an average of 25%. It continues to excel in large key space with one billion keys. As shown by Figure 14c, MDLIST outperforms the best alternatives including FRLIST and CTRBST by as much as 100%. For larger key space, insertions are less likely to collide with an

existing key. The size of the data structures grows much quicker too. Each insertion in MDLIST modifies at most two consecutive nodes, incurring less remote memory access than the skiplist-based and BST-based approaches.

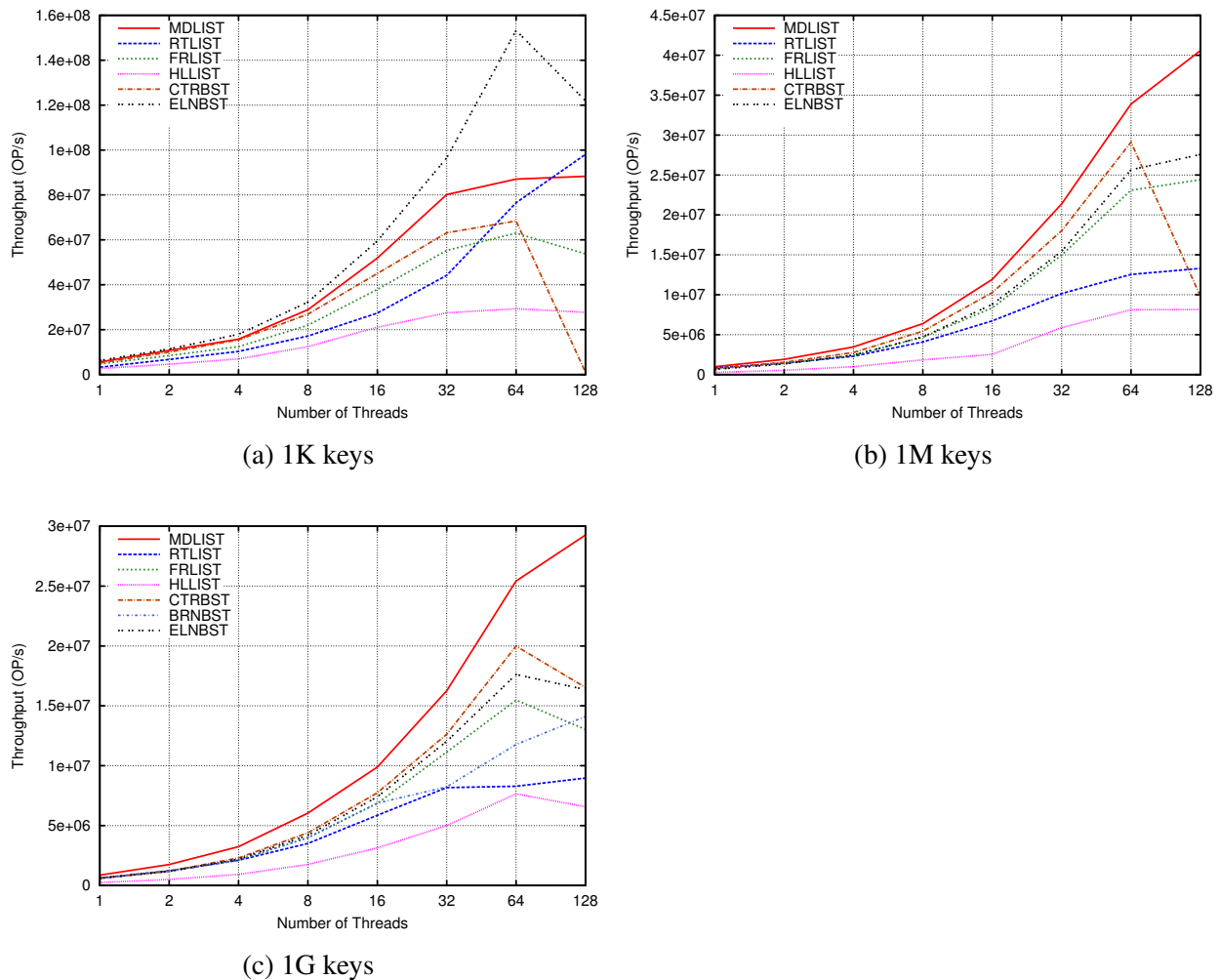


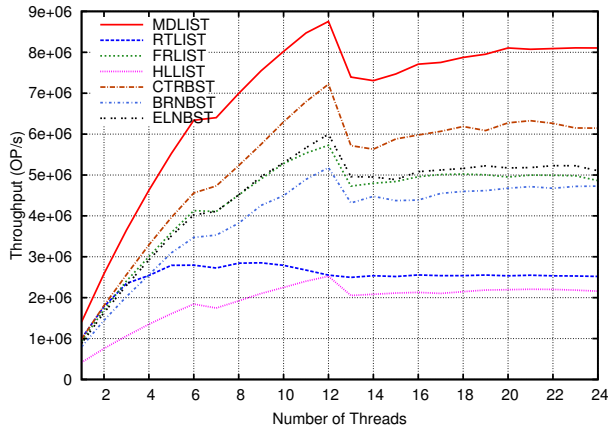
Figure 16: 9% INSERT, 1% DELETE, 90% FIND on the NUMA System

Figures 15a, 15b and 15c show the throughput for the mixed workload. The overall scalability trends for all three key ranges mimic those of the write-dominated workload respectively. In key spaces with 1M to 1G keys, MDLIST achieves 15% 30% speedup over FRLIST and ELNBST, which are two of the best skiplist-based and BST-based algorithms in mixed workload. Fig-

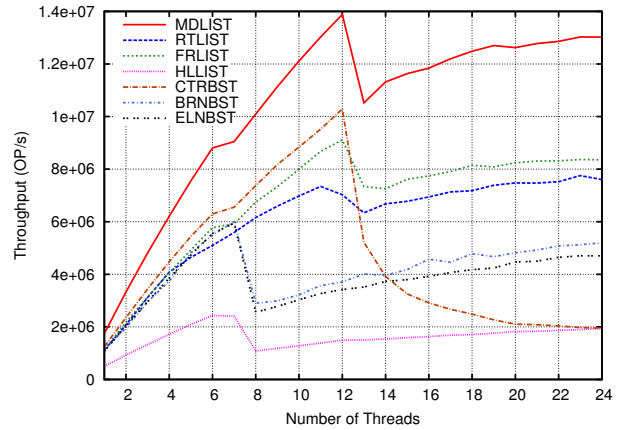
ures 16a, 16b and 16c show the throughput for the read-dominated workload. As the distribution of mutating operations further decreases to one tenth of the whole operations, the performance gaps among different algorithms begin to diminish. For 1G keys (Figure 16c, RTLIST, FRLIST, CTRBST, and ELNBST have almost identical performance up until 8 threads. This is because all the data structures being tested implement logarithmic search. Less writes means less interference between concurrent updates and traverse. The performance characteristics of difference algorithms thus converge towards an ideal straight line. They differ only in term of scalability at high levels of concurrency. CTRBST achieves the best scalability among the alternatives but is still 20% slower than MDLIST. Note that in Figure 16a CTRBST's performance degrades drastically when the benchmark spawn more threads than the number of available hardware cores. CTRBST employs a user level RCU library that uses global locks, which may delay running threads if the threads holding the locks get preempted.

Figure 17a and 17b show the throughput of the algorithms on the SMP system. The x -axis in these graphs is in linear scale. In Figure 17a, the executions consist solely of INSERT operations, which insert keys in the range of 32-bit integers. For all approaches, the overall system throughput peaks at 12 threads which is the maximum number of hardware threads. Executions beyond 12 threads are preemptive, and the throughput slightly dropped due to unbalanced load causing increasing amount of cache invalidation. MDLIST provides an average of 30% speedup over CTRBST on all levels of concurrency. While the performance of the BST-based approaches closely resembles each other, the performance of skiplist-based approaches varies. Notably, the throughput of RTLIST drops significantly beyond 6 threads. The impact of a dedicated maintenance is clearly visible on the SMP system, where the background thread has to compete with working threads for limited hardware cores. Because the background thread is the only thread that does physical deletion, the overall progress will stagnate once the it is suspended. In Figure 17b, I distribute the workload by having 30% insertions, 20% deletions and 50% searches. MDLIST outperforms FRLIST by as

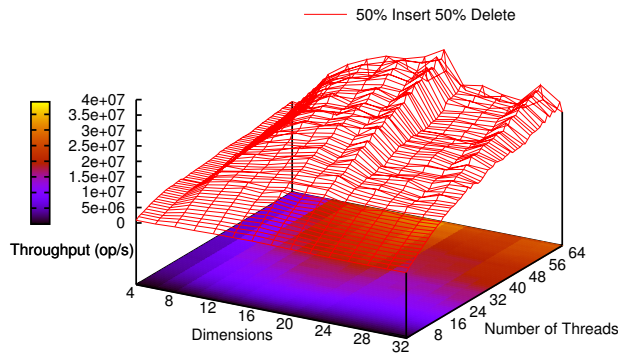
much as 60%. The throughput of CTRBST again drops drastically after exhausting all hardware threads.



(a) 4G keys, 100% INSERT



(b) 4M keys, 50% updates



(c) Dimension sweep

Figure 17: Throughput on SMP system and dimension sweep on NUMA

In Figure 17c, I sweep the dimension of MDLIST from 4 to 32 on the NUMA system, and show that the algorithm achieves maximum throughput with 20 dimensions on 64 threads. On all scale levels, we see that the throughput converges towards 20 dimensions. This means that the way the dimensionality of an MDLIST affects its performance is independent from the number of threads. The performance of MDLIST can be optimized if the access pattern of the user application is taken

into account.

Overall, MDLIST excels at high levels of concurrency with large key spaces. The locality of its operations makes it suitable for NUMA architectures where remote memory access incurs considerable performance penalties. On an SMP system with low concurrency, MDLIST performs equally well or even better than the state of the art skiplist-based and BST-based approaches.

CHAPTER 6: CONCLUSION

This dissertation presents a forward-looking and pragmatic approach that will lead to the discovery of the key principles for effective multiprocessor algorithm and application development. I introduced two methodologies for implementing high-performance transactional data structures based on their linearizable counterparts.

My multi-resource lock algorithm (MRLock) provides a scalable solution to lock-based transactional synchronization by solving the resource allocation problem on shared-memory multiprocessors. The MRLock algorithm guarantees FIFO fairness for contending threads, and is scalable with significant performance increase over the best available solutions. As demonstrated by our experimental evaluation, the MRLock algorithm exhibits reliability and scalability that can be beneficial to applications experiencing high levels of resource contention.

My lock-free transactional transformation (LFTT), on the other hand, transforms lock-free linked data structures into fast lock-free transactional data structures. My approach embeds the transaction metadata in each node, which enables resolving transaction conflicts cooperatively through thread-level synchronization. No undo logs nor rollbacks are needed because operations can correctly interpret the logical status of nodes left over by aborted transactions. Data structures that guarantees lock-free or weaker progress will be able to maintain their progress properties during the transformation. The performance evaluation results show that my transaction synchronization protocol has low overhead and high scalability—providing more than 10 times over the alternative word-based STM and 3 times over the object-based STM. Besides the performance advantages, my approach decreases spurious aborts to a minimum, which is desirable because transaction success rate is a decisive factor for a majority of the applications.

Furthermore, I presented a simple and efficient lock-free dictionary design based on MDList.

It maps keys into high dimensional vector coordinates and achieve deterministic layout that is consistent with nodes' logical ordering. I exploited spatial locality to increase the throughput of the INSERT operations, and adopted asymmetrical logical deletion to address the synchronization overhead of the DELETE operations. When compared to the best available skiplist-based and BST-based algorithms, my algorithm achieved performance gains in scenarios with medium to large key spaces. The performance of my dictionary can be tailored to different access patterns by changing its dimensionality.

Finally, I integrated the above mentioned components and introduced my open source library libtxd. It is a software framework to build future transactional data structures. The package includes four lock-free transactional data structures, two abstract data type interfaces, and helper classes to support MRLock-based transactions. The library provides intuitive interface and is developed with sound software engineering practices to benefit both researchers and industry users.

APPENDIX A: CORRECTNESS PROOF OF MRLOCK

In this section, I reason about the safety and liveness of the multi-resource lock algorithm. My lock manager is safe because it maintains the desired semantics under concurrent acquire and release: all requests to acquire locks are served in FIFO order and a thread must wait until its resource request is not in conflict with previous requests. The underlying lock-free queue guarantees starvation-freedom for threads within the queue and deadlock-freedom for all contending threads.

Safety

By using the *sequence number* as a sentinel the following properties of my algorithm are guaranteed: 1) The head always precedes the tail, i.e., $H_{pos} \leq T_{pos}$ where H_{pos} and T_{pos} denote the value of head and tail as defined on line 2.12 and 2.13 in Algorithm 2. The head advances only if the sequence number of the cell is equal to $H_{pos} + 1$ (line 4.9 and 4.10 of Algorithm 4). This occurs when a previous enqueue operation sets the sequence number to $T_{pos} + 1$ (3.line 18 of Algorithm 3). 2) The tail is at most *siz* away from the head, i.e., $T_{pos} - H_{pos} \leq siz$ where *siz* denotes the size of the ring buffer. In other words, the tail cannot overtake the head. The tail advances when the sequence number of the cell is equal to T_{pos} (line 3.11 and 3.12 of Algorithm 3). When the tail wraps around the buffer trying to overtake the head, the sequence number of the cell could be either $T_{pos} - siz$ or $T_{pos} - siz + 1$ depending on whether previous enqueue has updated the sequence number. This enqueue will wait until the head advances. Therefore, the cells between head and tail are always valid and store outstanding requests in FIFO order.

The queuing nature of my multi-resource lock allocates a cell exclusively for each contending thread, which drops the limitation of atomic bitset access required by the extended TATAS lock (Algorithm 1). In my algorithm, each bitset is set to 1 for all the bits during initialization (line 25 of Algorithm 2) and then alternates between 0s, 1s (lines 4 and 12 of Algorithm 4), and desired lock values (line 3.17 of Algorithm 3). A bitset can have maximum one writer because each cell

is allocated to one thread. Regardless the duration of the writing, the bitset maintains its “locking capability” throughout the whole procedure. Since occupied resources are denoted by 1, a bitset of all 1s denotes the set of all resources and any other values denotes a subset of it. When updating a bitset from all 1s to any specific request value, it’s essentially removing unwanted resources from the set by filling in 0s, thus the intermediate values always represent some supersets of the requested resources. Therefore, it is not possible for any overlapping reading thread to bypass with conflicting request. Similarly, when the bitset is set to all 0s during lock release, the intermediate values always represents some subsets of the requested resources. This prevents the unlocking operation from blocking threads with no conflicting resource request.

Liveness

My lock algorithm is deadlock-free for all threads because the concurrent queue I use for my implementation guarantees lock-free progress when it has not reached its capacity. This means that in a scenario of contending threads, at least one thread succeeds when attempting to acquire or release its desired resources. In Algorithm 3 a thread retries its enqueue operation when the CAS update fails (3.line 13) or the sequence number mismatches (line 3.12). After loading the most recent value of the tail (line 3.8), the CAS fails when the tail has been updated by an intervening thread. The sequence number check fails if either the queue is full ($diff < 0$) or the cell has been taken by an intervening thread ($diff > 0$). When an enqueue attempt fails while the queue is not full, this is an indication that another thread must have succeeded in completing an enqueue operation. Therefore, lock-free property is satisfied among all contending threads while starvation-freedom is provided to the threads within the queue. If a wait-free queue is used in place of the lock-free queue, my lock algorithm will provide starvation-freedom for all threads. Such an implementation is possible based on the method proposed by Kogan [62], but it requires performance trade-off.

With my motivation being to construct a practical design, I choose to employ a lock-free queue that strike a balance between performance and progress guarantee.

If the queue is full, any new enqueue operation waits to insert its request in the queue until some thread relinquishes its locks. For threads with already enqueued requests, a full queue does not impair the correct execution of lock acquisition/release in FIFO order. For threads that is waiting to insert new requests, this may cause performance degeneration and loss of the FIFO fairness guarantee. In practice, I can easily avoid this situation by allocating a sufficiently large buffer. If the number of threads is know beforehand, then a buffer size equal to the thread count will suffice because a thread can only file one request at a time. If, however, the number of thread is not determined at runtime, I can allocate a buffer up to the size of maximum number of supported threads by the operation system with acceptable memory overhead (with too many threads the system would experience slowdown due to context switch and other scheduling overhead though). For a typical Linux system, the maximal number of thread is determined by the amount of memory and the size of the stack per thread. For example, a 32-bit system supports up to 4GB memory, that is 4000 threads¹ with stacks of 1MB. To support the same number of threads in the multi-resource lock requires a buffer size of 12KB in total for 2-bytes bitset, a fraction of the available memory.

¹It would be less considering memory reserved for the kernel.

APPENDIX B: CORRECTNESS PROOF OF LFTT

I base my correctness discussion on the notion of commutativity isolation [52], which states that the history of committed transactions is *strictly serializable* for any transactional data structure that provides linearizable operations and obeys commutativity isolation¹. STM systems may prefer more strict correctness criteria, such as opacity [43], because they need to account for the consistency of intermediate memory access. In the case of data structure transactions, the intermediate computation is managed by linearizable methods, and only the end result of a transaction is accessible to users. Strict serializability [80], which is the analogue of linearizability [58] for transactions, provides enough semantics for such cases.

Definitions

I provide a brief recapitulation of the definitions and correctness rules from Herlihy and Koskinen's work [52]. A *history* of computation is a sequence of instantaneous events. Events associated with a method call include invocation I and response R . For a transaction, events associated with its status include $\langle T \text{ init} \rangle$, $\langle t \text{ commit} \rangle$, $\langle T \text{ abort} \rangle$ indicating T start rolling back its effects, and $\langle T \text{ aborted} \rangle$ indicating T finishes its rollback. I denote a sequence of events by concatenate them with \cdot : 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. A *concurrent history* is one in which events of different transactions are interleaved. A *subhistory* is a subsequence of the events of h . The subhistory of h restricted to transaction T is denoted as $h|T$. The subhistory *committed*(h) is the subsequence of h consisting of all events of committed transactions.

Definition 3. 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.

¹I omit the discussion of rules on compensating actions and disposable methods because they are not applicable to my approach

Definition 4. Histories h and h' define the same state if, for every history g , $h \cdot g$ is legal if and only if $h' \cdot g$ is.

Definition 5. For a history h and any given invocation I and response R , let I^{-1} and R^{-1} be the inverse invocation and response, such that the abstract 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 6. Two method calls I, R and I', R' commute if: for all histories h , if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same abstract state.

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

$$\begin{aligned}
 & \text{INSERT}(x) \leftrightarrow \text{INSERT}(y), \quad x \neq y \\
 & \text{DELETE}(x) \leftrightarrow \text{DELETE}(y), \quad x \neq y \\
 & \text{INSERT}(x) \leftrightarrow \text{DELETE}(y), \quad x \neq y \\
 & \text{FIND}(x) \leftrightarrow \text{INSERT}(x)/\text{false} \leftrightarrow \text{DELETE}(x)/\text{false}
 \end{aligned} \tag{B.1}$$

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

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

Rule 3. Compensating Actions: For any history h and transaction T , if $\langle T \text{ aborted} \rangle \in h$, then it must be the case that $h|T = \langle T \text{ init} \rangle \cdot I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot \langle T \text{ abort} \rangle \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1} \cdot \langle T \text{ aborted} \rangle$ where i indexes the last successfully completed method call.

Serializability and Recoverability

I now show that lock-free transactional transformation meets the two above correctness requirements. I 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 .

I show the transformed operations are linearizable by identifying their *linearization points*. Additionally, I use the notion of decision points and state-read points to facilitate my reasoning. The decision point of an operation is defined as the atomic statement that finitely decides the result of an operation, i.e. independent of the result of any subsequent instruction after that point. A state-read point is defined as the atomic statement where the state of the dictionary, which determines the outcome of the decision point, is read.

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

Proof. For the transformed INSERT operation, the execution is divided into two code paths by the condition check on line 22.6. The code path on line 22.7 updates the existing node's logical status. Note that if the operation reports failure on line 9.12 and 9.14, no write operation will be performed to change the logical status of the node. The state-read point for the former case is when the previous transaction status is read from *oldinfo.desc.status* on line 8.5. The state-read point for the later case is when the current transaction status is read from *info.desc.status* on line 9.13. The abstract states S' observed by all transactions immediately after the reads are unchanged, i.e., $\forall i, S'_i = S_i$. For a successful logical status update, the decision point for it to take effect is when the CAS operation on line 9.15 succeeds. The abstract states S' observed by the transactions T_d executing this operation immediately after the CAS is $i = d \implies S'_i = S_i \cup n.key$. For all other transactions $i \neq d \implies S'_i = S_i$. In all cases, the update of abstract states conforms to the

sequential specification of the insert operation. The code path for physically adding linkage to the new node (line 22.11) is linearizable because the corresponding DO_INSERT operation in the base data structure is linearizable.

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

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

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

Proof. As identified in Equation B.1, two set operations commute if they access the different keys. Because of the one-to-one mapping from node to keys, I have $\forall n_x, n_y \in N, x \neq y \implies n_x \neq n_y \implies n_x.key \neq n_y.key$. This means that two set operations commute if they access two different nodes. Let T_1 denotes a transaction that currently accesses node n_1 , i.e., $n_1.info.desc = desc_1 \wedge desc_1.status = Active$. If another transaction T_2 were to access n_1 , it must perform EXECUTEOPS for T_1 on line 9.7 because EXECUTEOPS always update the transaction status when it returns on line 10.26 or 10.29 (note that failed CAS also means the transaction status has been set, but another thread). I thus ensure that $desc_1.status = Committed \vee desc_1.status = Aborted$ before T_2 proceeds. \square

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

Proof. Follow Lemma 1, and 2, and the conclusion in Herlihy and Koskinen's work [52], the theorem holds. \square

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

Proof. Let T_1 be an aborted transaction with descriptor $desc_1$. I denote S as the abstract state immediately after T_1 aborts. Let history $h = F_1 \cdot F'_1 \cdots F_2 \cdot F'_2 \cdot F_x \cdots F'_y$ be the sequence of linearizable method calls after T_1 starts and until T_1 aborts, where $F_i, 1 \leq i \leq x$ denotes the method calls successfully executed by T_1 , and $F'_i, 1 \leq i \leq y$ denotes the method calls executed by other transactions. The interleaving of these method calls is arbitrary. Follow commutativity isolation in Lemma 2 I assure that the method calls after F_x must commute with F_x , thus I can swap them without changing the abstract state. By progressively doing this for $F_i, 1 \leq i \leq x$, I obtain an equivalent history $h = h' = F'_1 \cdots F'_2 \cdots F'_y \cdots F_1 \cdot F_2 \cdots F_x$. Let n_x be the node accessed by F_x . I denote S' as the abstract state before the invocation of F_x . Because of the inverse interpretation of logical status I can assert $n_x.key \in S' \implies n_x.key \in S \wedge n_x.key \notin S' \implies n_x.key \notin S$. Thus, I have $S' = S$ and I can remove F_x from h' without altering the abstract state. Doing this for $F_i, 1 \leq i \leq x$, I obtain $h = h' = h'' = F'_1 \cdots F'_2 \cdots F'_y \cdots$, hence T_1 is removed from the history. \square

Progress Guarantees

Lock-free transaction transform provides lock-free progress because it guarantees that for every possible execution scenario, at least one thread makes progress in finite steps by either committing or aborting a transaction. I reason about this property by examining unbounded loops in all possible executions paths, which can delay the termination of the operations. For a system with i threads, the upper bound of the number of active transactions is i . Consider the while loop that executes the operations on line 10.13. This loop is bounded by the maximum number of operations in a transaction, denoted as j , but threads may set out to help each other during the execution of each

of the operations. The number of recursive helping invocations is bound by the number of active transactions. In the worst case where only 1 thread remains live and $i - 1$ threads have failed, the system guarantees a transaction will commit in at most $i * j$ steps. In the presence of cyclic dependencies among transactions, the system guarantees that a duplicate transaction descriptor will be detected within $i * j$ steps.

My cooperative contention management strategy requires that the users specify all operations in a transaction beforehand. It is possible to allow dynamic transaction executions by adopting aggressive contention management (i.e., the later transaction always forcibly aborts the competitor). In this case, the progress guarantee provided by the system degrades to obstruction-free. I focus on the lock-free implementations because the obstruction-free versions can be trivially obtained by disabling the helping mechanism in the lock-free version.

APPENDIX C: CORRECTNESS PROOF OF MDLIST

In this chapter, I sketch a proof of the main correctness property of the presented dictionary algorithm, which is linearizability [58]. I begin by defining the *abstract state* of a sequential dictionary and then show how to map the internal state of my concrete dictionary object to the abstract state. I denote the abstract state of a sequential dictionary to be a totally ordered set P . Equation C.1 specifies that an INSERT operation grows the set if the key being inserted does not exist. Equation C.2 specifies that a DELETE operation shrinks a non-empty set by removing the key-value pair with the specific key.

$$\text{INSERT}(\langle k, v \rangle) = \begin{cases} P \cup \{\langle k, v \rangle\} & \forall \langle k', v' \rangle \in P, k' \neq k \\ P & \exists \langle k', v' \rangle \in P : k' = k \end{cases} \quad (\text{C.1})$$

$$\text{DELETE}(k) = \begin{cases} P \setminus \{\langle k, v \rangle\} & \langle k, v \rangle \in P \\ P & \langle k, v \rangle \notin P \end{cases} \quad (\text{C.2})$$

Invariants

Now I consider the concurrent dictionary object. By a *node*, I refer to an object of type **Node** that has been allocated and successfully linked to an existing node (line 15.21). I denote the set of nodes that are reachable from *head* by L . The following invariants are satisfied by the concrete dictionary object at all times. Invariant 1 states that if a node has no pending child adoption task, its dimension d child must have d invalid child slots leaving $D - d$ valid ones.

Invariant 1. $\forall n, n' \in L, \text{CLEARMARK}(n.\text{child}[d], F_{adp} | F_{del}) = n' \wedge n.\text{adesc} = \text{NIL} \implies \forall i \in [0, d), \text{ISMARKED}(n'.\text{child}[i], F_{adp}) = \text{true}$

Proof. By observing the statements at line 15.31 and 16.5 we see that the F_{adp} flags are properly initialized before linking a new node to its predecessor and updated properly whenever a child is adopted. \square

Invariant 2 states that any node in L can be reach by following a series child pointers with non-decreasing dimensionality.

Invariant 2. $\forall n \in L, \exists p = \{d_0, d_1, \dots, d_m\} : d_0 \leq d_1 \leq \dots \leq d_m \wedge head.child[d_0].child[d_1] \dots child.[d_m] = n$

Proof. At the start, the structure contains a dummy *head* node and the invariant holds trivially. Any new node is initially placed at a position reachable from *head* because the node traversed by LOCATEPRED (Algorithm 14) form a consecutive path p' . Note the condition checks in (line 14.3 and 14.9), we have $i < j \implies d_i \leq d_j \forall d_i, d_j \in p'$. Though subsequent insertions may alter the path, they do not unlink nodes from the data structure. The claim follows by noting that an insertion either adds a new node to p' or replaces an existing node in p' . \square

Lemma 3. *At any time, nodes in L , including those marked for logical deletion, form an MDList that complies with Definition 1.*

Proof. Invariant 1 shows that for any node n with dimension d , only children with dimension greater or equal to d is accessible, thus the dimension of a node is always no greater than the dimensions of its children. Follow Invariant 2, logically deleted key-value pairs still occupy valid nodes in the structure before they are physically removed. \square

I now show that the nodes *without* deletion marks form a well-ordered set that is equivalent to P . Invariant 3 states that the ordering property described by Definition 2 is kept at all times.

Invariant 3. $\forall n, n' \in L, n.child[d] = n' \implies n.key < n'.key \wedge \forall i \in [0, d) n.k[i] = n'.k[i] \wedge n.k[d] < n'.k[d]$

Proof. Initially the invariants trivially holds. The linkage among nodes is only changed by insertion, and child adoption. Insert preserves the invariants because the condition checks on line 15.3 and 15.9 guarantee that $\forall i \in [0, dp) pred.k[i] = node.k[i] \wedge pred.k[dp] < node.k[dp]$. Child adoption preserves the invariant because $\forall i \in [dp, dc) node.k[i] = curr.k[i] < curr.child[i].k[i]$. \square

Lemma 4. *Logically deleted nodes appear transparent to traversing operations.*

Proof. Note that a pointer to logically deleted nodes is marked by flag F_{del} , which renders the key-value pair stored in that node obsolete. However, the node's location within the data structure is still consistent with its embed coordinates, making it a valid routing node. The traversing operation treat logically deleted nodes transparently by explicitly clearing the pointer markings on line 14.8. \square

Let us define the set of logically deleted nodes by $S = \{n | n' \in L \wedge n'.child[d] = SETMARK(n, F_{del})\}$.

Following Lemma 3 and 4, the abstract state can then be defined as $P \equiv L \setminus S$.

Linearizability

I now sketch a proof that my algorithm is a linearizable dictionary implementation that complies with the abstract semantics by identifying *linearization points* for each operation. The concurrent operation can be viewed as it occurred atomically at its linearization point in the execution history. Additionally, I use the notion of decision points and state-read points to facilitate reasoning [94]. The decision point of an operation is define 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 define as the atomic statement where the state of the dictionary, which determines the outcome of the decision point, is read.

Theorem 3. A $\text{FIND}(k)$ operation takes effect atomically at one statement.

Proof. A find operation may return v if $\langle k, v \rangle \in P$, or NIL otherwise. The decision point for the first case is when the `while` loop terminates on line 14.2. The node with k must exist in P because the coordinates up to dimension D have been exhaustively examined. The state-read point is line 14.8 when $curr$ is read from child pointers. The subsequent execution branches to line 14.12 after comparing $curr.k$ with k . As the coordinate field k cannot be changed after initialization, the state of the dictionary immediate before passing the state-read point must have been $\langle k, v \rangle \in P$. The decision point for the second case is when the condition check on line 14.9 fails. The state-read point is when the value of $curr$ is read on line 14.8. For both case, the linearization point is the state-read point on line 14.8 □

Theorem 4. An $\text{INSERT}(\langle k, v \rangle)$ operation takes effect atomically at one statement.

Proof. An INSERT operation returns on line 15.24; given a legal key it must succeed by either adding a new node or replacing an existing node. The decision point for both cases to take effect is when the CAS operation on line 15.21 succeeds. The remaining atomic primitives in the child adoption process will be executed at least once and successfully complete through the use of helping mechanism. Equation C.1 holds for the first case because $L = L \cup \langle k, v \rangle$. It holds for the second case because $L = L \cup \langle k, v \rangle \setminus \langle k, v' \rangle$. □

Theorem 5. A $\text{DELETE}(k)$ operation takes effect atomically at one statement.

Proof. If $\langle k, v \rangle \in P$, a successful $\text{DELETE}(k)$ operation updates the abstract state by growing S . The decision point for it to take effect is when the CAS operation on line 17.10 successfully marks

a node for deletion. Equation C.2 holds because $S' = S \cup \langle k, v \rangle \implies P' = P \setminus \{\langle k, v \rangle\}$. The decision points for a DELETE(k) operation to fail are on line 17.7 when it cannot find the node with the target key, and on line 17.14 when the target node has been logically deleted by a competing deletion operation. The state-read point for the former is on line 14.8, which causes LOCATEPRED to abort before reaching the highest dimension. The state-read point for the latter is on line 17.10, where *child* is read. The linearization points for failed deletions are either of the state-read points. In these cases, Equation C.2 holds because $S' = S \implies P' = P$. \square

Lock Freedom

My algorithm is lock-free because it guarantees that for every possible execution scenario, at least one thread makes progress. I prove this by examining unbounded loops in all possible execution paths, which can delay the termination of the operations.

Lemma 5. FINISHINSERTING (Algorithm 16) and LOCATEPRED (Algorithm 14) complete in finite steps.

Proof. We observe that there is no unbounded loop in Algorithm 16. The `for` loop on line 16.4 is bounded by the dimensionality of data structure D , which in practice is a small number. For LOCATEPRED, the `while` loop (line 14.2) is also bounded by D . The inner unbounded `while` loop (14.3) ends after at most $\sqrt[D]{N}$ retries, which is the maximum number of nodes in each dimension. The maximum number of nodes to be examined by LOCATEDPRED is thus $D \cdot \sqrt[D]{N}$. \square

Theorem 6. FIND operations are wait-free.

Proof. The FIND operations invoke LOCATEPRED and does not contain additional loops. The

theorem holds by following Lemma 5

□

Theorem 7. *INSERT and DELETEMIN operations are lock-free.*

Proof. Note that all shared variables are concurrently modified by CAS operations, and the CAS-based unbounded loops (line 15.21, and 17.10 only retry when a CAS operation fails. This means that for any subsequent retry, there must be one CAS that succeeded, which caused the termination of the loop. All reads of child pointer are preceded by FINISHINSERTING, which completes child adoption in finite steps to ensure consistency. Furthermore, my implementation does not contain cyclic dependencies between CAS-based loops, which means that the corresponding operation will progress.

□

REFERENCES

- [1] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. Cbtree: A practical concurrent self-adjusting search tree. In *Distributed Computing*, pages 1–15. Springer, 2012.
- [2] J. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2):75–110, 2003.
- [3] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.
- [4] M. Arbel and H. Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14*, pages 196–205, New York, NY, USA, 2014. ACM.
- [5] H. Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 1–5. ACM, 2010.
- [6] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 65–74. IEEE, 1990.
- [7] J. Bar-Ilan and D. Peleg. Distributed resource allocation algorithms. In *Distributed Algorithms*, pages 277–291. Springer, 1992.
- [8] P. Bernstein and N. Goodman. Timestamp based algorithms for concurrency control in distributed database systems. In *Proceedings 6th International Conference on Very Large Data Bases*, 1980.

- [9] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.
- [10] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.
- [11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.
- [12] 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.
- [13] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 329–342. ACM, 2014.
- [14] 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.
- [15] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *ACM SIGPLAN Notices*, volume 43, pages 304–315. ACM, 2008.
- [16] 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.
- [17] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Citeseer, 1994.
- [18] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing*, pages 229–240. Springer, 2013.

- [19] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 196–205. IEEE, 2013.
- [20] 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.
- [21] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Principles of Distributed Systems*, pages 142–156. Springer, 2006.
- [22] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. 2009.
- [23] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 65–74. ACM, 2011.
- [24] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.
- [25] I. Dick, A. Fekete, and V. Gramoli. Logarithmic data structures for multicores. 2014.
- [26] E. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1(2):115–138, 1971.
- [27] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 343–356. ACM, 2014.
- [28] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 332–340. ACM, 2014.

- [29] 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.
- [30] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *ACM SIGPLAN Notices*, volume 42, pages 291–296. ACM, 2007.
- [31] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
- [32] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 234–254. IEEE, 1979.
- [33] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):90–114, 1989.
- [34] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.
- [35] K. Fraser. *Practical lock-freedom*. PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.
- [36] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.
- [37] 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.

- [38] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic semantic locking. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 385–386. ACM, 2014.
- [39] 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.
- [40] 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.
- [41] K. Gudka, T. Harris, and S. Eisenbach. Lock inference in the presence of large libraries. In *ECOOP 2012–Object-Oriented Programming*, pages 308–332. Springer, 2012.
- [42] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313. ACM, 2008.
- [43] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.
- [44] L. J. Guibas and R. Sedgwick. A Dichromatic Framework for Balanced Trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.
- [45] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.
- [46] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

- [47] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing*, pages 265–279. Springer, 2002.
- [48] A. Hassan, R. Palmieri, and B. Ravindran. Integrating transactionally boosted data structures with stm frameworks: A case study on set. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.
- [49] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *Principles of Distributed Systems*, pages 437–452. Springer, 2014.
- [50] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
- [51] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.
- [52] 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.
- [53] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.
- [54] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.
- [55] 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.

- [56] 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.
- [57] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
- [58] 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.
- [59] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 161–171. ACM, 2012.
- [60] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35. ACM, 2009.
- [61] B. Karlsson. *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.
- [62] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.
- [63] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. *ACM Sigplan Notices*, 45(1):19–30, 2010.
- [64] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5:354–382, 1980.

- [65] 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.
- [66] C. Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.
- [67] N. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 70–81. ACM, 1980.
- [68] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7. ACM, 2004.
- [69] 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.
- [70] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.
- [71] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.
- [72] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.
- [73] M. Michael and M. 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*, pages 267–275. ACM, 1996.

- [74] 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.
- [75] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.
- [76] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328. ACM, 2014.
- [77] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.
- [78] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 192–198. ACM, 1991.
- [79] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 23–32. ACM, 2013.
- [80] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.
- [81] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*, volume 47, pages 151–160. ACM, 2012.

- [82] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Department of Computer Science, University of Maryland, 1990.
- [83] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
- [84] M. Raynal. A distributed solution to the k-out of-m resources allocation problem. *Advances in Computing and Information-ICCI'91*, pages 599–609, 1991.
- [85] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, 1986.
- [86] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, pages 47–56, New York, NY, USA, 2010. ACM.
- [87] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 340–347. ACM, 1984.
- [88] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.
- [89] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPOPP '01, pages 44–52. ACM, 2001.
- [90] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. *ACM SIGPLAN Notices*, 46(10):51–64, 2011.

- [91] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [92] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM, 1999.
- [93] M. Spiegel and P. Reynolds. Lock-free multiway search trees. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 604–613. IEEE, 2010.
- [94] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445. ACM, 2004.
- [95] B. Wicht and C. Evequoz. Binary trees implementations comparison for multicore programming. Technical report, Information and Communications Technology, University of applied sciences in Fribourg, 2012.
- [96] D. Zhang, , and D. Dechev. An efficient lock-free logarithmic search data structure based on multi-dimensional list. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 6 2016. Acceptance Rate: 17%.
- [97] 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.