Concurrent Algorithms in Transactional Data Structures

A THESIS PRESENTED
BY
LILLIAN TSAI
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2017

© *2017 - Lillian Tsai*
All rights reserved.

Thesis advisor: Professor Eddie Kohler Lillian Tsai

Concurrent Algorithms in Transactional Data Structures

Abstract

Software transactional memory (STM) simplifies the challenging, yet increasingly critical task of parallel programming. Using STM allows programmers to reason about concurrent operations in terms of transactions—groups of operations guaranteed to have atomic effect. Our STM system, STO (Software Transactional Objects), outperforms previous STM systems, but its performance still falls short of that of the fastest concurrent programming techniques. This work aims to make STO as fast as these techniques, and, when this appears impossible, to characterize precisely why. We implement and benchmark the most performant concurrent programming algorithms for abstract datatypes within STO's transactional framework. Our results indicate that certain concurrent datatype algorithms lose their scalability and performance in a transactional setting, while other algorithms successfully support transactions without incurring a crippling performance loss. We claim that this discrepancy arises because various concurrent algorithms have different levels of dependency on operation commutativity, and suffer different amounts of commutativity loss in a transactional setting. To support this claim, we pose an alternative operation interface that allows for greater operation commutativity, and, with this interface, re-implement a concurrent datatype whose performance is crippled in a transactional setting. This concurrent datatype is then able to retain its performance and scalability in a transactional setting. We conclude that examining both a datatype's dependency on operation commutativity, and the loss of commutativity of a particular datatype interface in a transactional setting, is enough to determine whether a concurrent, non-transactional data structure will achieve high scalability and performance when integrated with STO.

Contents

1 Introduction **1**

1.1 STM and STO 1

1.2 Motivation 3

1.3 Overview 3

2 Background and Related Work **7**

2.1 Transactional Memory 7

2.2 Transactional Memory Systems 11

2.3 Abstraction-based STMs 13

3 FIFO Queue Algorithms and Analysis **21**

3.1 Transactional Queue Specification 22

3.2 Naive Synchronization Queue Algorithms 22

3.3 Flat Combining Queue Algorithms 25

3.4 Evaluation 36

4 Commutativity and Scalability of Queue Specifications **56**

4.1 Histories 57

4.2 The Scalable Commutativity Rule 60
4.3 Commutativity of the Strong Queue Specification 61
4.4 Commutativity of the Weak Queue Specification 70

5 Hashmap Algorithms and Analysis **77**
5.1 Algorithms 78
5.2 Evaluation 85
5.3 Commutativity of Transactional Hashmaps 101

6 Future Work and Conclusion **107**
6.1 Future Work 107
6.2 Conclusion 108

References **114**

Appendices **116**

A Queue Results **116**
A.1 Cache Misses 117
A.2 Performance of Non-transactional Concurrent Queues 118
A.3 Performance of Various Transactional Queues 120
A.4 Push-Pop Test: Ratio of Pops to Pushes 121
A.5 Abort Rate Results 122

B Hashmap Results **123**
B.1 Cache Misses 124
B.2 Performance: 10K Buckets, 33%Find, 33% Insert, 33% Erase 127
B.3 Performance: 10K Buckets, 90%Find, 5% Insert, 5% Erase 128
B.4 Performance: 125K Buckets, 33%Find, 33% Insert, 33% Erase 130

B.5 Performance: 125K Buckets, 90%Find, 5% Insert, 5% Erase 131
B.6 Performance: 1M Buckets, 33%Find, 33% Insert, 33% Erase 133
B.7 Performance: 1M Buckets, 90%Find, 5% Insert, 5% Erase 134
B.8 Abort Rate Results 136

List of Figures

3.2.1 Queue Operations Interface 23

3.3.1 Invalid interleaving of pop requests in the flat combining queue . . . 30

3.3.2 Abort when performing conflicting pop requests 32

3.3.3 Transactional flat combining pop request execution 33

3.3.4 Abort and cleanup when checking the empty status of the queue . . 35

3.4.1 T-QueueO vs. T-QueueP Performance: Multi-Thread Singletons Test 40

3.4.2 T-QueueO vs. T-QueueP Performance: Push-Pop Test (2 threads) . 41

3.4.3 T-QueueO vs. T-QueueP Push-Pop Test: Speed at which the push- and pop-only threads complete 10 million transactions 42

3.4.4 Non-transactional, Concurrent Queue vs. Transactional Queue Performance: Push-Pop Test (2 threads) 45

3.4.5 Non-transactional, Concurrent Queue vs. Transactional Queue Performance: Multi-Thread Singletons Test 47

3.4.6 Example usage of STO wrapper calls 49

3.4.7 NT-FCQueue vs. NT-FCQueueWrapped Performance: Multi-Thread Singletons Test 50

3.4.8 NT-FCQueue vs. NT-FCQueueWrapped Performance: Push-Pop Test (2 threads) 51
3.4.9 T-FCQueue Performance: Push-Pop Test (2 threads) 53
3.4.10 T-FCQueue Performance: Multi-Thread Singletons Test 53

4.4.1 Weak Queue Operations Interface 71
4.4.2 WT-FCQueue Performance: Multi-Thread Singletons Test 75

5.1.1 Hashmap Operations Interface 78
5.1.2 Bucket Structure of Different Hashmaps 79
5.2.1 Hashmap Cache Misses (Maximum Fullness 5) 90
5.2.2 Hashmap Cache Misses (Maximum Fullness 10) 91
5.2.3 Hashmap Cache Misses (Maximum Fullness 15) 92
5.2.4 Hashmap Performance: 10K Buckets, Maximum Fullness 5 94
5.2.5 T-Chaining vs. T-CuckooKF: Performance as fullness increases (1M Buckets, 33F/33I/33E) 96
5.2.6 T-Chaining vs. T-CuckooKF: Performance as fullness increases (1M Buckets, 90F/5I/5E) 97

List of Tables

3.4.1 T-QueueO vs. T-QueueP Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously 42

3.4.2 Non-transactional Queues vs. T-QueueO and T-QueueP Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously 46

3.4.3 T-FCQueue Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously 52

4.3.1 Strong queue operations that do not commute 62

4.3.2 Examples of strong queue transactions that do not commute 64

4.3.3 Invalid operation interleavings in transactional histories. 66

4.4.1 Examples of valid weak queue transaction histories. 73

5.2.1 Hashmap Abort Rate (Maximum Fullness 10, 33% Finds/Inserts/Erases) 100

5.3.1 Hashmap operations that do not commute 103

5.3.2 Example hashmap transactions that do not commute 104

To my loving family, forgiving friends, and persistent teachers for showing me that life is just like a transaction: the threads of our lives weave together and sometimes one of us has to abort, but we can always retry.

Acknowledgments

This thesis exists because of a single sentence, uttered at the end of a CS61 final exam: “You should do research with me!” I still wonder at my luck to have Professor Eddie Kohler as a mentor and advisor. He has done much more than simply provide invaluable guidance and ideas for my thesis research: over two years ago, Eddie stunned me with his request that I join his research group, and his conviction that I might, in fact, have potential as a researcher in computer science. Without Eddie, this thesis and my hopes as a computer scientist might never have come to be.

Of course, this thesis also owes its life to those who kept me going throughout the past four years (and for some, much longer): my ever-supportive parents who put up with me even at my most irritable; my sister who tolerates my bad thread puns; and my friends who remind me to take breaks, and (perhaps unwillingly) receive the latest news on trees, queues, and hashmaps. Your presence during these past few years—whether it be through contributing ideas to my thesis, or joining me for breakfast—has made all the difference.

I would also like to thank Nathaniel Herman and Yihe Huang, who worked on the STO system with me, as well as my thesis readers, Margo Seltzer and James Mickens. Both Professor Seltzer and Professor Mickens have been remarkable teachers and supporters; I am truly fortunate to have had such amazing mentors during my years here.

1

Introduction

1.1 STM and STO

Parallelism is increasingly critical for performance in computer software systems, but parallel programming remains enormously challenging to get right. Low-level mechanisms for coordinating threads, such as lock-based strategies, are fragile and error-prone. To address this problem, researchers have developed programming tools and methodologies for managing parallelism. Prominent among these is software transactional memory (STM), which allows programmers to write concurrent code

using sequential programming paradigms. By using STM, programmers reason about concurrent operations on shared memory through *transactions*—atomic groups of operations—instead of single operations.

Unfortunately, STM often results in high overhead and is rarely considered practical. To provide transactional guarantees, an STM system tracks the different memory words accessed within a transaction, and ensures that these words are not touched by concurrently executing transactions. Traditional word-based STM tracks all words of memory read or written in a transaction, incurring enormous overhead [5]. To address this problem, a research team at Harvard has produced a novel type of STM, STO (Software Transactional Objects), that greatly improves upon the performance of traditional STM [23]. The system's implementation works at a higher level than most previously developed systems: data structure operations, rather than individual memory words. For example, a word-STM tracks every word accessed in the path from the root during a binary search tree lookup, which introduces significant overhead from transactional bookkeeping and unnecessary conflicts: the transaction will abort if there is a concurrent update to the path, even though the result of the lookup may be unaffected. STO allows datatypes to track datatype-specific *abstract objects* instead of memory words. Thus, during a binary search tree lookup, STO will track only the abstract object corresponding to the parent of the searched-for node, and use only this object to detect a conflict in which the node is modified, removed, or inserted. Compared to a word-STM, STO reduces the number of false conflicts and tracks hundreds of times fewer objects.

1.2 Motivation

The focus of this work is to make STO as fast as the fastest non-transactional, concurrent programming patterns available, and, when this is impossible, to characterize precisely why. Although STO outperforms traditional STM, STO's performance still falls far below that of other concurrent programming paradigms, such as flat combining [17]. STO's library of transactional datatypes—datatypes exposing transactional operations—allows programmers to add transactional memory to their programs. Thus, STO programs are only as fast as STO datatypes. Transactional data structure algorithms are therefore a natural focus for our research: by defining the limits and potentials of these algorithms, we learn how to maximize the performance of the STO system.

1.3 Overview

While the scope of our research includes all the different datatypes supported by STO, this thesis focuses on a few core data structures: queues and hashmaps. We began our research by implementing the initial version of these STO datatypes and, more broadly, developing design techniques for transactional data structures. These techniques define general patterns for designing transactional algorithms, such as how to handle reads and writes of the same object within the same transaction. These patterns, however, do not maximize scalability or performance.

To discover how to maximize performance, we analyze and compare the performance of existing STO data structures against implementations of highly-concurrent data structure algorithms from recent research. These concurrent data structure algorithms strive to maximize scalability and performance in a

non-transactional setting: they ensure that single operations can execute concurrently in multiple threads, but provide no guarantees that a thread can execute a sequence of operations (a transaction) without interruption from another thread (atomically). Transactional systems have strictly more work to do than these non-transactional concurrent systems: both systems must ensure correct synchronization of single operations, but a transactional system must also correctly synchronize multiple operations within a transaction such that they have atomic effect. Thus, the performance of the fastest concurrent datatypes available acts as an upper bound for the performance that transactional STO data structures may reasonably hope to achieve. Our benchmarks highlight which concurrent, non-transactional data structures are the highest-performing, and which parts of the transactional data structure algorithms are bottlenecks and areas for improvement.

We first hypothesize that combining concurrent, non-transactional programming patterns with our transactional design patterns will produce transactional datatypes that greatly outperform our previous implementations. To evaluate this hypothesis, we take the fastest concurrent datatype algorithms and implement them within the STO framework. We discover that the algorithmic changes necessary to move certain highly-concurrent data structures into STO results in a significant decrease in their performance; at times, they even underperform our initial STO data structures that use naive algorithms for concurrency. Furthermore, our experience combining highly-concurrent, non-transactional algorithms with transactional algorithms leads us to conclude that reasoning about scalability in transactional datatypes is inherently different than reasoning about scalability in non-transactional, concurrent datatypes. In particular, to handle transactions, one must reason about invariants regarding the datatype's state during the entirety of the transaction's execution. We formalize this

argument as a commutativity argument, drawing on previous work regarding commutativity in concurrent interfaces [6] and transactional objects [35, 40].

The stateful nature of transactions, and therefore the reduced operation commutativity in a transactional setting, leads us to revise our hypothesis. We claim that certain highly-concurrent datatype algorithms may be intrinsically non-transactional, because the optimizations taken by these algorithms to achieve their high performance are incompatible with providing transactional guarantees. In other words, if the synchronization technique of a concurrent data structure relies on operation commutativity invalidated by transactional guarantees, then the data structure is unlikely to perform well in a transactional setting. If, however, there are few or no added commutativity constraints in a transactional setting, then the added synchronization costs to support transactions do not cripple the concurrent algorithm.

To evaluate our claim, we take highly-concurrent data structures that underperform when integrated with STO, and investigate how these data structures act when we modify their *specifications*—their operation interfaces—to allow for greater commutativity in a transactional setting. With an alternative specification, we achieve a better separation between the logic needed for transactional support, and the logic of the concurrent synchronization algorithm: this allows us to adapt concurrent data structures for transactional settings while retaining their high performance. Comparing data structure performance in the original specification to performance in the alternative specification demonstrates the trade-off between the guarantees certain STO datatypes can provide, and their performance. For example, by preventing a thread from viewing the result of its read until its transaction completes, performance of the datatype more than doubles.

1.3.1 Roadmap

This work is divided into the following parts:

- Background information on transactions and transactional memory, and related work on transactional data structure algorithms and their scalability (Chapter 2)
- An evaluation of different transactional and concurrent algorithms for queue and hashmap data structures (Chapters 3 and 5)
- An explanation based on operation commutativity and invalid histories for why concurrent algorithms for queues fail to retain their performance and scalability in a transactional setting (Chapter 3)
- A demonstration of how the performance loss of queue concurrent algorithms can be avoided through changing the queue operation specification to allow for greater commutativity in a transactional setting (Chapter 4)
- A demonstration of, and explanation why highly-concurrent hashmap algorithms do retain performance and scalability in a transactional setting (Chapter 5)

2

Background and Related Work

2.1 Transactional Memory

A transaction—a group of operations that together form one logical unit of work and have atomic effect—allows a programmer to reason more simply about concurrent access to shared state. The concept first developed in database theory, trickled into file systems, and then expanded to other domains with the development of hardware transactional memory (HTM) in 1986 and software transactional memory (STM) in 1995 [14]. A transaction is commonly defined by

the ACID properties: atomicity, consistency, isolation, and durability. Transactional memory (TM) [14, 21] guarantees transactional properties in shared memory; this differs from transactions in a database which (traditionally) work with data on disk. TM is therefore unconcerned with the durability property: memory does not persist on some permanent medium. However, TM must still adhere to the remaining three ACI properties:

Atomicity ensures that, if a transaction commits, all changes made by the transaction are instantly visible to other actors performing transactions (for example, threads modifying shared memory or processes modifying a database). If a transaction aborts, none of the changes made by the transaction are visible to any other actor. Thus, either all or none of the operations of the transaction should appear to succeed.

Consistency guarantees that a transaction begins and ends with the state of the database or data structure satisfying particular (data structure-specific) invariants: for example, an invariant could be that the data structure is left in a state without duplicate entries.

Isolation ensures that a transaction's execution appears to be isolated from any other transaction's (possibly simultaneous) execution. This allows for transactions to be *serializable*, which means that one can find an ordering of committed transactions that satisfies the observed history of results. In this ordering, it should appear as if operations within one transaction are never interleaved with operations in another transaction.

Transactional memory transactions also are *linearizable* [22]: all transactions performed at a later clock time than a committed transaction observe the changes made by the committed transaction. This allows programmers to easily determine the order and effects of transactions.

2.1.1 Transaction Execution and Commit Protocol

To use a transactional system, a programmer indicates the start and end of the transaction. When the transaction finishes, it runs a commit protocol. The basic commit protocol has four phases [14]:

1. Lock: Acquire the necessary locks to ensure protected access to any state the transaction intends to modify.
2. Check: Any state observed by the transaction must be checked to ensure that the state has not changed in a way that would invalidate the results of the transaction. If the check fails, the transaction aborts. If the transaction passes the check phase, the transaction will commit at this time.
3. Install: All modifications of the transaction are installed.
4. Cleanup: Release the locks acquired to protect any state modified by the transaction. If the transaction aborts during Phase 2 (check) or at any time during the transaction's execution, cleanup also removes any changes that the transaction may have made to the state.

The exact details of the commit protocol depend on whether the transactional memory system is optimistic and/or pessimistic, and whether it uses eager and/or lazy updates [14]. Optimistic transactional memory systems and pessimistic transactional memory systems both track the transaction's intended changes in the transaction's *write set* and any state observed during the transaction in the transaction's *read set*. An optimistic system checks for invalidated values in the read set only at commit time. If all values are valid, the system performs the changes in the transaction's write set and the transaction is marked committed. Otherwise, the transaction aborts and the system ensures that the transaction leaves no visible effects (potentially rolling back any changes the transaction has made). An

optimistic system therefore assumes (optimistically) that no other thread will conflict with another thread's transaction, and executes all operations of the transaction before it checks if any conflict has indeed occurred. A pessimistic transactional system instruments every read and write with additional checks to see if another thread is simultaneously accessing the same part of memory and creating a conflict. If any such conflicts are detected during the read or write, then the thread either aborts or stalls until the conflicting transaction completes by either committing or aborting.

Transactional memory systems also practice eager updates or lazy updates. A system is eager when it updates shared memory during the transaction's execution. It maintains an "undo" log of changes, which is used if the transaction aborts and the changes need to be undone. A system is lazy if it performs no updates to shared memory until the transaction commits: all intended changes during execution are buffered by the system, or written to a log and applied at commit time. Systems can range from fully eager to fully lazy, with most practicing a mix of the two techniques.

Transactional systems can also provide the *opacity* property [13], a property that guarantees that the code running transactions will never observe an inconsistent memory state. In other words, all the transaction's reads must be consistent with a single snapshot of memory at some point during the transaction. Opacity ensures that potential failures such as infinite loops and invalid pointer dereferences will never occur. With opacity, a transaction will abort immediately upon observing an inconsistent state that would cause the transaction to fail at commit time. Note that pessimism does not guarantee opacity: a transaction performing read-only operations will not acquire locks on the data structure, and may therefore observe an inconsistent state at some point during its execution. In

certain systems such as STO, providing opacity requires keeping track of a global transaction ID (TID). This global TID is used to check the consistency of previously read items when the transaction observes a memory state. The TID is updated whenever a transaction commits and modifies memory. Opacity hinders the performance of the code running transactions because this TID must be accessed whenever a transaction accesses memory (to check the state of previously read items) and when a transaction commits. In our benchmarks, we disable opacity in order to measure our transactional system at its maximum achievable performance.

2.2 Transactional Memory Systems

Transactional memory can be implemented in both hardware and software. Although hardware transactional memory (HTM) naturally outperforms software transactional memory (STM), a purely hardware TM has several inherent limitations. HTM will fail when the working sets of the transactions exceed hardware capacities; for example, the buffer used to track read and writes of the transaction is restricted in size. In addition, HTM lacks flexibility because the granularity of reads and writes is at the word level [41]. Nevertheless, with the increasing support for HTM in computer hardware [24], integrating STM with HTM offers performance improvements over what STM alone can achieve. We see integration with HTM as potential future work to improve the performance of our data structures. For example, we might take inspiration from the hybrid hardware-software TM system first presented by Lie in 2004 [30]: a transaction would first run in hardware, and if it fails, it would retry as a software transaction in the transactional data structure. This would reduce the overhead incurred from transactional tracking, since most of the overhead can be avoided if the transaction completes in hardware.

The STO system [23] is one of several STM systems. For simplicity, we can generalize STMs into three groups: word/object-based STMs, which track individual memory words or objects touched during a transaction; STMs that expose non-transactional APIs for performance gains; and abstraction-based STMs which track items based on abstract datatypes.

TL2 [7] and LarkTM [42] are highly-optimized word-STMs that track memory words touched during a transaction. SwissTM [8] is also a word-STM, but increases performance by tracking memory in 4-word groups, resulting in less overhead than tracking individual memory words. There have also been object-based STMs which track objects instead of memory words, but incur extra cost by shadow copying any objects written to within the transaction [20].

Open nesting [33], elastic transactions [10], transactional collection classes [4], early release [19], and SpecTM [9] are techniques for implementing transactions that, like STO, speed up transactional performance by reducing bookkeeping costs and the number of false conflicts. However, these techniques expose non-transactional APIs to the programmer. This complicates, rather than simplifies, concurrent programming. For example, transactional collection classes remove unnecessary memory conflicts in data structures by wrapping the datatype with semantic locks; this requires designing multi-level, open-nested transactions, and presents a much more complicated framework than does STO, which allows the datatype to be designed specifically to support transactions.

STO falls in a category of STM systems that use abstraction to improve STM performance [3, 12, 16, 18]. These systems expose a transactional API to programmers in the form of transactional abstract datatypes that are written on top of an STM. However, systems other than STO build their data structures on top of traditional word-STMs, whereas STO builds data structures on top of an abstract

STM which tracks abstract items defined by each data structure. This lets STO improve performance beyond that of previous systems.

Our work focuses on abstract datatypes and how they perform within transactional settings. We now take a closer look at STO and other abstract STM systems that expose an API in the form of transactional data structures.

2.3 Abstraction-based STMs

STO consists of a core system that implements a transactional, optimistic commit-time protocol, and an extensible library of transactional datatypes built on top of this core. While programmers can use the many transactional datatypes already implemented in STO (ranging from queues to red-black trees), programmers can also use STO's transactional framework to add transactional support to other datatypes based on their particular semantics. STO allows datatypes to track conflicts and changes using datatype-specific *items*, thereby reducing bookkeeping costs by exploiting the semantics of the particular datatype.

Many STO datatypes enforce transactional correctness using *versions* that correspond to items, which represent some part of the data structure state. A version acts as a lock on the data structure: in order to update the data structure state, a thread must first lock the version corresponding to the part of the state being modified. A version tracks changes to the data structure by monotonically increasing whenever a thread modifies the corresponding data structure item. Thus, any version seen by a thread is equivalent to some previous or current state of the data structure. When a transactional operation performs a read of some data structure state, it adds a read of the corresponding version value. This read version value is checked when the transaction commits to ensure that the observed data

structure state is still valid: if the version value has changed, then the data structure state has changed as well. The version's value from the *first* time we perform a read of the version is validated at commit time; this ensures that every observed state since the start of the transaction is still valid.

STO also allows datatypes to use a variety of different concurrency control algorithms. For example, STO's transactional hashmap defines abstract items for each bucket, which are invalidated only when the bucket size changes. This means that transactions conflict only when modifying or reading the same bucket, allowing scalable access to the data structure. In addition, at most one item is added to the read or write set per operation, which can be orders of magnitude fewer than the number of items tracked by a word- or object-based STM. STO allows datatypes to define their own strategies for transaction execution: a datatype can insert elements during transaction execution (eagerly) or wait until commit time to insert (lazily). The specifics of the commit protocol are implemented as datatype callbacks. This allows a datatype to use pessimistic strategies for certain operations, while using optimistic strategies for others. STO is therefore a flexible hybrid of optimistic/pessimistic and eager/lazy versioning strategies.

Boosting [18] is a method to convert concurrent (non-transactional) data structures into transactional data structures. Like STO, boosting determines conflicts between transactions by relying on a particular data structure's semantics. Instead of allowing each datatype to define read and write set items, however, boosting maps a datatype's operations to abstract locks. If two operations do not commute—i.e., swapping the order of their invocations affects the final state of the data structure or the responses returned by the operations—then the abstract locks for the operations will conflict. For one of the operations to be performed, both the abstract lock for that operation and the abstract lock for the conflicting operation

must be acquired. Thus, the granularity of synchronization of the original linearizable data structure is only achievable for commutative operations in its boosted form. Because a transaction may abort, boosting practices undo logging and requires that each operation has an inverse. These constraints mean that boosting is not applicable to all data structures, but allow boosting to be particularly useful for data structures like sets. Unlike boosting, STO allows us to work with data structures whose operations have no inverse. Boosting is an inherently pessimistic strategy, using eager versioning, whereas STO allows for a hybrid approach that can improve performance.

Optimistic boosting [16] is a technique meant to improve the performance of boosting. It proposes an optimistic approach in which acquisition of the abstract locks is delayed until commit time. During execution, semantic items are added to the read and write sets of the transaction and validated at commit time; if all reads are valid, then the appropriate abstract locks are acquired and modifications applied to the data structure. Because execution is optimistic and abstract locks are not eagerly acquired at the higher, "boosted" level, the underlying concurrent data structure can be lazy and wait until commit time to execute operations. This adds support for operations that may not have an inverse. However, optimistic boosting has not been shown to be effective in practice. STO provides a more flexible, hybrid transactional framework and outperforms optimistic boosting.

Automated locking [12] takes a similar approach to boosting by pessimistically acquiring abstract locks corresponding to each operation. It differs from boosting because it also takes a datatype-specific commutativity specification of conditions in which operations commute. The commutativity specification of an abstract datatype is compiled into a symbolic set (called a "locking mode") that is used to prevent conflicting operations from being run concurrently. If two operations

commute, the “locking mode” allows them to run concurrently. This approach optimizes and automates the creation of abstract locks for an abstract data structure. A similar approach may help STO data structures choose which abstract read and write items to track during a transaction to avoid false conflicts.

Predication [3] is a technique that maps operations to “predicate words” contained in a shared-memory table managed by the STM. Each predicate word represents a property of the data structure. For example, looking up an element e in a set would be associated with an `{in_set?(`e`)}` predicate word, and the STM would add reads or writes of the predicate word when e is removed or added to the set. Unlike boosting, transactional predication achieves semantic conflict detection without keeping an undo log. However, predication must insert predicate words into the table for absent as well as present lookups, causing a garbage collection problem that STO and other systems do not face. Transactional predication also focuses on making transactional data structures perform equally as well as highly-concurrent data structures in non-transactional settings. This is orthogonal to our work here, and can be a future line of optimization for STO data structures.

The Transactional Data Structures Libraries (TDSL) [38], like STO, offer collections of transactional data structures for programmers to use. Similar to STO, each TDSL data structure executes transactions with a customized mix of pessimistic, optimistic, eager, and lazy strategies. This allows for optimizations that rely on the specific data structure’s semantics. Unlike STO, data structures in TDSL are also optimized for single-operation transactions (which we see as an orthogonal line of work). While TDSL contains only a queue and a skiplist, STO has implementations of many other data structures in transactional settings, including a hashmap, list, priority queue, and red-black tree. Our work in this thesis draws upon some of the algorithm designs for the queue implemented in TDSL.

This thesis investigates the integration of several non-transactional, concurrent data structures with STO. In particular, we test and modify different concurrent queue and concurrent hashmap algorithms, which we describe in more detail in Chapters 3 and 5. Similar work has been done with lazy sets [15], in which transactional support is added to a lazy concurrent set. We extend this work to other data structures, and investigate further how to achieve performance close to highly concurrent, non-transactional data structures.

2.3.1 Commutativity in Transactional Data Structures

Both STO and the other methods of integrating concurrent abstract datatypes with STM declared above build upon the ideas introduced by Weihl in the late 1980s [40]. Weihl defines optimal *local atomicity* properties that datatypes must satisfy in order for transactions to be serialized, and uses these properties to provide an upper bound on the amount of concurrency achievable in transactional datatypes. These local properties are derived from algebraic properties of the data structure, such as commutativity of particular operations. Weihl also demonstrates an inherent relation between commutativity-based concurrency control and transactional recovery algorithms. Unlike our work with STO, however, Weihl does not focus on the concrete implementation of transactional datatypes.

Schwarz and Spector [35] introduce a theory for ordering concurrent transactions based on the semantics of shared abstract datatypes and the dependencies of different datatype operations. For each operation, the programmer specifies the operation's preconditions, postconditions, and invariants. To describe interactions between operations in a transaction, the programmer additionally provides an interleaving specification. This defines dependency relations between

datatype operations. From this set of dependency relations, Schwarz and Spector can define the limits of concurrency for the datatype by drawing upon results from Korth [26] that show that when two operations commute (i.e., have no inter-dependencies), the order in which the operations are ordered does not affect serializability. We discuss this work further in Chapter 4.

Schwarz and Spector also demonstrate that increased concurrency can be achieved by weakening the serializability of transactions: once the semantics of a datatype have been taken into account, the remaining constraints on concurrency come from enforcing transactional guarantees [28]. A transactional datatype that makes weaker ordering guarantees than serializability is able to achieve a level of concurrency closer to that implied by the semantics of its operations. They exemplify this with a WQueue design: a higher-concurrency queue with modified semantics that preserves a weaker ordering property than serializability. However, their paper focuses on the theoretical result instead of the implementation, and is not concerned with explicit synchronization algorithms for the queue. Our work in Chapter 4 with the weak transactional queue builds off this idea of weaker transactional guarantees, but modifies the queue operation interface rather than weakening the serializability requirement of transactions.

Badrinath and Ramamritham [2] define *recoverability*, a weaker notion of commutativity that can achieve enhanced concurrency. An operation q is recoverable with respect to another operation p if the observable effects of q are the same regardless of whether p executes *immediately* before q, or whether p executes some time prior to, but not immediately before q. If p and q are in separate transactions and do not commute, but q is recoverable with respect to p, then both operations can be eagerly applied to the data structure under the condition that if p is applied before q, then the transaction calling p must commit before the

transaction calling q. This condition implies a *commit dependency* between the two transactions. If the commit dependencies generated by all recoverable operations form a cycle between parallel-executing transactions, one of the transactions in the cycle must abort and undo its operations. However, only one transaction in any commit dependency cycle will have to abort. This prevents issues such as cascading aborts, in which one transaction's abort causes the abort of another transaction, which causes the abort of a third transaction (and so on).

Thus, recoverability enforces operation orderings at transaction commit time rather than execution time: if one operation is recoverable with respect to another, both operations can be executed in parallel so long as the transactions commit in the correct order. If the transactions cannot both commit (because of a dependency cycle), then one transaction must abort and undo its operations. Our work in Chapter 4 similarly examines how eagerly executing operations induces an ordering on transactions' commit operations, but discusses commutative, instead of recoverable, operations.

Kulkarni et al. [27] define the notion of a commutativity lattice (predicates between pairs of methods) to reason about commutativity in a data structure. The Galois system, upon which this idea is tested, provides a framework in which the programmer defines a commutativity lattice for individual data structures and, by exploiting commutativity, improves the performance of irregular parallel applications. Galois, however, focuses on constructing commutativity checkers instead of serializing transactions.

Commutativity work has also played a large part in optimizing distributed transactions. Mu et al. [32] introduce a system, ROCOCO, that first distributes pieces of concurrent transactions across multiple servers. These servers then determine dependencies between their pieces of concurrent transactions based on

the commutativity of operations in the transactions. Transaction execution is delayed until commit time, when its corresponding dependency information is sent to all servers via a coordinator, allowing the servers to re-order conflicting pieces of the transaction and execute them in a serializable order. This reduces aborts and unnecessary conflicts. Finally, commutativity has also been explored in network consistency algorithms and conflict-free replicated data types (CRDTs) [37].

3

FIFO Queue Algorithms and Analysis

This chapter investigates different concurrent and transactional algorithms for queues in order to draw conclusions about concurrent queue algorithms in transactional settings. We begin with an overview of concurrent and transactional queue specifications and algorithms. We then evaluate how these queues perform on several microbenchmarks. Given our results, we conjecture that highly-concurrent queue algorithms are inherently non-transactional: the optimizations taken by these algorithms rely on data structure state and behaviors that must be modified to support transactions. In other words, the synchronization mechanisms of

highly-concurrent queue algorithms interfere with the mechanisms that STO uses to provide transactional guarantees.

3.1 Transactional Queue Specification

A concurrent queue supporting operations push and pop must adhere to the following specification:[1]

- No duplicate pops: a value is popped off the queue only once.
- No duplicate pushes: a value is pushed onto the queue only once.
- Correct ordering: values are popped in the order in which they are pushed.

A transactional queue adds the following invariants to the specification. There must be a serial order of all transactions such that, within one transaction:

- Any two pop operations pop consecutive values in the queue starting from the head of the queue. This includes values pushed onto the queue by previous push operations in the transaction.
- Any two push operations push consecutive values at the tail of the queue.

To satisfy these invariants, transactional data structures must support *read-my-writes*. This is when the effect of a transactional operation depends on the effects of previous operations within the same transaction.

3.2 Naive Synchronization Queue Algorithms

STO provides two transactional FIFO queues that support push and pop operations with the interface shown in Figure 3.2.1. These transactional queue algorithms are

[1] In the following discussion of our queue algorithms, we omit the discussion of the front operation to simplify reasoning about the state of the queue. An appropriate algorithm for front can be easily inferred from that used for pop.

```
// push adds value v onto the tail of the queue
// always succeeds
void push(const value_type& v);

// pop removes a value from the head of the queue
// succeeds if the queue is nonempty
bool pop();
```

Figure 3.2.1: Queue Operations Interface

designed with transactional correctness primarily in mind, and concurrency as a secondary concern.

3.2.1 T-QueueO

T-QueueO is an optimistic transactional queue, and implements a bounded-length transactional queue using a circular buffer. It supports transactional operations push and pop, and is implemented using optimistic concurrency control (OCC). This means that two threads can simultaneously access the queue while executing their transactions. At commit time, the threads check if the queue has changed in a way that would invalidate their transactions. T-QueueO exposes two versions for checking the state of the queue: the head version and the tail version. The head version, which tracks the state of the head, is used to check if another thread has popped from the queue, and the tail version, which tracks the state of the tail, is used to check if another thread has pushed onto the queue.

A transactional push adds to an internal `write_list`, which holds a thread-local list of values to be pushed onto the queue at commit time. At commit time, the tail version acts as a lock to prevent any other thread from pushing onto the queue. After locking the tail version, the thread pushes all values on the `write_list` onto the queue and increments the tail version. If the queue is full, the

queue will raise an assertion error. If a transaction performs only pushes, then the transaction will always commit unless the bounded size overflows: a push does not observe any property of the queue, such as the value at the head of the queue or the emptiness of the queue.

A transactional pop first checks if the queue will be empty by observing the current state of the queue, and by taking into account how many values the current transaction is already intending to pop. If the queue will not be empty, the pop returns `true`. When the thread commits, it must ensure that the head of the queue has not been modified by another thread. This is done by comparing the value of the head version at the time of the pop with the value at commit time.

If the queue will be empty, the thread checks for earlier pushes by the same transaction: if the thread intends to push a value onto the queue in this transaction, then the thread removes the value from its `write_list` and returns `true`. Otherwise, the return value of the pop is `false`. At commit time, the thread must check that the queue is still empty by validating the value of the tail version, which increments each time a value is pushed onto the queue. When a transaction that performs one or more pops commits, it locks the head version (ensuring atomic access to the head of the queue), removes a value from the head of the queue for every successful transactional pop call, and increments the head version.

3.2.2 T-QueueP

T-QueueP is also a bounded-length transactional queue with a circular buffer supporting operations push and pop. T-QueueP's algorithm is a hybrid design, using T-QueueO's optimistic algorithm for pushes, and pessimistic locking for pops. This takes inspiration from the transactional queue from the Transactional Data

Structures Libraries [38]. We hypothesize that T-QueueP will perform better than T-QueueO, based on the TDSL benchmarks results showing that the TDSL pessimistic transactional queue achieves better performance than T-QueueO.

Adding pessimistic locking is done by locking the queue when any pop (a naturally contentious operation) is invoked. The queue is then unlocked only after the transaction is complete. This ensures that no thread will execute a pop that will invalidate another thread's transactional pop. However, a push in T-QueueP follows the same protocol as a push in T-QueueO. Because execution of a push is lazy and delayed until commit time, a transactional push can execute without invalidating another transaction. A push therefore does not acquire a lock at execution time, but rather only needs to lock while installing all the transaction's pushes at commit time.

Because a transactional pop locks the queue, there are no conflicts at commit time after a thread performs a transactional pop. A thread only aborts if it fails to obtain the lock after a bounded period of time. The one version, "queueversion," acts as the global queue lock.

3.3 Flat Combining Queue Algorithms

Given the relatively slow performance of T-QueueO and T-QueueP compared to the best-performing highly-concurrent queue algorithms (see Section 3.4.5), we looked for a highly-concurrent, non-transactional queue algorithm that might be promising to use in STO's transactional framework. After running several benchmarks (see Figure 3.4.5), we found the most promising to be the flat combining technique, which not only outperforms other queue algorithms, but also addresses several of the bottlenecks we observe in T-QueueO and T-QueueP.

3.3.1 Non-Transactional Flat Combining Queue

Flat combining, proposed by Hendler et al. in 2010 [17], is a synchronization technique that is based upon coarse-grained locking and single-thread access to the data structure. The key insight is that the cost of synchronization for certain classes of data structures often outweighs the benefits gained from parallelizing access to the data structure. These classes of data structures include high-contention data structures such as stacks, queues, and priority queues. Created with this insight, the flat combining algorithm proposes a simple, thread-local synchronization technique that allows only one thread to ever access the data structure at once. This both reduces synchronization overhead on points of contention (such as the head of a queue) and achieves better cache performance by leveraging the single-threaded access patterns during data structure design.

A flat combining data structure has three parts: (1) a sequential implementation of the data structure, (2) a global lock, and (3) per-thread records that are linked together in a global publication list. A thread uses its record to publish to other threads the specifics of any operation it wants to perform; the result of the operation is subsequently written to and retrieved from the record.

When a thread T wishes to perform an operation O:

1. T writes the opcode and parameters for O to its local record. Specifically for the queue, T writes <PUSH, value> or <POP,()> to its local record.
2. T tries to acquire the global lock. Depending on the result:
   (a) T acquires the lock and is now the "combiner" thread. T continually iterates through the publication list and applies all the thread requests in the list in sequence, writing both the result and an <OK> response to each requesting thread's local record. T stops this process when the number of

iterations in which no operations are performed increases above 50%, and then releases the lock.

(b) T failed to acquire the lock. T spins on its record until another thread has written the result to T's record with the response `<OK>` or the lock is released, in which case T acquires the lock and becomes the combiner thread.

When used to implement a concurrent queue, flat combining proves to be an effective technique for handling the contention caused by parallel access on the head and tail of the queue. In addition, the flat combining queue uses a sequential queue implementation with “fat nodes” (arrays of values, with new nodes allocated when the array fills up), which both improves cache performance and allows the queue to be dynamically sized. Both T-QueueO and T-QueueP suffer from the contention and cache performance issues pointed out in the flat combining paper, leading us to believe that the alternative synchronization paradigm offered by flat combining may improve the performance of a transactional queue just as it does for a concurrent one.

3.3.2 Transactional Flat Combining Queue

Recall that, in addition to the requirements for a correct concurrent queue, a transactional queue must guarantee that there exists a serial order of all transactions such that, within one transaction, any two pops pop consecutive values in the queue starting from the head of the queue, and any two pushes push consecutive values at the tail of the queue. This means that we must consider the order in which threads' requests are applied to the queue to be able to create a transactionally correct flat combining queue. For example, let a transaction in thread $T1$ be `{pop, pop}` and a

transaction in thread $T2$ be `{pop}`. The combiner thread sees that $T1$ has published `<POP,()>` and $T2$ has published `<POP,()>` to the publication list. The combiner thread then applies `T1:Pop` (popping the head of the queue) and `T2:Pop` (popping the second value on the queue). When the next combining pass executes, the combiner thread will see that $T1$ has published `<POP,()>` again to the queue. However, performing $T1$'s second pop violates transactional guarantees: the two popped values in $T1$'s transaction will not be consecutive. This sequence is shown in Figure 3.3.1. $T1$ must now abort, which means that $T2$'s pop becomes invalid: it popped the second-frontmost value of the queue, rather than the head of the queue.

Detecting these invalid orderings requires two important changes to flat combining (we describe the rationale for these changes in Chapter 4):

1. A push cannot be applied to the queue during a transaction's execution, and must instead be performed when a transaction commits. A push still only needs to make one flat combining call, because pushes do not need to access the queue until commit time: a push returns no observable value to the caller.
2. An uncommitted pop in a thread's transaction must be unobservable by any other thread. This can be implemented in two ways:
    (a) The algorithm can delay a transaction's pops until commit time. This then means the algorithm must track which values in the queue are going to be popped by the transaction. This prevents duplicate pops and detects if the queue will be "empty" by tracking how many values will be popped off the queue during this transaction. If another thread performs a pop or push during the transaction's lifetime, this can cause the transaction to abort: the "empty" status of the queue at commit time may now be inconsistent with what the transaction saw during execution. A pop therefore accesses the queue twice, once at execution time to see if

the pop can succeed, and again to check and commit the pop at commit time.

(b) The algorithm does not execute flat combining requests from other threads until the transaction has committed or aborted (a pessimistic approach). Because only this thread can execute commands, pops can be performed eagerly at execution time, and restored to the head of the queue if the transaction aborts. This can be implemented either by aborting the other threads' transactions, or by forcing the other threads to block or spin. A pop therefore accesses the queue twice, once at execution time to execute the pop, and again to release the queue after committing so that other threads can execute flat combining requests.

Because both approaches for a pop access the queue twice, a pop now requires two flat combining requests instead of the single request required by the non-transactional flat combining queue. We choose to implement approach (a) based on microbenchmarks that show that approach (b) is both more complicated and less performant.

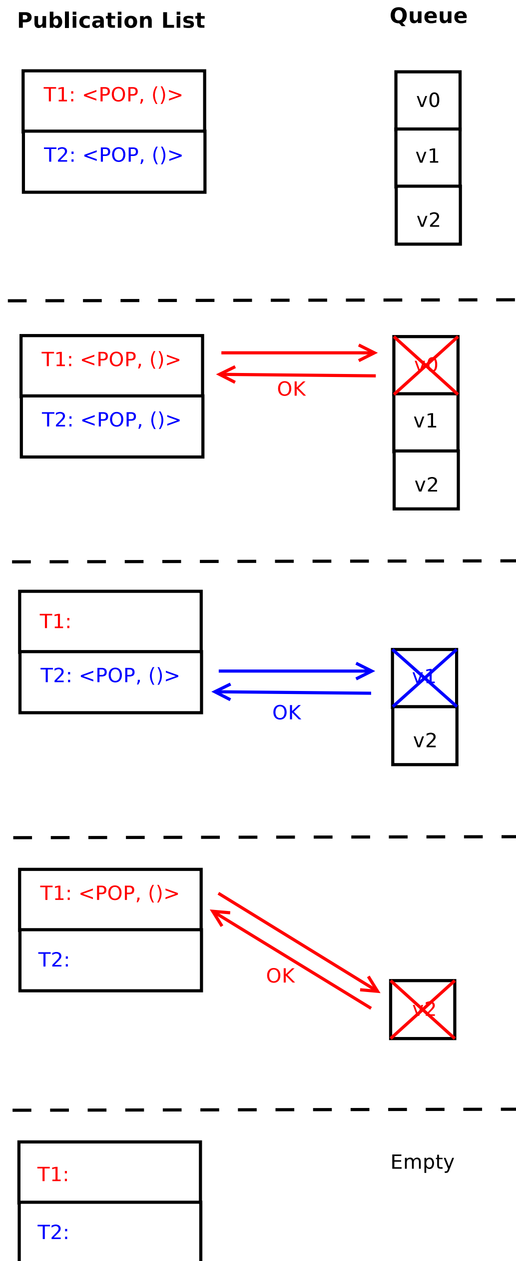


Figure 3.3.1: This sequence of operations applied to the queue is a possible interleaving that will cause both transactions $T1$ and $T2$ to abort. A transactional flat combining queue must ensure that all pops in one transaction are consecutive and are of the head of the queue. In this example, $T1$ performs two pops and $T2$ performs one pop. The combiner thread applies $T1$'s first pop, then $T2$'s pop, and finally $T1$'s second pop. Because $T1$'s pops are not popping consecutive values off the queue, $T1$ is an invalid transaction, and must abort when it commits (not shown). When $T1$ aborts, the head ($v0$) of the queue is restored. $T2$ now becomes an invalid transaction because it popped a value that was not the head of the queue.

We now describe the new algorithms for push and pop. We change the types of requests a thread can publish to its record on the publication list. Recall that the original flat combining queue supports two requests: `<PUSH, value>` and `<POP, ()>`. The transactional queue supports the follow requests:

- `<PUSH, list>` : push a list of values onto the queue
- `<MARK_POP, thread_id>` : mark a value in the queue as "to be popped" by this `thread_id`
- `<DEQ, thread_id>` : dequeue all values in the queue that are marked "to be popped" by this `thread_id`
- `<EMPTY?, thread_id>` : check that (1) the queue, after popping all items marked by this `thread_id`, is empty, and that (2) no other transactions have performed a sequence of concurrent updates that increased the queue size, but then returned it to empty.
- `<CLEANUP, thread_id>` : unmark all values that are marked with this `thread_id`

As with T-QueueO and T-QueueP, a push within a transaction adds to an internal `write_list`. At commit time, the thread will post a `<PUSH, list>` request with the `write_list` passed as the argument.

A pop is implemented with a pessimistic approach. Performing a pop within a transaction invokes the `<MARK_POP, thread_id>` request. The combiner thread, upon seeing a `MARK_POP` request, looks at the first value at the head of the queue. If this value is marked with another thread's `thread_id`, the combiner thread returns `<ABORT>` to the calling thread. This scenario is shown in Figure 3.3.2.

If the value is not marked, the combiner thread marks the value with the caller's `thread_id` and returns `<OK>`. Note that in this scenario, no other thread will

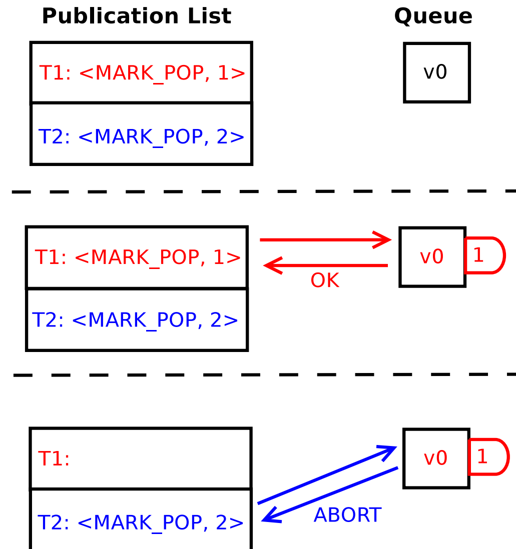


Figure 3.3.2: $T1$ and $T2$ both attempt to mark the head of the queue with their `thread_id`. $T1$'s request is applied first, and marks the value $v0$ with the `thread_id` 1. The combiner thread attempts to apply $T2$'s request and sees $T1$'s `thread_id` marking the head of the queue. It then signals $T2$ to abort.

be able to mark values in the queue until the calling thread commits or aborts: another thread will abort if it sees that the head value is marked by the calling thread's `thread_id`.

If the value is neither marked by another thread nor unmarked, then the value must already be marked with this thread's `thread_id`. The combiner thread then iterates sequentially through the queue, starting from the head, until it reaches a value not marked by the calling thread's `thread_id`. It then marks the value with the caller's `thread_id` and returns `<OK>`. Upon receiving the response, the calling thread adds a write to a `pop_item` to tell the thread to post a `<DEQ, thread_id>` request at commit time. This request will tell the combiner thread to remove the popped value from the queue. This procedure is shown in Figure 3.3.3.

If the queue is empty, or all values are marked with the caller's `thread_id`, the

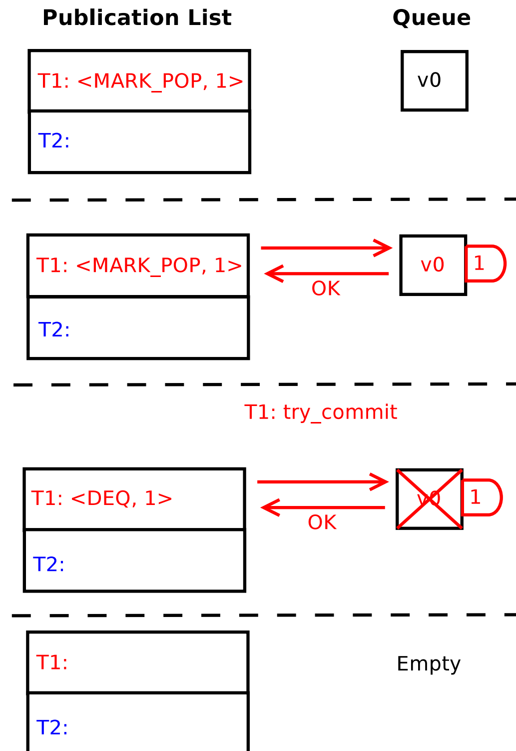


Figure 3.3.3: $T1$ performs a pop by sending a MARK_POP request, and marks the value in the queue with its thread_id 1. At commit time, $T1$ actually performs the pop by sending a DEQ request.

combiner thread will return <EMPTY>, which is remembered by the calling thread. An <EMPTY> response requires that the size of the queue be checked at commit time. If the calling thread has previously performed a push in the same transaction, the transaction removes the head of the write_list; in this case, the transactional pop returns true. Otherwise, the pop returns false.

The <EMPTY?, thread_id> request is posted during the check at commit time when a thread tries to commit a transaction that observed an empty queue at some point in its execution. This happens when the thread observes an <EMPTY> response to a <MARK_POP> request during the transaction's execution. If EMPTY? returns true, then the queue is empty at commit time. No concurrent modifications to the queue have been performed since the time the thread saw an empty queue while

performing a `<MARK_POP>`, and the transaction can safely commit. If instead `EMPTY?` returns `false`, then the thread knows the queue is no longer empty—another thread has pushed values onto the queue—and this thread's `<MARK_POP>` result is invalid. The transaction must therefore abort. This scenario is shown in Figure 3.3.4.

Note that in order for the check of `EMPTY?` to correctly verify the empty state of the queue at commit time, it needs to check both that the queue is empty, and that there have been no concurrent updates by other transactions that add values to the queue and then return the queue to an empty state. Our implementation checks for such concurrent updates using an *empty predicate version*. This version is updated when a transaction installs a push, and a transaction that observes an empty queue during a pop adds a read of this version. This version is checked by the `EMPTY?` call at commit time, which will return `false` if the version has changed since the time it was observed.

The `<CLEANUP, thread_id>` request is posted when a thread aborts a transaction and must unmark any values in the queue that it had marked as pending pops. The combiner thread iterates through the queue from the head and unmarks any values with the `thread_id`. An example of this is shown in Figure 3.3.4.

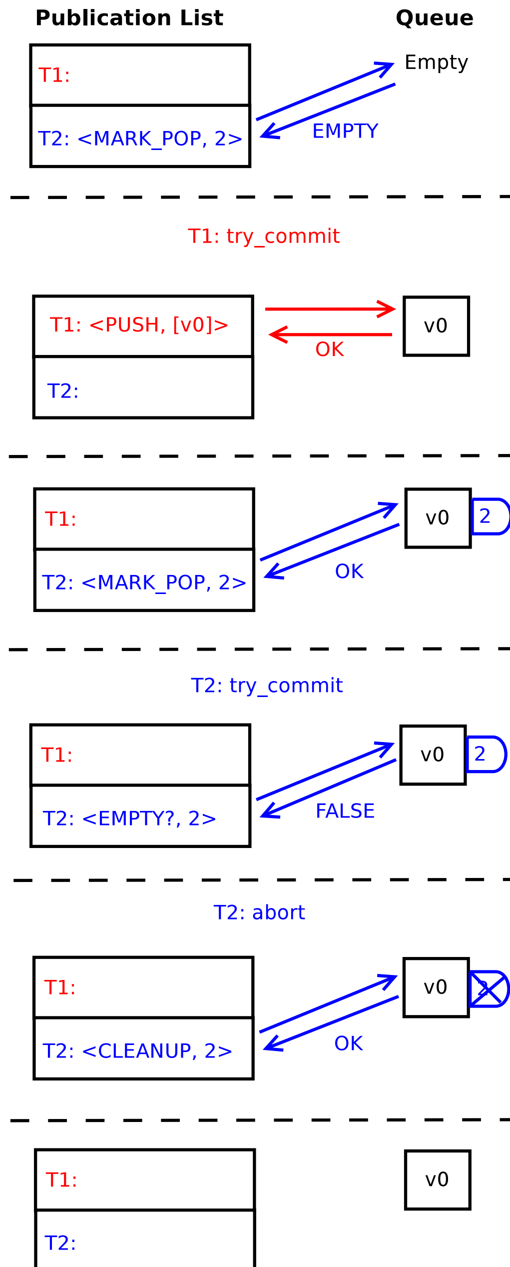


Figure 3.3.4: This sequence shows how a transaction can abort when checking `<EMPTY?>` in the transactional flat combining queue. $T2$ tries to pop from an empty queue, and sees the queue is empty. This means that when $T2$ commits, $T2$ will have to check if the queue is empty. $T1$ commits its transaction and pushes $v0$ onto the queue (recall that a push only executes at commit time). $T2$ then tries to pop another value off the queue and sees $v0$, marking it with its `thread_id` 2. $T2$ tries to commit, but observes that the queue is no longer empty: $T2$ must abort. When $T2$ aborts, it must clean up any markers it left in the queue.

3.4 Evaluation

3.4.1 Microbenchmarks

We evaluate all the queue implementations on a set of microbenchmarks to determine their scalability and performance. The controlled nature of these microbenchmarks allows us to compare particular aspects of each algorithm, such as transactional overhead introduced by STO. All experiments are run on a 100GB DRAM machine with two 6-core Intel Xeon X5690 processors clocked at 3.47GHz. Hyperthreading is enabled in each processor, resulting in 24 available logical cores. The machine runs a 64-bit Linux 3.2.0 operating system, and all benchmarks and STO data structures are compiled with `g++-5.3`. In all tests, threads are pinned to cores, with at most one thread per logical core. In all performance graphs, we show the median of 5 consecutive runs with the minimum and maximum performance results represented as error bars.

Parameters

Value Types. Each queue benchmark uses randomly chosen integers because the benchmarks do not manipulate the push or popped values, and the queue algorithms are agnostic to the actual values being placed in the queue.

Initial Queue Size. We run our tests with varying numbers of initial entries in the queue. This affects how often the structure becomes empty, which can cause aborts and additional overhead (as described in the algorithms above). It also affects the number of cache lines accessed: a near-empty queue will never require iterating over values contained in more than one cache line.

Operations per transaction. We set the number of operations per transaction to 1 (i.e., the transactions are singleton transactions). By keeping a transaction as

short as possible, we maximize the impact of any fixed per-transaction overhead. However, we also minimize the performance hit from the additional, variable transactional overhead required to execute and commit larger transactions: running multiple-operation transactions requires multiple-item support in read and write sets, creates scenarios of read-my-writes, and increases the number of aborts and retries, all of which incur additional overhead. Because most highly-concurrent data structures provide guarantees equivalent to those of singleton transactions, we use singleton transactions when comparing transactional and non-transactional data structures.[2]

Tests

2-Thread Push-Pop Test. This test has one thread that performs only pushes and another thread that performs only pops (a traditional "producer-consumer" model). Each thread performs 10 million transactions. Unless the queue is empty, the two threads should never be modifying the same part of the data structure and will never conflict, leading to an abort rate that should be near 0. We use this test to measure the speed of push/pops on the queue under low or no contention. We expect that our transactional queues should perform just as well as the highly-concurrent queues, if not better: while highly-concurrent, non-transactional algorithms are optimized for multi-threaded access, our simpler implementation should be just as fast with low contention and low abort rates.

Multi-Thread Singletons Test. In this test, a thread randomly selects an operation (push or pop) to perform within each transaction. This keeps the queue at approximately the same size as its initial size during the test. Each thread performs

[2]Our data structure implementations can correctly handle multiple-operation transactions: we simply benchmark them with singleton transactions to compare performance.

10 million transactions. We run this test with different initial queue sizes and different numbers of threads, with each thread performing singleton transactions. Altering the number of threads allows us to benchmark performance under variable amounts of contention. We expect that T-QueueO and T-QueueP will perform significantly worse once the number of threads is increased, and that these naive synchronization algorithms will underperform synchronization algorithms optimized for contentious situations.

3.4.2 Overview of Results

We first present an overview of our conclusions, then explain each conclusion in more detail by proposing a sequence of five hypotheses tested using the benchmarks described above. For each hypothesis, we use our benchmark results to formulate a conclusion that either refutes or supports the hypothesis. We provide a few figures that highlight our results; full results (including abort rates) can be found in Appendix A. We draw the following conclusions from our results (corresponding to our five hypotheses):

1. An implementation of a transactional queue using a pessimistic approach for pop outperforms one using an optimistic approach.
2. Transactional queues using a naive synchronization algorithm can perform reasonably well compared to concurrent, non-transactional queue algorithms.
3. The flat combining technique is a highly effective synchronization technique for concurrent queues, and the flat combining, non-transactional queue outperforms all transactional and concurrent queues.
4. Fixed overhead from bookkeeping STO wrapper calls is negligible.
5. The transactional flat combining queue underperforms all other transactional

STO queues. Transactional flat combining's low performance is caused by the greater number and greater complexity of flat combining calls that are necessary in a transactional setting.

3.4.3 Hypothesis 1

A transactional queue using a pessimistic algorithm for pop outperforms one using an optimistic algorithm for pop. (Supported)

We test this hypothesis by comparing the performance of T-QueueO and T-QueueP. Recall that a thread running on T-QueueP locks the queue immediately when it performs a transactional pop—therefore pessimistically assuming that any other thread accessing the queue will cause a conflict with its pop operation—while a thread running on T-QueueO does not acquire any locks until commit time.

The comparative performance of T-QueueO and T-QueueP (Figure 3.4.1) on the Multi-Thread Singletons Test demonstrates that our pessimistic implementation of pop is more effective than our optimistic one. T-QueueP performs slightly better than T-QueueO. This is likely due to T-QueueP's lower abort rate (1/3 that of T-QueueO). When a pop is performed, T-QueueP locks the queue and prevents any other thread from observing an inconsistent state, whereas T-QueueO does not acquire any locks and allows other threads to observe inconsistent state (i.e., execute a pop of the head that is about to be popped by another thread). Observation of inconsistent state causes aborts at execution and commit time. This result supports the claim that a pessimistic approach to contentious operations such as pop benefits performance.

On the Push-Pop Test, T-QueueP's performance is double that of T-QueueO's

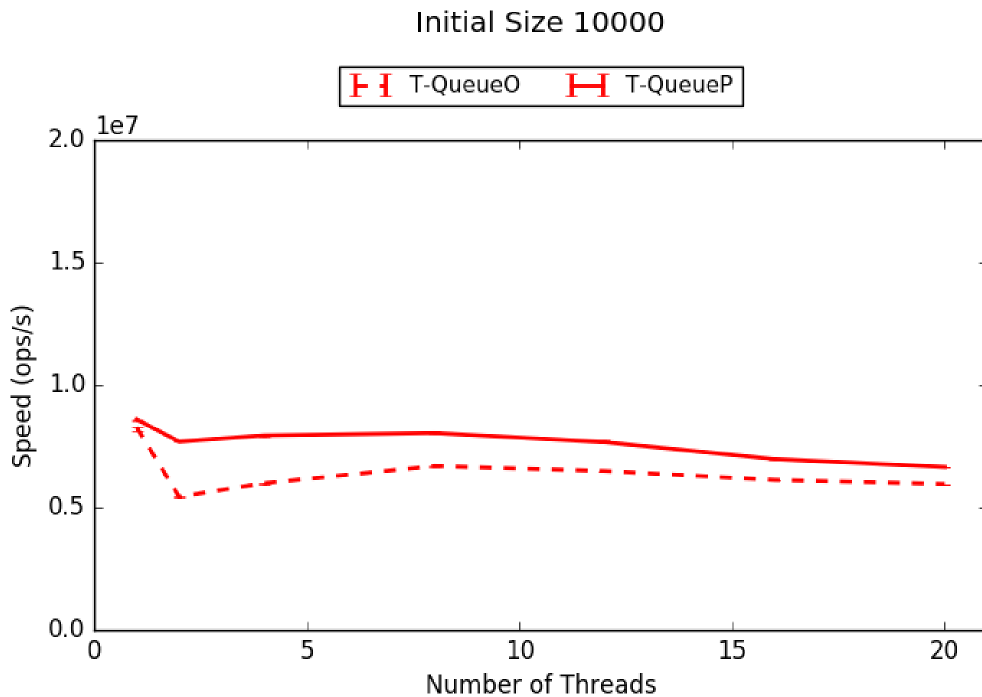


Figure 3.4.1: T-QueueO vs. T-QueueP Performance: Multi-Thread Singletons Test

performance (Figure 3.4.2). This result is initially surprising, because the Push-Pop Test is a low-contention test; the abort rates, as expected, are near 0, with aborts only occurring because the threads spin too long while acquiring locks. We should therefore expect that T-QueueP performs approximately equal to T-QueueO. However, this is clearly not the case.

Our results for the Push-Pop Test can be explained by the different speeds of the push and pop operations in T-QueueO and T-QueueP. Figure 3.4.3 shows that T-QueueP's push-only thread performs 10 million pushes per second, while T-QueueO's push-only thread performs only about 3.1 million pushes per second. T-QueueP's pop-only thread, *when simultaneously running with the push-only thread*, performs about 0.2 million pops per second, while T-QueueO's pop-only

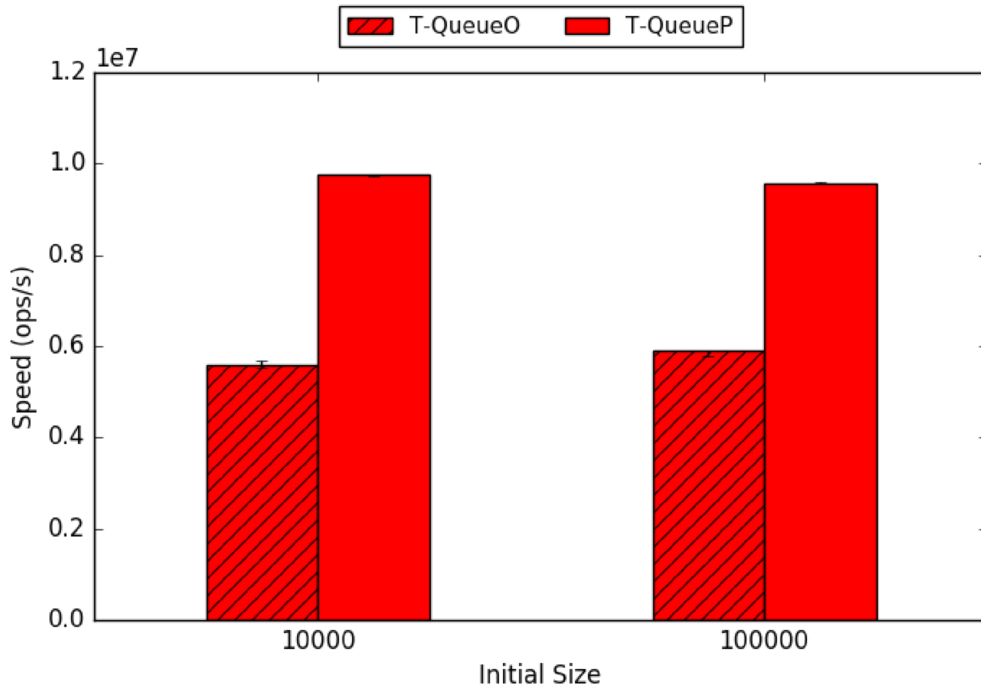


Figure 3.4.2: T-QueueO vs. T-QueueP Performance: Push-Pop Test (2 threads)

thread, when running simultaneously with the push-only thread, performs about 2 million pops per second (achieving speed 10× that of T-QueueP's pop-only thread). Table 3.4.1 summarizes this result in terms of the ratio of pops to pushes completed by each queue: when both the push-only and pop-only threads are simultaneously running, T-QueueP's pop-only thread performs only 28 pops for every 100 pushes the push-only thread performs, while T-QueueO's pop-only thread performs 64 pops for every 100 pushes the push-only thread performs. Once the push-only thread exits, however, both queues' pop-only threads perform pops at a speed of about 10 million pops per second.

The increased speed of a pop once the push-only thread has exited can be explained by the decrease in contention when only one thread is executing. The

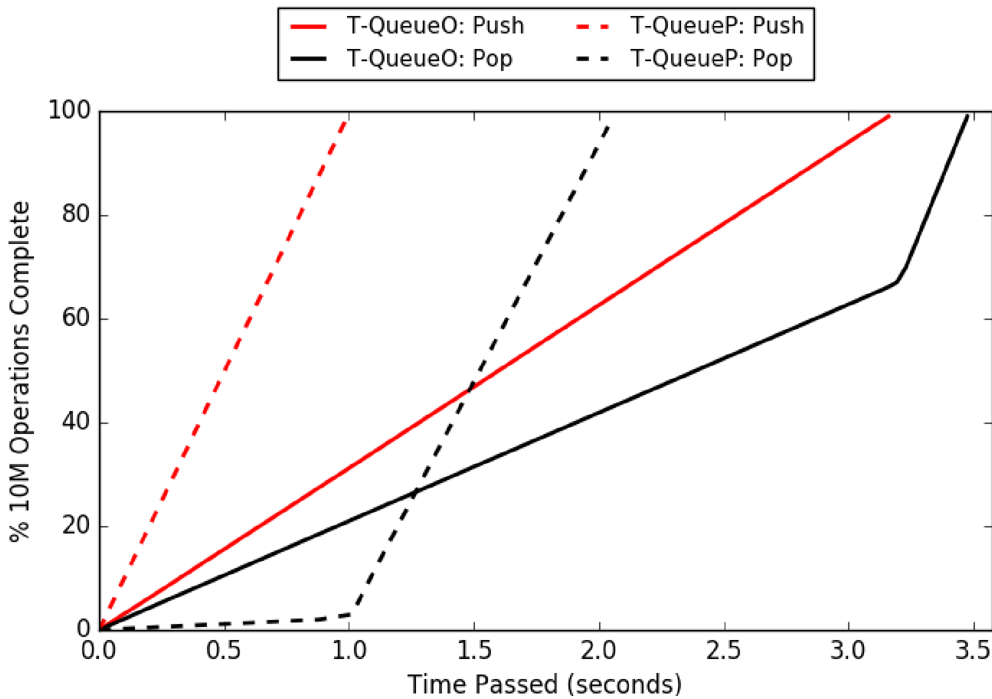


Figure 3.4.3: T-QueueO vs. T-QueueP Push-Pop Test: Speed at which the push- and pop-only threads complete 10 million transactions

Queue	Pops per 100 Pushes
T-QueueO	64
T-QueueP	28

Table 3.4.1: T-QueueO vs. T-QueueP Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously

speed of a pop when the push-only thread is no longer executing is equal to the speed of a single-threaded pop execution. Single-thread execution on the queue results in better cache line performance (there can never be cache line bounces, because only one thread accesses the queue); furthermore, there is zero contention on the queue locks.

T-QueueP's pushes execute significantly faster than do T-QueueO's, and the pushes also appear to be preventing T-QueueP's pops from executing quickly. This is because T-QueueP uses only one queue version as a global queue lock. Both a

push and pop contend on this lock in order to install an operation; the push-only thread can potentially starve the pop-only thread by continuously succeeding in acquiring the lock. Because of this starvation property, the push-only thread and pop-only thread in T-QueueP run in a near-sequential fashion, with the push-only thread running first, and the pop-only thread running second. Sequential execution, as discussed earlier, achieves better performance because of the lack of contention. T-QueueP therefore has greater performance because its threads run in nearly a sequential fashion.[3]

T-QueueO, on the other hand, uses two versions (head and tail versions) as locks, and each thread acquires only one of these locks when performing an operation. This allows both threads to execute operations at a more equal rate, since one cannot starve the other. However, because of the added contention on the cache lines containing the head and the tail of the queue, both the push-only and pop-only threads execute at rates far slower than sequential execution.

We conclude that the unexpected results of the Push-Pop Test are caused by the lock shared by T-QueueP's push and pop operations. This leads to a near-sequential execution of 10 million pops followed by 10 million pushes.

While the improved performance of T-QueueP on the Push-Pop Test is likely caused by our choice of how to implement the pessimistic pop, our results on both tests nevertheless support our hypothesis. They indicate that our pessimistic approach does reduce the abort rate under high contention and lead to slightly better performance. However, the better performance of T-QueueP comes with the caveat that a thread attempting to perform pop operations may experience

[3]We note that T-QueueP's execution pattern also has the fortunate side effect of decreasing the probability that the queue becomes empty, since a greater number of pushes complete for every pop operation that completes. However, starvation of pops can lead to issues in certain scenarios, such as ones that require the queue to be kept at a relatively constant size.

starvation if another thread is performing push operations.

SUPPORTED: T-QueueP outperforms T-QueueO, indicating that a pessimistic approach for pop can outperform an optimistic approach.

3.4.4 Hypothesis 2

Under low contention, a transactional queue with a naive concurrent algorithm performs reasonably well compared to the best concurrent, non-transactional queue algorithms. (Supported)

We benchmark a set of the best-performing highly-concurrent queue algorithms against our transactional queue implementations, T-QueueO and T-QueueP, using our low-contention test (the 2-Thread Push-Pop Test) that is also optimized for a low abort rate. This acts as a best-case scenario for T-QueueO and T-QueueP algorithms. Selected results are shown in Figure 3.4.4.

Our implementation of the non-transactional flat combining queue, which we call NT-FCQueue, uses the flat combining queue implemented by the authors of the flat combining paper [17] (with minor modifications to remove memory leaks). Our implementations of the other concurrent queues are taken from the open source Concurrent Data Structures (CDS) library implementations [25].

All concurrent, non-transactional queues achieve approximately equal performance on the Push-Pop Test besides NT-FCQueue, Segmented Queue [1], and TsigasCycle Queue [39]. T-QueueP outperforms all queues by at least 150% on the 2-thread Push-Pop Test. This is, as we discussed earlier, likely caused by T-QueueP's near-sequential execution on this test. We see in Table 3.4.2 that most queues perform more than 1 pop for every 2 pushes while the push-only thread is running. We also observe that NT-FCQueue is the only queue in which 10 million

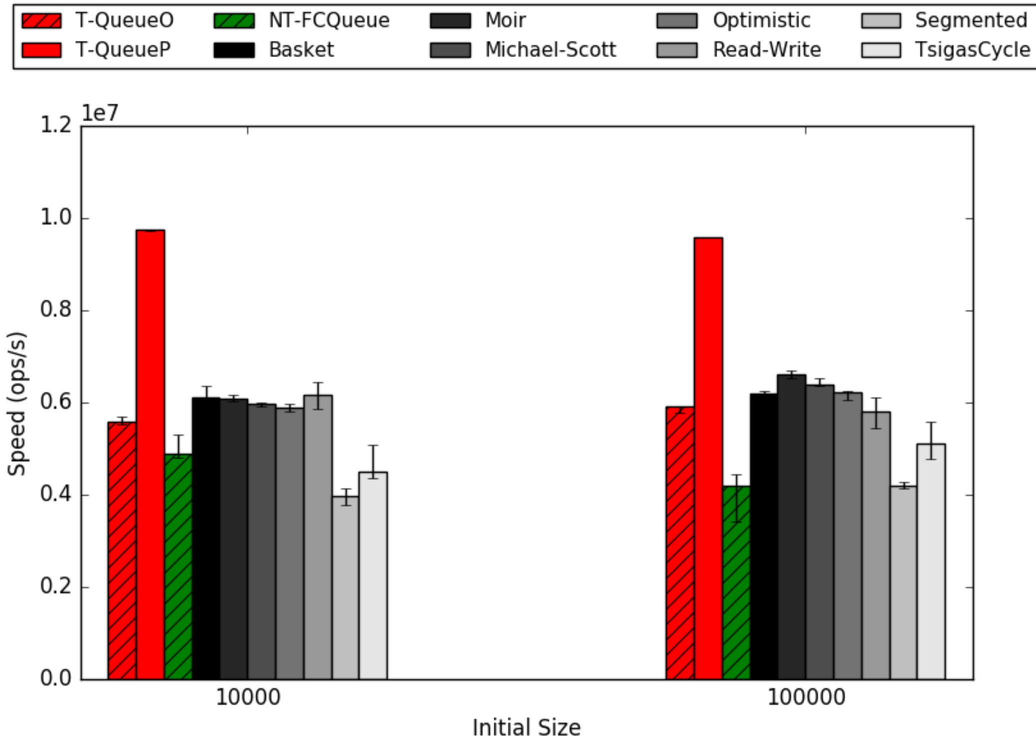


Figure 3.4.4: Non-transactional, Concurrent Queue vs. Transactional Queue Performance: Push-Pop Test (2 threads)

pops complete faster than 10 million pushes, and has nearly a 1:1 ratio of pops to pushes. This may explain its poor performance on the Push-Pop Test as compared to the other queues (fewer pop operations are executed in a single-threaded manner after the push-only thread has exited).

Regardless of the difference in speed between the push-only and pop-only threads, we see that our naive algorithms perform at least as well as the majority of concurrent algorithms on this test. T-QueueO performs about equally as well as any non-transactional queue, and the ratio of pops to pushes in T-QueueO is close to that of these non-transactional queues.

The Push-Pop Test is designed for low abort rates and minimal transactional overhead from tracking items in read and write sets. It is therefore unsurprising

Queue	Pops per 100 Pushes
T-QueueO	64
T-QueueP	28
NT-FCQueue	110
Basket	42
Moir	62
Michael-Scott	54
Optimistic	50
Read-Write	84
Segmented	50
TsigasCycle	52

Table 3.4.2: Non-transactional Queues vs. T-QueueO and T-QueueP Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously

that, on this test, a simple synchronization strategy can outperform the majority of highly-concurrent algorithms which are optimized for scalability. Our results demonstrate that a simple implementation of a naive algorithm can consistently outperform more complex concurrent queue implementations, even when supporting transactions using STO. The overhead added from STO does not inherently cripple performance—our transactional data structures can compete with several highly-concurrent, non-transactional data structures in particular cases.

SUPPORTED: T-QueueP and T-QueueO outperform or match the performance of all concurrent, non-transactional queues on the 2-Thread Push-Pop Test. This indicates that a simple concurrent algorithm in a transactional setting can perform well under optimal circumstances.

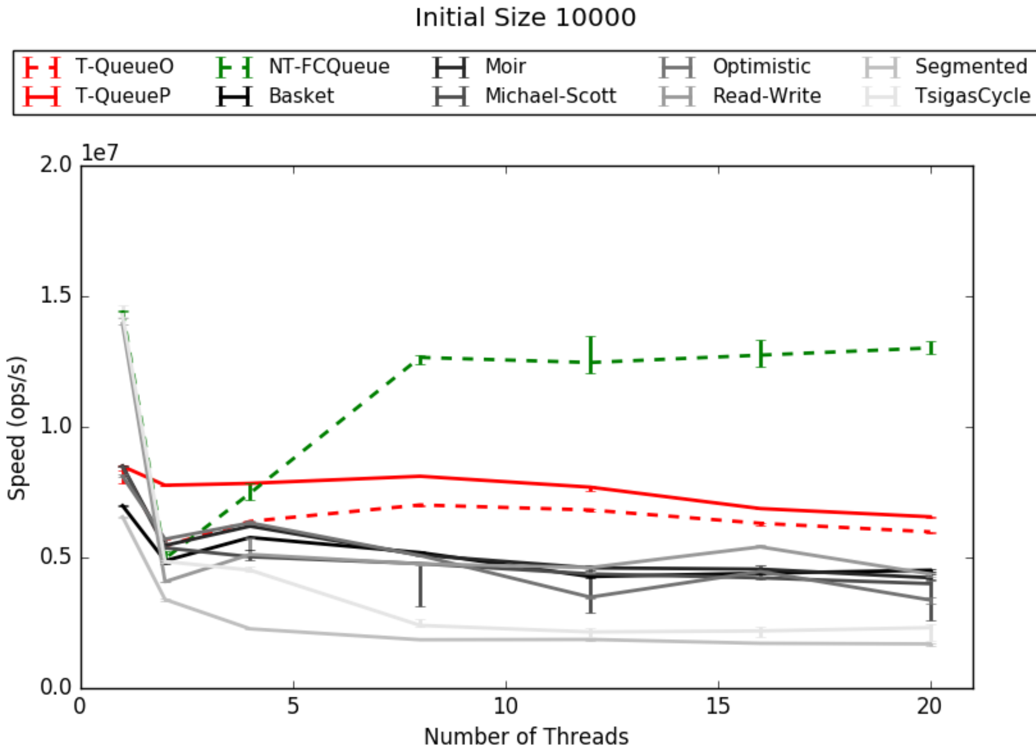


Figure 3.4.5: Non-transactional, Concurrent Queue vs. Transactional Queue Performance: Multi-Thread Singletons Test

3.4.5 Hypothesis 3

The flat combining algorithm is the most promising concurrent queue algorithm to integrate with STO. (Supported)

We run the same set of concurrent queues from the previous hypothesis on the Multi-Thread Singletons Test, which provides a more realistic example of high-contention workloads that a queue may experience. We investigate how different concurrent, non-transactional algorithms perform in high-contention situations compared to the transactional T-QueueP, and look for the most scalable and performant concurrenct queue that outperforms T-QueueP to integrate with STO. Selected results are shown in Figure 3.4.5.

As the number of threads increases above 2, NT-FCQueue achieves

performance over 2.5× greater than any other concurrent, non-transactional queue. The Multi-Thread Singletons Test highlights the performance benefits of the flat combining queue: as contention increases, the flat combining queue reaches performance approximately double that of T-QueueP. In addition, the flat combining queue is the only queue that scales. All the other concurrent algorithms perform worse than T-QueueP, regardless of the number of threads accessing the queue or the initial queue size. Although an increase in the duration of a transaction and number of operations per transaction would cause T-QueueP to perform far worse than other concurrent queues, our results demonstrate that a simple synchronization algorithm can achieve equal performance to more complex synchronization algorithms even in some highly contentious scenarios.[4]

A comparison with NT-FCQueue indicates that the simple synchronization algorithms used by T-QueueO and T-QueueP are certainly not optimal for performance in a non-transactional setting. Given these results, as well as the algorithmic benefits of the flat combining technique described in Section 3.3.1, we choose the flat combining queue to integrate with STO.

SUPPORTED: NT-FCQueue significantly outperforms all concurrent, non-transactional queues *and* T-QueueP on the Multi-Thread Singletons Test, indicating that flat combining may be the most promising algorithm to integrate with STO to create a more performant and scalable transactional queue.

[4]We rely on the specific libcds [25] implementation of these concurrent, non-transactional data structures, which may not be the most optimized versions of these data structures. However, the performance of these implementations on our tests matches their performance in other research evaluating these data structures [29, 31].

```
Sto::start_transaction();
try {
    do_queue_op(push, 1);
    do_queue_op(pop, NULL);
    if (Sto::try_commit()) {
        printf("committed");
    }
} catch (Transaction::Abort e) {
    printf("aborted");
}
```

Figure 3.4.6: Example usage of STO wrapper calls

3.4.6 Hypothesis 4

Overhead from general STO bookkeeping does not cripple performance of a highly-concurrent queue algorithm. (Supported)

We create a version of NT-FCQueue, called NT-FCQueueWrapped, that invokes general STO bookkeeping calls. The relative performance of NT-FCQueueWrapped to NT-FCQueue indicates how much of the overhead added by the STO system is unavoidable (without modifying STO itself).

The STO wrapper functions called by NT-FCQueueWrapped must be called by any user of the data structure in order for the data structure to perform the necessary bookkeeping to support transactions. These two calls are `start_transaction` and `try_commit`, and allow a user to mark which operations should occur together in the same transaction. An example of how these calls are used is shown in Figure 3.4.6. After invoking the `start_transaction` call, the thread is able to track items in its read or write sets. At the end of a transaction, the thread invokes the `try_commit` call to run the commit protocol. NT-FCQueueWrapped adds no items to the read or write sets after invoking

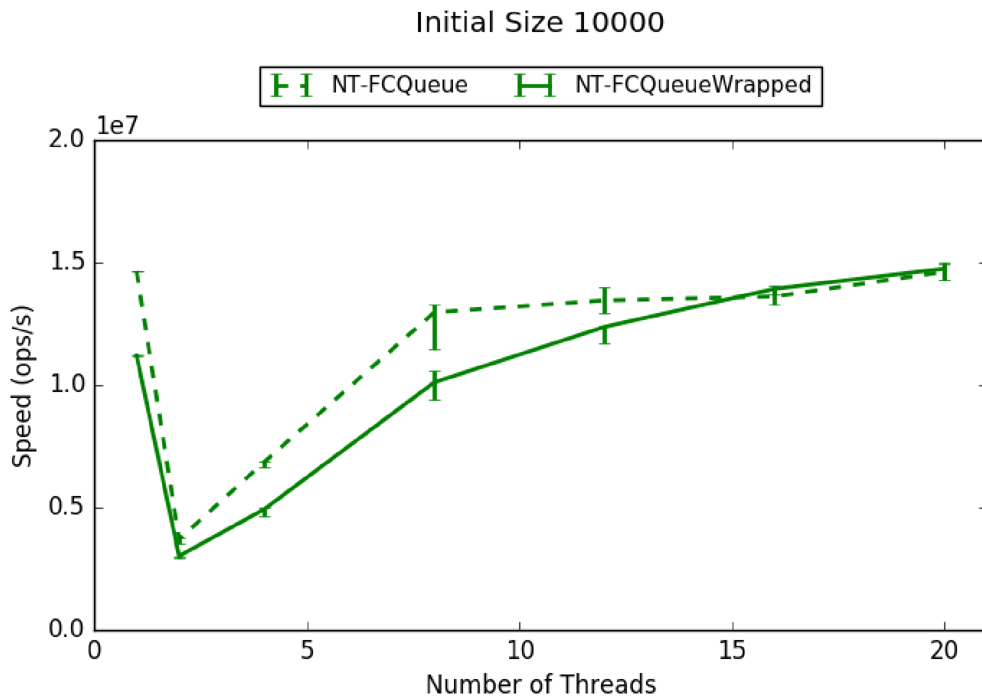


Figure 3.4.7: NT-FCQueue vs. NT-FCQueueWrapped Performance: Multi-Thread Singletons Test

`start_transaction` and does nothing in its commit protocol. This means that NT-FCQueueWrapped incurs the minimum amount of overhead necessary to use STO, and therefore represents the upper bound on the performance we can expect from a fully transactional flat combining queue (T-FCQueue).

We see from our Multi-Thread Singletons Test results (Figure 3.4.7) that the STO wrapper calls can lead to a performance loss ranging from 0% at 20 threads to 30% at 4 threads compared to the performance of NT-FCQueue. With fewer threads accessing the queue, the proportion of overhead from the STO wrapper calls is greater, because the overhead from synchronizing access to the queue is minimal. As the number of threads increases, the overhead from STO wrapper calls becomes negligible in comparison to the cost of synchronization. NT-FCQueueWrapped

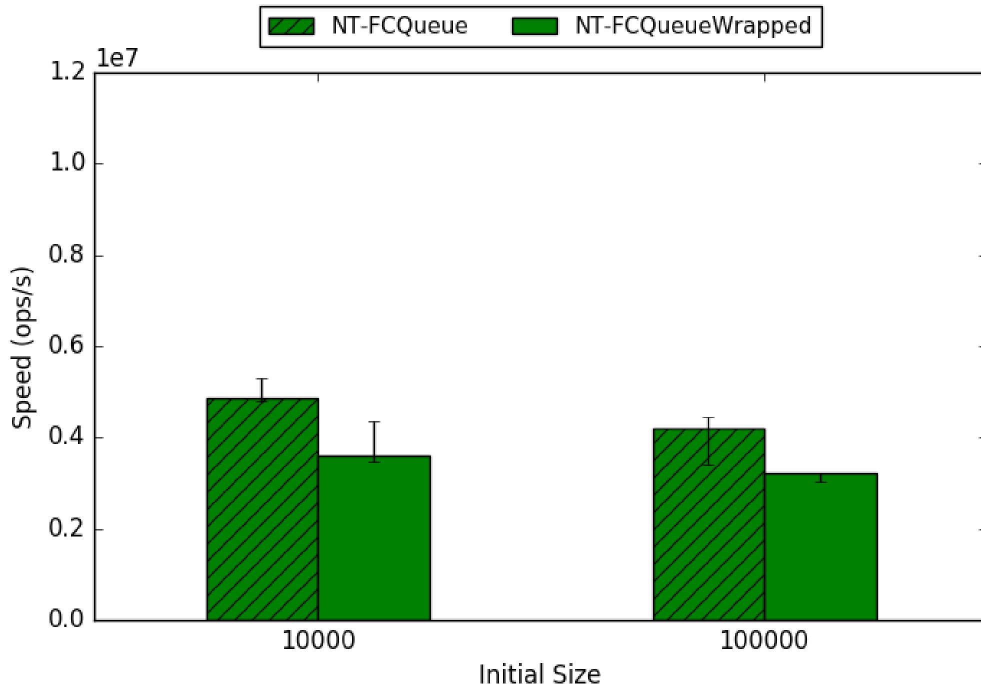


Figure 3.4.8: NT-FCQueue vs. NT-FCQueueWrapped Performance: Push-Pop Test (2 threads)

retains most of NT-FCQueue's scalability, and the two queues perform equally well after the number of threads reaches approximately 14.

We also see an impact on performance of NT-FCQueue in the Push-Pop Test, shown in Figure 3.4.8: performance drops by approximately 20%. The push-only thread now beats the pop-only thread in NT-FCQueueWrapped, but the ratio of 94 pops per 100 pushes is still close to 1:1. The performance impact likely comes from the overhead added by STO wrapper calls; this is expected, as we discussed before, because this constant overhead affects performance more significantly at low thread counts and low contention.

The comparison of NT-FCQueueWrapped and NT-FCQueue demonstrates that the unavoidable overhead of STO becomes negligible at high thread counts. Even

Queue	Pops per 100 Pushes
T-QueueP	28
NT-FCQueueWrapped	94
T-FCQueue	57

Table 3.4.3: T-FCQueue Push-Pop Test: Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously

with the wrapper calls, our results indicate it can still be possible to achieve performance at 20 threads up to nearly 2× greater than that of T-QueueP (the second-best performing queue).

SUPPORTED: Invoking the general STO wrapper functions that are necessary for any data structure to support transactions does not cripple the performance of highly-concurrent queues such as NT-FCQueue, particularly at high thread counts.

3.4.7 Hypothesis 5

A transactional flat combining queue outperforms and scales better than a transactional queue with a naive concurrent algorithm. (Not Supported)

We compare T-FCQueue against NT-FCQueueWrapped and T-QueueP to measure how the flat combining transactional approach described in Section 3.3.2 performs.

In the Push-Pop Test (Figure 3.4.9), T-QueueP outperforms both flat combining variants. This is an unsurprising result given our results from the concurrent queues benchmark in Figure 3.4.4, and the fact that T-FCQueue has a more equal ratio of pops to pushes than does T-QueueP (Table 3.4.3).

The Multi-Thread Singletons Test (Figure 3.4.10) shows that T-QueueP performs approximately 2× better than T-FCQueue, regardless of initial queue size. Both queues do not scale, and the performance ratio remains constant regardless of

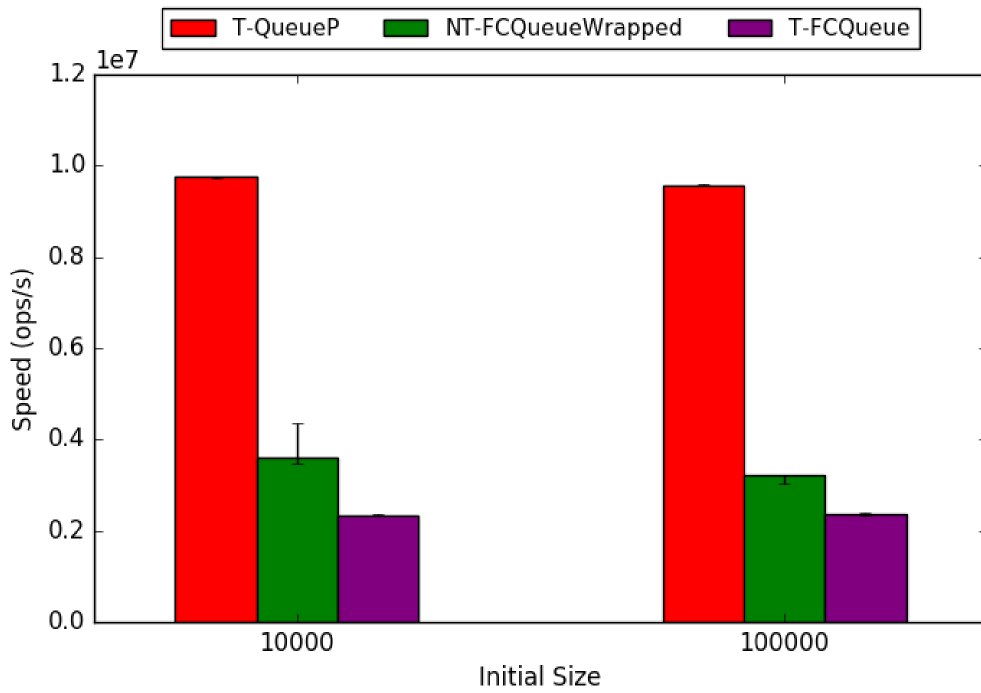


Figure 3.4.9: T-FCQueue Performance: Push-Pop Test (2 threads)

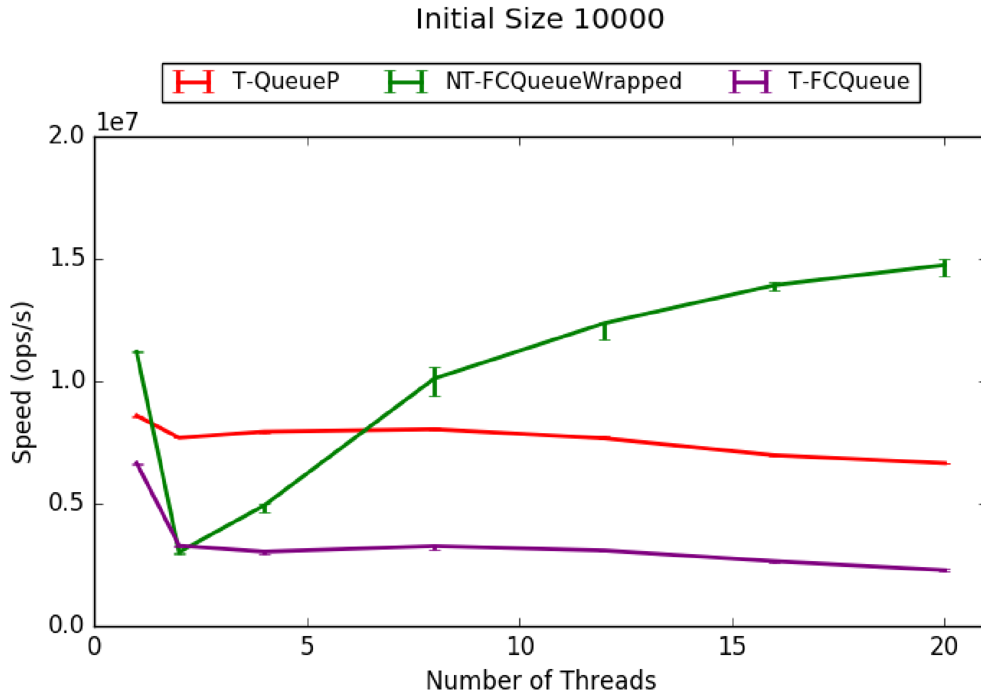


Figure 3.4.10: T-FCQueue Performance: Multi-Thread Singletons Test

the number of threads. T-FCQueue also experiences abort rates around 5%, which are 1.5–2× the abort rates of T-QueueP.

NOT SUPPORTED: The transactional flat combining queue does not outperform or scale better than other transactional queues using naive synchronization mechanisms. T-FCQueue's poor performance compared to that of T-QueueP demonstrates that the flat combining algorithm performs poorly when modified to support transactions.

3.4.8 Conclusion

Analysis with the `perf` tool indicates that the majority of T-FCQueue's overhead comes from spinning on the flat combining lock (acquired by the combiner thread), or waiting for a flat combining call to complete. In addition, the number of cache misses is over 4× greater than that of NT-FCQueue (see Appendix A.1). This overhead occurs for two reasons:

1. *Higher Quantity*: As described in Section 3.3.2, a thread must make multiple flat combining calls to perform a pop within a transaction (recall that a push only requires one flat combining call).
2. *Higher Complexity*: Existing flat combining calls need to be made more complex to support transactions (for example, installing a list of values per push rather than a single value).

We conclude that the flat combining technique, while perhaps near-optimal for a concurrent, non-transactional queue, is no better in a transactional setting than a naive synchronization technique such as that used in T-QueueO and T-QueueP. The flat combining algorithm must track the state of the queue during a transaction's lifetime to provide transactional guarantees (e.g., marking values in the queue, or observing that the queue was empty when performing a pop). To do so requires

both adding new flat combining calls and increasing the complexity of existing ones; these modifications cripple flat combining's performance. In the next chapter, we formalize this argument using commutativity, and claim that the flat combining technique fundamentally depends on operation commutativity that is present in only a non-transactional setting in order to achieve its high performance.

4

Commutativity and Scalability of Queue Specifications

This chapter describes the commutativity of our queue operations in both a non-transactional setting and a transactional setting, and relates the amount of queue operation commutativity to queue implementation performance. For clarity, we refer to the queue operation interface shown in Figure 3.2.1 as the *strong queue specification*; a transactional queue with this interface is the *strong transactional queue*. We hypothesize that the strong queue specification cannot be implemented in

a transactional setting in an efficient way due to the lack of operation commutativity in the strong queue specification. We follow this by proposing an alternative queue specification—the *weak queue specification*—that allows for greater operation commutativity, and hypothesize that this alternative specification will allow for greater transactional queue scalability.

As a supporting example of our hypotheses, we examine the flat combining technique in detail, and argue that the flat combining technique cannot implement the strong queue interface efficiently in a transactional setting. While the flat-combining technique is perhaps near-optimal for a concurrent, non-transactional queue, it performs no better than a naive synchronization technique in a transactional queue. This is because the flat combining algorithm's high performance comes from exploiting the greater operation commutativity present in a non-transactional setting. The flat combining algorithm's optimizations must be heavily modified in order to support transactions, which leads to significant performance loss.

We then implement a weak transactional flat combining queue—a flat combining queue with operations satisfying the weak queue specification—with the expectation that the flat combining technique can achieve scalable performance close to its performance in a non-transactional setting. Our experimental results illustrate that the greater commutativity of operations in the weak queue specification is essential for the flat combining technique to be effective in a transactional setting.

4.1 Histories

We introduce some terminology about histories and transactional histories that will be used in our discussion of operation commutativity.

Definition 1. A *history* is a sequence of `(thread, operation, result)` tuples that represent an interleaving of operations of all threads. Knowledge of both the history and initial conditions of a data structure leads to complete knowledge of the (high-level) end state of the structure.

Example 1.

```
// Q.size() == 0
(T2, Q.push(a), ())
(T1, Q.pop(), true)
(T2, Q.push(a), ())
(T1, Q.pop(), true)
// Final State: Q.size() == 0
```

Definition 2. A *transactional history* is a specific type of history in which the tuples represent an interleaving of operations of the threads' committed transactions. A transactional history includes `(thread, START_TXN, ())` and `(thread, COMMIT_TXN, commit_result)` operation tuples that represent the time the thread starts and commits the transaction. `commit_result` represents the observable effects of the installation procedure at commit time.

Example 2.

```
// Q.size() == 0
(T1, START_TXN, ())
(T2, START_TXN, ())
(T2, Q.push(a), ())
(T1, Q.pop(), true)
(T2, Q.push(a), ())
(T1, Q.pop(), true)
(T1, COMMIT_TXN, ())
(T2, COMMIT_TXN, ())
// Final State: Q.size() == 0
```

Definition 3. A history H' is *consistent* with H if:

1. H' contains the same tuples as H: the same operations were executed with the same return values for all operations within the transactions.

2. The order of a single thread's calls in H' remains consistent with the thread's order of calls in H.

Definition 4. A transactional history H is *serial* if all tuples are grouped by transaction: if $i \leq j \leq k$ and H_i and H_k are from the same transaction, then H_j is also from that transaction. This means the tuples form a serial transaction order.

Definition 5. A transactional history H is *serializable* if there exists a serial history H' s.t. H' is consistent with H.

Example 3. H is a serializable transactional history whose corresponding serial execution is H'. H'' represents a serial transactional history, but is inconsistent with H because its pop operations return different results.

H	H'	H''
// Q.size() == 0	// Q.size() == 0	// Q.size() == 0
(T1, START_TXN, ())	(T2, START_TXN)	(T1, START_TXN)
(T2, START_TXN, ())	(T2, Q.push(a), ())	(T1, Q.pop(), false)
(T2, Q.push(a), ())	(T2, Q.push(a), ())	(T1, Q.pop(), false)
(T1, Q.pop(), true)	(T2, COMMIT_TXN)	(T1, COMMIT_TXN)
(T2, Q.push(a), ())	(T1, START_TXN)	(T2, START_TXN)
(T1, Q.pop(), true)	(T1, Q.pop(), true)	(T2, Q.push(a), ())
(T1, COMMIT_TXN, ())	(T1, Q.pop(), true)	(T2, Q.push(a), ())
(T2, COMMIT_TXN, ())	(T1, COMMIT_TXN)	(T2, COMMIT_TXN)

Definition 6. A transactional history is *linearizable* if all transactions appears to occur instantaneously between their start time and their commit time: if transaction $T1$ commits before transaction $T2$ begins, then $T1$ must appear before $T2$ in the serial history [14].

Definition 7. A transactional history H is *strictly serializable*, or *valid*, if it is both serializable and linearizable. Any data structure implemented in a transactional setting requires strictly serializable transactional histories.

Example 4. H is a serializable, but not linearizable, transactional history. This is because $T2$ should have observed the pushes committed by $T1$. We can find a serial ordering of H, shown in H', but H' violates the rule that the serial order of transactions corresponds to the real time order of the transactions' commits.

H	H'
`// Q empty`	`// Q empty`
`(T1, START_TXN)`	`(T2, START_TXN)`
`(T1, Q.push(a), ())`	`(T2, Q.pop(), false)`
`(T1, Q.push(a), ())`	`(T2, COMMIT_TXN)`
`(T1, Q.pop(), true)`	`(T1, START_TXN)`
`(T1, COMMIT_TXN)`	`(T1, Q.push(a), ())`
`(T2, START_TXN)`	`(T1, Q.push(a), ())`
`(T2, Q.pop(), false)`	`(T1, Q.pop(), true)`
`(T2, COMMIT_TXN)`	`(T1, COMMIT_TXN)`

4.2 The Scalable Commutativity Rule

The *scalable commutativity rule*, formally defined by Clements et al. [6], asserts that whenever interface operations *commute*, there exists an implementation of the interface that scales. Operations *commute* in a particular interface when there is no way to distinguish their execution order: exchanging the order of the operations in the history does not modify the return values of the operations seen by each thread, and no possible future sequence of operations can distinguish the two orders. In the following discussion, we use the scalable commutativity rule to argue for the presence (or lack) of scalable implementations of operation interfaces in non-transactional and transactional settings.

4.3 Commutativity of the Strong Queue Specification

In a non-transactional setting, we consider histories in which the only operations are push and pop (i.e., the histories are not transactional). Given the strong queue specification (Figure 3.2.1), in which push returns `void` and pop returns `bool`, we determine the commutativity of these operations by examining the effects of exchanging the order in which the operations appear in the history. Two operations do not commute if exchanging the operation's order changes either (a) the operations' return values, or (b) the resulting global state (if the resulting state is the same in both orders, then no future operation sequences can distinguish the orders). We show operations that do not commute in Table 4.3.1.

Based on this commutativity analysis, we note that a pop operation does not commute with a push operation when the queue is empty, and it does not commute with another pop operation when the queue is near empty. By the scalable commutativity rule, this means that there is no concurrent queue implementation for pop that scales in these particular scenarios. A push operation commutes with all operations because it returns `void`, and has a scalable implementation in all scenarios.

In practice, the pop operation is rarely used without an accompanying call to the front operation immediately prior to the pop. This is because a user of the queue will not only want to know if removing a value from the queue succeeded, but also want to know the contents of the popped value. If every pop is immediately preceded by a front operation, we note that a front-pop operation pair will never commute in *any* scenario, even when the queue is empty. This is because a front-pop will return the contents of the value at the head of the queue, and the same value

Operations	H	H'

push vs. pop	`// Q empty`
	`(T, Q.push(a), ())`
	`(T, Q.pop(), true)`
		`// Q empty`
		`(T, Q.pop(), false)`
		`(T, Q.push(a), ())`

pop vs. pop	`// Q.size() = 1`
	`(T1, Q.pop(), true)`
	`(T2, Q.pop(), false)`
		`// Q.size() = 1`
		`(T2, Q.pop(), true)`
		`(T1, Q.pop(), false)`

Table 4.3.1: Strong queue operations that do not commute. H is the original history; H' is the history with the order of the two operations exchanged. In the push vs pop scenario, note that any thread may have performed the operations, and the operations will still not commute.

cannot be popped off the queue twice. To simplify our commutativity and queue algorithms discussions, we omit the front operation; however, it is important to note that omitting the front operation has the side effect of deceiving us into believing that a pop commutes in more scenarios that it does in practice.

We reason about commutativity of a queue implemented in a transactional setting using transactional histories, which include `START_TXN` and `COMMIT_TXN` operations. A transactional setting calls for strict serializability of the transactional history, which by definition entails serializability (i.e., that tuples in histories be grouped by transaction). This adds an additional level of commutativity, namely commutativity between transactions.

Because a valid transactional history is strictly serializable, we can find a corresponding serial history for every valid transactional history. This means that operations belonging to the same transaction must occur in a group in the history. To reason about transaction commutativity, we have a parallel notion to exchanging operations in the history: we exchange *groups* of operations in a `START_TXN` and `COMMIT_TXN` block of a serial history, and detect whether there is any observable change in either the return values within the exchanged transactions, or the

resulting global state. We note that the occurrence of each transaction in the serial transactional history can be uniquely identified by its `COMMIT_TXN` tuple. Thus, if exchanging the positions of two transactions does not commute in a particular scenario, we can say that the two `COMMIT_TXN` operations do not commute in that scenario.

In a transactional setting, we determine the commutativity of the queue specification by considering both individual operation commutativity and transaction commutativity. Note that when transactions contain only one operation, then a lack of commutativity between two transactions is equivalent to saying that that the two operations do not commute (and vice versa). In general, two transactions—i.e., two `COMMIT_TXN` operations—do not commute if (a) a pop operation in one transaction observes an empty queue given one ordering of the transactions, but not the other; and/or (b) the resulting size of the queue immediately after both transactions have committed changes when the order of the transactions changes. We provide some examples of larger transactions that fail to commute in Table 4.3.2.

Based on this commutativity analysis, we note that, in addition to the non-commutativity of pop operations in particular scenarios involving empty, or near-empty queues, we now have a further lack of commutativity of transactions (identified by their `COMMIT_TXN` operations), even in scenarios in which the queue may contain far more than one value. For example, even the queue is relatively full, a large transaction performing several pops may reduce the queue to a near-empty state, and one of its pop operations may observe the empty status of the queue. This transaction will then *not* commute with any other transaction performing a push or pop. By the scalable commutativity rule, this means that there is no

Example	H	H'
1.	// Q empty (T1, START_TXN, ()) (T1, Q.pop(), false) (T1, Q.push(a), ()) (T1, COMMIT_TXN, ()) (T2, START_TXN, ()) (T2, Q.pop(), true) (T2, COMMIT_TXN, ())	// Q empty (T2, START_TXN, ()) (T2, Q.pop(), false) (T2, COMMIT_TXN, ()) (T1, START_TXN, ()) (T1, Q.pop(), false) (T1, Q.push(a), ()) (T1, COMMIT_TXN, ())
2.	// Q empty (T1, START_TXN, ()) (T1, Q.push(a), ()) (T1, Q.pop(), true) (T1, Q.push(a), ()) (T1, COMMIT_TXN, ()) (T2, START_TXN, ()) (T2, Q.pop(), (true)) (T2, Q.push(a), ()) (T2, COMMIT_TXN, ())	// Q empty (T2, START_TXN, ()) (T2, Q.pop(), (false)) (T2, Q.push(a), ()) (T2, COMMIT_TXN, ()) (T1, START_TXN, ()) (T1, Q.push(a), ()) (T1, Q.pop(), true) (T1, Q.push(a), ()) (T1, COMMIT_TXN, ())

Table 4.3.2: Examples of strong queue transactions that do not commute. For clarity, we show only the serial history corresponding to the original valid transactional history. H is the original serial history; H' is the history with the order of the two transactions exchanged.

scalable queue implementation for a strong transactional queue whenever `COMMIT_TXN` operations do not commute.

We hypothesize that any transactional implementation of our strong queue specification that must handle scenarios in which queues may become empty will not scale. A concurrent queue already lacks a scalable implementation for pop due to the high likelihood that a pop operation will not commute with any other operation when the queue nears empty. When this queue is put in a transactional setting, there is an even higher likelihood that two transactions will not commute, even if the queue contains more values.[1] This prevents an efficient, scalable

[1] Furthermore, a front-pop will never commute with another operation, even in a non-empty queue; every transaction containing a front-pop will not commute with another transaction containing a front-pop.

implementation of a strong transactional queue. In Section 4.4, we provide a counterexample, a transactional queue satisfying the *weak queue specification*, and demonstrate how increased commutativity in this new specification allows for a scalable implementation.

4.3.1 The Strong Transactional Flat Combining Queue

The flat combining algorithm is an example of a queue algorithm that implements the strong queue interface and loses its effectiveness in a transactional setting. Recall that our results from testing Hypothesis 5 demonstrate that flat combining's effectiveness is lost in a transactional setting. Here, we argue that this is due to a decrease in the amount of commutativity in the transactional setting: in addition to non-commutative single operations (pops and pushes), transactions may not always commute.

Flat combining's fundamental principle is that requests posted to the publication list can be blindly applied to the queue in an arbitrary order. It handles non-commutativity on the level of individual pop operations by synchronizing concurrent access to the queue with the combiner thread: only one thread at a time is allowed to perform an operation on the queue.[2] In a strong transactional queue, the flat combining algorithm must also correctly handle transactions that may not commute, which means synchronizing the `COMMIT_TXN` operations of different transactions. This requires that the algorithm prevent operations within transactions from interleaving in the transactional history in ways that prevent the transactional history from being serialized. In other words, the order in which

[2] We note that flat combining is not a scalable implementation of pop (or push): the combiner thread's application of requests can only be as efficient as a sequential execution of all thread operations. As our results from testing Hypothesis 3 show, flat combining is nonetheless the most efficient of all concurrent queue algorithms evaluated.

	Interleaving
1.	(T1, Q.pop(), true/false) (T2, Q.pop(), true/false) (T1, Q.pop(), true/false)

2.	// Q.size() == 1
	(T1, Q.pop(), true) // Q empty
	(T2, Q.push(a), ())
	(T1, Q.pop(), true)

3.	// Q.size() == 1
	(T1, Q.pop(), true) // Q empty
	(T2, Q.pop(), false)
	(T1, Q.push(a), ())

4.	(T1, Q.push(a), ()) (T2, Q.push(a), ()) (T1, Q.push(a), ())

5.	// Q.size() == 0
	(T1, Q.push(a), ())
	(T2, Q.pop(), true) // Q empty
	(T1, Q.pop(), false)

Table 4.3.3: Invalid operation interleavings in transactional histories.

operations are applied becomes important.

We show all invalid interleavings of operations that will violate transactional guarantees (i.e., cause histories to be non-strictly serializable) in Table 4.3.3.[3] We describe several methods to prevent these interleavings, and argue that these methods cannot be integrated with the flat combining algorithm without introducing overhead that reduces its performance to below that of T-QueueO or T-QueueP. Each of these methods adds complexity to the push and pop flat combining calls, and creates new flat combining calls to check, undo, or install at commit time.

A standard method for a transactional queue push is to delay the push

[3]We derive these interleavings using Schwarz's method [35]. Schwarz reasons about invalid histories using *dependencies*. All operations performed by a transaction can be thought of in terms of reads and writes, and these operations create read-write, write-write, etc. dependency edges between two transactions. Schwarz asserts that invalid histories must necessarily include cycles in the dependency graph consisting of some number of read-write, write-read, or write-write edges.

execution until commit time. This is allowable because a push observes none of the queue state, and therefore has no effect at execution time. Delaying all pushes until commit time prevents interleavings 4 and 5. These interleavings can occur only if $T1$'s first push is visible to $T2$ prior to $T1$'s commit. If we delay pushes until commit time, $T2$ will not detect the presence of a pushed value in the queue.

Because pop operations immediately return values that depend on the state of the queue (`false` if the queue is empty, or `true` if the queue is nonempty), interleavings 1, 2, and 3 cannot be prevented by delaying pop operations until commit time. Instead, we can take one of two approaches. Let $T1$ be a transaction that has performed a pop.

1. Optimistic: Abort $T1$ at commit time if $T2$ has committed an operation that would cause an invalid interleaving.
2. Pessimistic: Prevent $T2$ from committing any operation until after $T1$ commits or aborts.

T-QueueO implements the optimistic method: at commit time, checks of the tail version and the head version determine whether the empty status of the queue has been modified by another, already committed transaction. T-QueueP implements the pessimistic approach, which locks the queue after a pop is performed, and only releases the lock if the transaction commits or aborts, therefore preventing any other transaction from committing any operation after the pop.

The flat combining approach can do either approach (1) or (2) to support transactions; however, the flat combining approach cannot do either without introducing overhead that reduces its performance to below that of T-QueueO or T-QueueP.

If we take approach (1), a pop cannot be performed at execution time because no locks on the queue are acquired at execution time: other transactions are allowed

to commit pops, which may pop an invalid head if this transaction aborts. Thus, in order to determine if a pop should return `true` or return `false`, a transactional pop request requires much more complexity than a non-transactional one: the thread must determine how many values the queue holds, how many values the current transaction is intending to pop, and if any other thread intends to pop (in which case the transaction aborts). The transactional push request is also more complex, as it requires installing all the pushes of the transaction. Additional flat combining calls are necessary to allow a thread to perform checks of the queue's empty status (the `<EMPTY?>` flat combining call) to determine whether the transaction can commit or must abort, and to actually execute the pops at commit time. Thus, approach (1) requires adding both more flat combining calls and more complexity to the existing flat combining calls.

If we take approach (2), the flat combining approach can either perform a pop at execution time, or delay the pop until commit time. If a pop is performed at execution time, then a thread must acquire a global lock on the queue when it calls a pop, and hold the lock until after its transaction completes: this prevents another thread from observing an inconsistent state of the queue. If a pop removes the head of the queue prior to commit, and the transaction later aborts, the popped value must be re-attached to the head of the queue. Any thread performing a pop must acquire a global lock to ensure that no other thread can commit a transaction that pops off the incorrect head of the queue (given that values may be reattached to the head if the transaction aborts). Additional flat combining calls are necessary to acquire or release the global lock.

We can also imagine a mix of approaches (1) and (2). If a transaction $T1$ executes a pop, we can disallow any pops from other transactions (using the equivalent of a global lock) but allow other transactions containing only pushes to

commit prior to $T1$ completing. This approach prevents interleavings 1 and 3, but requires performing a check of the queue's empty status, as in approach (1), if one of the transaction's pops observed an empty queue. This is because another transaction may have committed a push between the time of $T1$'s pop and $T1$'s completion. This mixed approach outperforms both approach (2) and approach (1), and is the approach described as the transactional flat combining algorithm in Section 3.3.2.

As previously noted, all possible approaches to prevent interleavings 1, 2, and 3 rely on implementing additional flat combining calls, and increasing the complexity of previously existing flat combining calls. In addition, acquisition of a global "lock" on the queue for approach (2) prevents the combiner thread from applying *all* of the requests it sees; instead, requests will either return "abort" to the calling thread or not be applied, leading to additional time spent spinning or repeating requests. Together, these modifications to the flat combining algorithm allow the combiner thread to prevent all invalid transactional history interleavings in Table 4.3.3.

We see through our experiments that these changes to the flat combining algorithm reduce its performance such that it performs worse than a naive synchronization algorithm; furthermore, we claim that these changes, or changes similar in nature, are necessary in order to provide transactional guarantees. The non-transactional, flat combining algorithm does not need to synchronize transactions, and any interleaving of operations in the history is allowed, so long as the operations return the correct results given their place in the history. The combiner thread in a non-transactional flat combining queue is therefore allowed to immediately apply all threads' operation requests in an arbitrary order. However, this property that makes flat combining so performant disappears as soon as the algorithm has to deal with transactions that do not commute, and handle invalid, non-serializable histories. In the next section, we demonstrate how changing the

queue specification to allow for greater operation commutativity in a transactional setting leads to a version of flat combining that can outperform our T-QueueO and T-QueueP algorithms. This supports our claim that the flat combining algorithm's performance is heavily dependent on the commutativity of the particular queue specification in a transactional setting.

4.4 Commutativity of the Weak Queue Specification

To demonstrate how the strong queue interface, and the corresponding commutativity of transactions, prevents a scalable transactional queue implementation, we create a different queue operation interface—a *weak queue specification*. This weak specification increases commutativity of operations and transactions, and *does* allow for a scalable transactional queue implementation. This interface is shown in Figure 4.4.1, and differs from the strong queue specification because a pop operation returns a `future<bool>` instead of `bool`. A `future<bool>` has the property that the value of the future (a boolean) is only available after the operation completes (in a non-transactional setting), or after the containing transaction commits (in a transactional setting). This interface for pop does not change the behavior or guarantees of a non-transactional, strong queue pop described in Section 3.1.

In a transactional setting, however, these pop operations must satisfy both the invariants of a concurrent queue, as well as the invariant that, at commit time, two consecutive pops in the same transaction remove consecutive values off the queue (but perhaps not from the *head* of the queue). Furthermore, the pop operations

```
void push(const value_type& v);
future<bool> pop();
```

Figure 4.4.1: Weak Queue Operations Interface

cannot pop the values that were pushed within the same transaction. The invariants for a push remain the same (two pushes in the same transaction must appear consecutively in the queue). The weak queue retains all the fairness properties of a concurrent queue: no value remains in the queue forever, because values are still removed in the order in which they are added. Like Schwarz [35], we see uses for the weak transactional queue as a buffer between producer and consumer activities, in which the exact ordering of values in the buffer is unimportant, and a transaction does not need to know exactly how many values were actually popped. Examples of valid weak queue transactional histories are shown in Table 4.4.1.

A weak transactional queue has greater commutativity than a strong transactional one. Individual pop operations within a transaction now commute with any operation, because they all return `future<bool>` when called. `COMMIT_TXN` operations, however, now return a list of `bool` values (one for each pop operation within the transaction). Thus, instead of having to synchronize *both* individual pop operations and `COMMIT_TXN` operations of transactions (as the strong transactional queue must), the weak transactional queue only needs to synchronize the `COMMIT_TXN` operations. This is because only `COMMIT_TXN` operations have return values that can change based on the ordering of operations in the history, and only `COMMIT_TXN` operations fail to commute. Informally, we can think of moving from the strong to the weak queue specification as condensing all individual strong pop operations into one operation: the `COMMIT_TXN` operation. The non-commutativity of all individual strong pop operations is now "contained" by the `COMMIT_TXN`

operation.[4]

We claim that synchronizing these COMMIT_TXN operations is no more difficult than synchronizing strong queue pops in a non-transactional setting. We use the same commutativity reasoning we used to reason about commutativity between individual strong pops and pushes to reason about commutativity between weak COMMIT_TXN operations. A weak COMMIT_TXN operation can be described as a pair of a pop-group operation and a push-group operation: in order to satisfy our specification, all pops of the transaction need to be installed together at commit time, and all pushes of the transaction need to be installed together at commit time. It is clear that a pop-group that encounters an empty queue will not commute with another pop-group or another a push-group; thus, we see that a pop-group has equivalent commutativity behavior to a strong pop, and a push-group has equivalent commutativity behavior to a strong push. Indeed, performing a weak COMMIT_TXN operation, like performing a strong pop operation, requires protected access to the queue's head, and does not scale.

To further support our point, we implement a weak transactional flat combining queue, and demonstrate that flat combining's synchronization mechanism for a strong, non-transactional pop can be used by this weak transactional queue with minimal modifications.

[4]Again, we can consider adding front operations to create front-pop operations (a pop operation immediately preceded by a front operation). A weak front-pop returns a future representing the contents of the front and the return value of the pop, which is instantiated at commit time. Given this specification, all weak front-pops commute (because they all return futures), but COMMIT_TXN operations fail to commute in precisely those scenarios in which strong front-pops fail to commute. Thus, moving from the strong queue specification to the weak queue specification is equivalent to condensing of all strong front-pop operations into one operation, the COMMIT_TXN operation.

Valid History Interleaving	Serialized Forms of History
// Q empty (T1, START_TXN, ()) (T2, START_TXN, ()) (T2, Q.pop(), ()) (T1, Q.pop(), ()) (T1, Q.push(a), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {false}) (T2, COMMIT_TXN, {true/false})	// Q empty (T1, START_TXN, ()) (T1, Q.pop(), ()) (T1, Q.push(a), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {false}) (T2, START_TXN, ()) (T2, Q.pop(), ()) (T2, COMMIT_TXN, {true}) ------------------------------ // Q empty (T2, START_TXN, ()) (T2, Q.pop(), ()) (T2, COMMIT_TXN, {false}) (T1, START_TXN, ()) (T1, Q.pop(), ()) (T1, Q.push(a), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {false})
// Q empty (T1, START_TXN, ()) (T2, START_TXN, ()) (T1, Q.push(a), ()) (T1, Q.pop(), ()) (T1, Q.pop(), ()) (T2, Q.pop(), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {false, false}) (T2, COMMIT_TXN, {true/false})	// Q empty (T1, START_TXN, ()) (T1, Q.push(a), ()) (T1, Q.pop(), ()) (T1, Q.pop(), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {false, false}) (T2, START_TXN, ()) (T2, Q.pop(), ()) (T2, COMMIT_TXN, {true}) ------------------------------ // Q empty (T2, START_TXN, ()) (T2, Q.pop(), ()) (T2, COMMIT_TXN, {false}) (T1, START_TXN, ()) (T1, Q.push(a), ()) (T1, Q.pop(), ()) (T1, Q.pop(), ()) (T1, Q.push(a), ()) (T1, COMMIT_TXN, {true, false})

Table 4.4.1: Examples of valid weak queue transaction histories. The history interleavings shown on the left are valid regardless of whether $T2$ returns `true` or `false`, because we can find a serialized form of the history in either case.

4.4.1 The Weak Transactional Flat Combining Queue

The Weak Transactional Flat Combining Queue (WT-FCQueue) demonstrates how the flat combining technique's performance depends upon the commutativity of operations of a particular queue specification, and how this commutativity changes in a transactional setting. It also supports our claim that the commutativity in a weak transactional queue is equivalent to the commutativity in a strong non-transactional queue.

WT-FCQueue implements the weak transactional queue specification using *futures*: instead of returning `bool`, a pop will return a future. The future will be instantiated with a `true` or `false` boolean value at commit time, but prior to that time, the future's value cannot be accessed. At commit time, our implementation performs a pop using the non-transactional flat combining `<POP>` request, and assigns the corresponding boolean return value to the corresponding future. All pushes from the transaction are also installed at commit time using the `<PUSH, write_list>` request described earlier. We note that both pop and push requests do not need to access the queue during execution time. This allows us to minimize the number of flat combining calls: we do not need to generate additional flat combining calls during transaction execution because the queue does not need to be accessed until commit time.

Our implementation satisfies a weaker specification than the weak queue specification described above: in our implementation, pops are not guaranteed to be consecutive. However, we hypothesize that a queue that installs all pops at once—by posting, at commit time, a `<POP, num_pops>` request that returns a list of booleans—will perform no worse than our current method of pops (or may perhaps even perform better). This is because the synchronization is still restricted to the

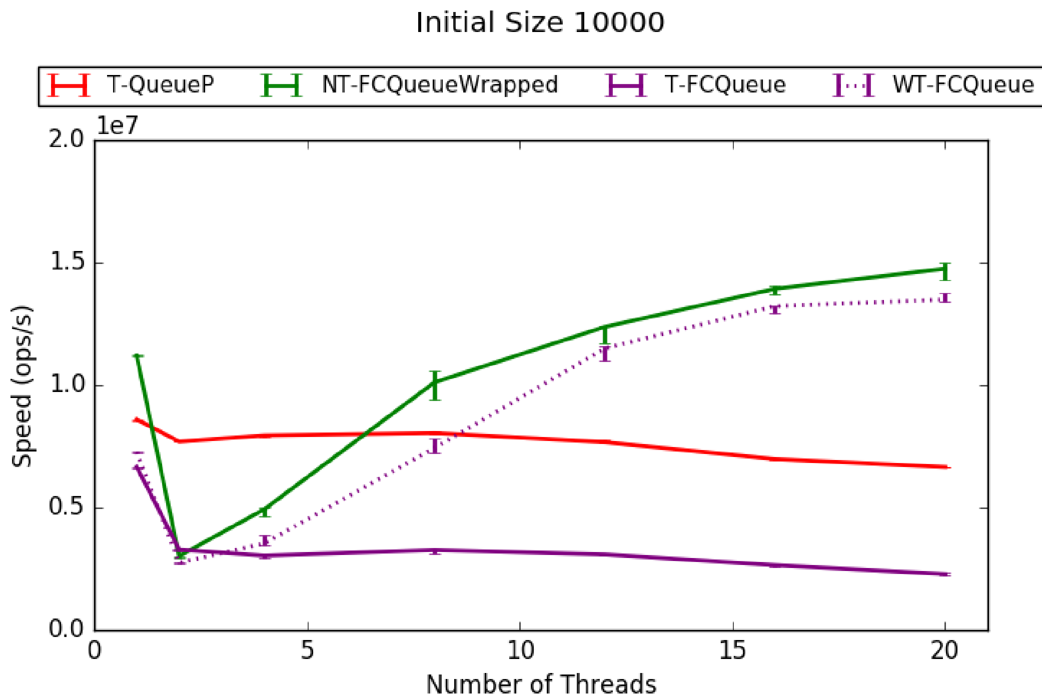


Figure 4.4.2: WT-FCQueue Performance: Multi-Thread Singletons Test

point at which the `COMMIT_TXN` operations occur in the history, and in both our actual and proposed implementations, only a single thread can access the queue at this point.

4.4.2 Evaluation and Results

We evaluate the weak transactional flat-combining queue on the same benchmarks described in Section 3.4.1 to compare against the strong transactional flat-combining queue (T-FCQueue), T-QueueP, and NT-FCQueueWrapped. Selected results are shown in Figure 4.4.2.

While WT-FCQueue does not perform as well as its non-transactional counterpart, NT-FCQueue, the performance of WT-FCQueue exceeds that of T-QueueO, T-QueueP, and T-FCQueue, which all provide transactional guarantees

under the strong queue specification. We see gains in performance over T-QueueP up to 1.5× as the number of threads accessing the queue increases to 20; WT-FCQueue begins to outperform T-QueueP as the number of threads increases past 7. WT-FCQueue outperforms T-FCQueue starting at 4 threads and achieves performance up to about 5× by 20 threads.

WT-FCQueue does not experience any aborts. This is because the weak transactional flat combining algorithm never needs to prevent other threads' requests from being performed by the combiner thread. Because of the lack of aborts, WT-FCQueue significantly outperforms T-FCQueue; this demonstrates the effectiveness of the flat combining technique in the weak transactional setting.

Our results show significant improvements in the performance of the flat combining algorithm when satisfying the weak transactional queue specification, compared to its performance when satisfying the strong transactional specification. This demonstrates that the choice of queue specification, and therefore the commutativity of queue operations, directly affects the effectiveness of the flat combining algorithm in a transactional setting. We argue that the modifications to provide transactional guarantees with our strong queue specification critically impair the performance of the flat combining algorithm, and that these modifications cannot be avoided. Thus, scalable performance of a strong flat combining queue is unlikely to be achievable in a transactional setting.

5

Hashmap Algorithms and Analysis

This chapter investigates different concurrent and transactional algorithms for hashmaps, and demonstrates how a concurrent hashmap algorithm—cuckoo hashing—can retain the benefits from its performance optimizations even in a transactional setting. We begin with an overview of concurrent and transactional hashmap algorithms, and evaluate how these hashmaps perform on several microbenchmarks. By examining the commutativity of hashmap operations and hashmap transactions, we discuss why and how a transactional cuckoo hashing algorithm, unlike the flat combining algorithm, retains the scalability and

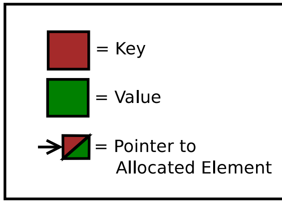
```
// find searches for key k, storing the
// associated value it finds in v
// succeeds only if the key is in the map
bool find(const key_type& k, mapped_type& v);

// insert adds the key-value pair (k, v) to the map
// succeeds only if the key is not already in the map
bool insert(const key_type& k, const mapped_type& v);

// erase removes the element corresponding to k from the map
// succeeds only if the key is in the map
bool erase(const key_type& k);
```

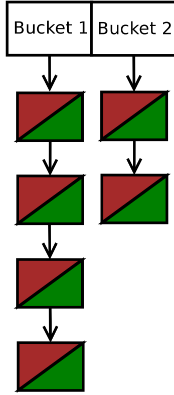
Figure 5.1.1: Hashmap Operations Interface

characteristic behaviors of non-transactional cuckoo hashing.

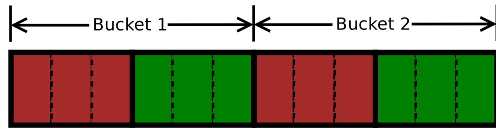
5.1 Algorithms

STO provides transactional hashmaps that support find, insert, and erase operations with the interface shown in Figure 5.1.1. This section presents the concurrent and transactional hashmap algorithms we implemented and analyzed in our work. We will use some general terminology: an *element* refers to the key-value pair inserted into the hashmap, a *bucket* is a set of elements, and a hashmap consists of a set of buckets. Various hashmap algorithms use various methods to place elements in buckets and track how buckets and elements are modified. Figure 5.1.2 depicts the bucket structures of the various hashmaps described in this section.

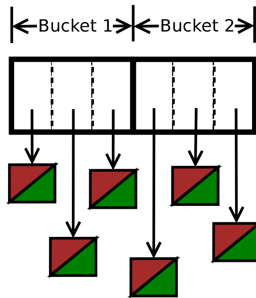




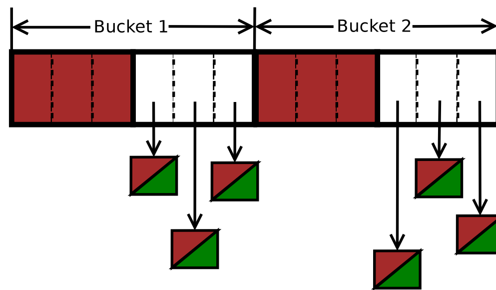
(a) T-Chaining



(b) NT-Cuckoo and T-CuckooNA



(c) T-CuckooA



(d) T-CuckooKF

Figure 5.1.2: This figure depicts two buckets in each type of hashmap. The static size of a bucket in each cuckoo hashmap is set to 3. Arrows represent pointers to allocated elements; contiguous memory is represented horizontally.

5.1.1 Transactional Chaining Hashmap

The transactional chaining hashmap is a concurrent, transactional hashmap implemented using a standard chaining algorithm. If two elements are mapped to the same bucket, they are chained in a linked list, shown in Figure 5.1.2 (a). Thus, the worst case time complexity for a lookup, insert, or erase is $O(n)$. Inserts require allocating an element corresponding to the key-value pair to insert into the bucket. Each bucket is associated with a *bucket version* that increments when any transaction commits an erase or insert of an element in the bucket. A thread uses the bucket version to verify that no other thread has added an element that was absent during one of its finds. In addition, the bucket version acts as a lock to synchronize access to the bucket. Each inserted element is associated with an *element version* that tracks if the value of the element has been modified or if the element has been removed. Each element version is tied to the inserted element's memory address.

Elements are inserted eagerly at execution time with a *phantom* flag, allowing another transaction that sees one of these phantom, uninstalled elements to realize it is viewing an inconsistent state of the map and abort. If a transaction that performs an insertion later aborts, these phantom elements are removed from the map. Otherwise, the phantom mark is erased during commit. An alternative approach would be to insert all elements at commit time. However, this requires either relying on the bucket version to determine if another transaction has inserted the same element (which would result in false aborts since the bucket version increments for *any* inserted value), or redoing the search for the element to see if the insertion can still occur. Thus, we insert at execution time to allow for more fine-grained validation checks at commit time, and to reduce redundant computations.

Erasures are delayed until commit time (an optimistic approach). The same

logic that we used for inserts does not apply here. If we used the same procedure as used by an insert, we would flag the bucket that contained the erased element to indicate that a (still-executing) transaction performed an erase of an element in the bucket. Any thread seeing the flag would know that the bucket is in an inconsistent state, and would abort. However, this loses the per-element granularity of checks and causes false aborts. Thus, erasing elements at execution time would decrease the granularity of validation checks, rather than increase it (as was the case for inserting elements). Delaying erasures until commit time requires careful handling of *read-my-writes* scenarios, such as deleting an element inserted in the same transaction.

5.1.2 Non-Transactional Cuckoo Hashmap

The Non-Transactional Cuckoo Hashmap (NT-Cuckoo) implements a concurrent, non-transactional cuckoo hashing algorithm. We modify an implementation of a dynamically-resizing, concurrent cuckoo hashmap [11] by simplifying several procedures; the main difference is that our cuckoo hashmap is statically sized.

We choose the cuckoo hashing algorithm as a potential transactional algorithm because the cuckoo hashing algorithm has several advantages over a chaining one. For one, cuckoo hashmaps have been shown to outperform chaining hashmaps when the hashmap can mostly fit in cache. Although cuckoo hashing has the disadvantage of performing two memory accesses for a lookup (as described below), the overhead from performing memory accesses is eliminated when the map can fit in cache. Furthermore, modern processors can optimize these memory accesses to alleviate some of the overhead [43]. Cuckoo hashmaps with large bucket sizes also outperform chaining hashmaps at high loads (when the average number of elements per bucket

is high) [34]. As we explain below, this is because the cost of a lookup is always constant time, whereas a chaining hashmap may experience $O(n)$ lookup time. In addition, a lookup that encounters a long chain in a chaining hashmap will follow $O(n)$ pointers, whereas a lookup in a cuckoo hashmap will need to only access two buckets, each of which is a contiguous array in memory.

In a concurrent setting, both cuckoo and chaining hashmaps must synchronize access to buckets. Because the granularity of synchronization is no different for the two algorithms, concurrent cuckoo hashing is still expected to outperform chaining for the same reasons (better cache usage due to lack of pointers, and improved asymptotic time complexity for all operations).

The cuckoo hashing algorithm works as follows: each element is placed in one of two buckets; these buckets are determined by two different hash functions. A bucket has a fixed number of elements it can hold. This means that lookups and erasures only require executing two hash functions and checking the contents of two buckets (an $O(1)$ operation).

Inserts run in amortized $O(1)$ time; the running time may occasionally be $O(n)$. If an element e is hashed by the first hash function to a bucket that is already full, the algorithm attempts to place e in its alternate bucket by hashing e with the second hash function. If both buckets are full, cuckoo shuffling occurs. This process evicts an element e' in one of e's buckets and places e' in e''s alternate bucket. If e''s alternate bucket is full, an element e'' is ejected from this bucket, and so on. As long as the cuckoo shuffling does not encounter a bucket cycle, e can now be placed in one of its buckets, as the removal of e' has made space for e. However, if the shuffling encounters a bucket cycle, the hashmap raises an `out_of_space` assertion error. We can imagine an alternative implementation that allows the hashmap to grow in number of buckets or otherwise change its hash functions (which requires

reinserting all elements), but for simplicity, we keep the hashmap statically sized.

Because the buckets are statically sized, elements are contained in fixed-size key and value arrays and therefore do not require extra allocations. The bucket structure is shown in Figure 5.1.2 (b).

5.1.3 Transactional Cuckoo Hashmap

We hypothesize that, unlike the flat combining algorithm, the key optimizations of the cuckoo hashing algorithm—better cache usage than a chaining hashmap when the number of elements in each bucket is high, and constant time (or amortized constant time) operations—are not crippled by integrating the algorithm into a transactional setting. This is because the modifications necessary to support transactions do not interfere with the optimizations taken by cuckoo hashing. We investigate this further in our evaluation (Section 5.2) and hashmap commutativity discussion (Section 5.3).

The transactional cuckoo hashmap comes in three flavors: two that allocate an element per insertion, and one that does not perform any allocations. All cuckoo hashmaps instrument the non-transactional cuckoo hashmap with STO calls that provide transactional guarantees. Both allocating and non-allocating versions use the same synchronization algorithm: like the chaining hashmap, each bucket has a *bucket version*, and each element has an *element version*. A lock separate from the bucket version is used to synchronize access to buckets for cuckoo shuffling. Because elements can be moved into a bucket because of cuckoo shuffling as well as an external insertion call, the bucket version increments only when an element *not already contained in the map* is inserted into the bucket (i.e., elements inserted via a call to insert and not via cuckoo shuffling). Elements are inserted at execution time

with a *phantom* flag that is then erased at commit time, and erasures are delayed until commit time.

The two variants of the allocating transactional cuckoo hashmap allocate elements upon insertion. One variant (T-CuckooA) consists of buckets containing pointers to the elements (Figure 5.1.2 (c)), allowing STO to track elements by their memory address to verify element versions at commit time. The key-fragments variant (T-CuckooKF) expands buckets to contain both an array of keys and an array of element pointers (Figure 5.1.2 (d)). This allows for a lookup to compare keys directly, because the keys are themselves contained in the bucket. Without the key-fragments, the lookup would have to follow pointers contained in the bucket to access and compare the key value. For many workloads, the key-fragments variant reduces the number of cache line accesses. STO can still track elements by their memory address in the element pointer array, but these pointers are only accessed if the value of the element is needed. This only happens during a present lookup, which accesses exactly one pointer, namely that of the searched-for element.

The non-allocating transactional cuckoo hashmap (T-CuckooNA) consists of buckets containing a fixed-sized array of elements (Figure 5.1.2 (b)). In this variant of the cuckoo hashmap, STO tracks elements by their keys rather than their memory addresses. Therefore, to verify if an element version has changed at commit time, the check procedure performs a find of the element using the key (searching at most two buckets) and validates the corresponding element version. Although this reduces the number of allocations, elements can now move between buckets and invalidate the values in the previous bucket: this greatly complicates the process of correctly checking and synchronizing element version reads. Our transactional algorithm for this hashmap reflects this complexity: the tracking set size is not minimal, and therefore we achieve more coarse-grained locking at commit time. For

example, the non-allocating cuckoo hashmap locks both bucket versions to check any element version, because the element could move between buckets during the check. The allocating cuckoo hashmap needs only to lock the element version, because the hashmap has access to the element's unchanging memory address.

Although a non-allocating transactional cuckoo hashmap is more complex, we build it to improve the allocating transactional cuckoo hashmaps' cache performance during lookups: the non-allocating algorithm does not need to follow any pointers. Nevertheless, we may not see a great improvement in cache usage because we must execute approximately double the number of lookups: each find requires performing two lookups (one during execution, and another during commit to check if the element version is still valid).

5.2 Evaluation

As we did with the queue, we evaluate all hashmaps on a set of microbenchmarks to determine their scalability and performance. We aim to provide evidence for our hypothesis that, unlike the flat combining queue, the cuckoo hashmap will retain its scalability and behavior even in a transactional setting.

5.2.1 Microbenchmarks

All experiments are run on the same machine as the queue experiments (with 100GB DRAM, two 6-core Intel Xeon X5690 processors with hyperthreading clocked at 3.47GHz and a 64-bit Linux 3.2.0 operating system). All benchmarks and STO data structures are compiled with `g++-5.3`. In all tests, threads are pinned to cores, with at most one thread per logical core. In all performance graphs, we show the median of 5 consecutive runs with the minimum and maximum performance

results represented as error bars.

Cache misses are recorded by running the benchmark with 8 threads, with each thread performing 10M transactions, under the profiling tool Performance Events for Linux (`perf`). The sampling period is set to 1000, meaning that every 1000th cache miss is recorded. We report the number of cache misses reported by perf (approximately 1/1000 of the actual number of cache misses).

Parameters

Ratio of Find:Insert:Erase Operations. The ratio of inserts:erasures is kept at 1 to ensure that the hashmap does not always become empty or only grow in size. Keys are drawn uniformly at random from a predetermined range, in such a way that half of the inserts will succeed (the key to insert is not present) and half of the erasures will succeed (the key to erase is present). Tests of 5% insert, 5% erase, and 90% find simulate the most likely use cases for hashmaps [36]. Tests of equal proportion (33%) of all operations investigate how the hashmap reacts to an increased rate of inserts and erasures.

Operations per transaction. We choose to run all tests comparing transactional to non-transactional data structures using single-operation transactions. As discussed in Section 3.4.1, this provides a more fair evaluation of transactional data structures against concurrent ones. In addition, it allows us to minimize the differences between transaction hashmap implementations so we can get a baseline comparison.

Number of buckets. All hashmaps statically set the number of buckets in the data structure. The number of elements allowed in one bucket of a cuckoo hashmap is fixed at a particular value, which we will call the *maximum fullness.* The *capacity* (number of buckets $\times$ number of elements per bucket) of the cuckoo hashmap is

fixed at some finite value because the cuckoo hashmap has a fixed size bucket; the chaining hashmap has no fixed capacity because a bucket's chain can grow arbitrarily long. The number of buckets and the size of each bucket affect the number of cache lines accessed during the test (for example, a larger hashmap may not be expected to fit into the L2 cache, whereas a small hashmap at full capacity will fit entirely in cache). During all tests, the number of keys present in the hashmap is not allowed to outgrow its capacity.

Fullness. Fullness indicates the ratio of the number of keys to the number of buckets. This determines the average number of elements to be found per bucket. In the discussion that follows, we use fullness to indicate the load on the chaining hashmap, and the number of items in the buckets of the cuckoo hashmap. Greater fullness therefore corresponds to a greater load in the chaining hashmap, and a higher occupancy of each bucket in the cuckoo hashmaps. (Note, however, that chaining hashmaps can never truly be "full", as they have unlimited capacity.)

We set fullness at steady state by picking a maximum fullness for the cuckoo hashmap: the tests are implemented in such a way that, at steady state, fullness is expected to be 75% of the maximum fullness of the cuckoo hashmap. This is controlled by picking a maximum key value. The maximum key value of inserted elements is twice the number of elements the hashmap will contain when its size reaches a steady state. The initial size of the hashmap is set to its size at steady state to ensure that the size of the hashmap should remain relatively constant throughout the benchmark.

Multi-Thread, Variable-Capacity Singletons Test

This test performs find, insert, and erase singleton transactions, with the probability of each operation specified by a probability distribution. We run the test with

varying numbers of threads; each thread runs 5 million singleton transactions. To test hashmap behavior at different sizes, we set both the number of buckets in the hashmap, and the maximum fullness of the cuckoo hashmap. The steady-state size is 75% maximum capacity (of the cuckoo hashmap).

We run two variations of this test: one with a probability distribution of 33% insert, 33% erase, and 34% find; and another with a probability distribution of 5% insert, 5% erase, and 90% find.

5.2.2 Overview of Results

We measure the speed (operations performed per second), abort rate, and cache performance (number of cache misses) of each hashmap. The full results can be found in Appendix B. As we did with the queue results, we proceed in our discussion by first giving an overview of our conclusions, and then showing how we reach these conclusions through a sequence of hypotheses. We draw the following conclusions from our results (corresponding to our four hypotheses):

1. The transactional, key-fragments cuckoo hashmap achieves the best overall cache performance.
2. When the hashmap can fit entirely or mostly in cache, the transactional cuckoo hashmaps outperform the transactional chaining hashmap.
3. When the cuckoo hashmaps have a high maximum fullness, and the average number of elements per bucket is high, the transactional cuckoo hashmaps outperform the transactional chaining hashmap.
4. The number of cache misses is strongly correlated with overall performance, particularly because the abort rate of our tests is negligible.

Our overall conclusion is that the relative performance of transactional cuckoo

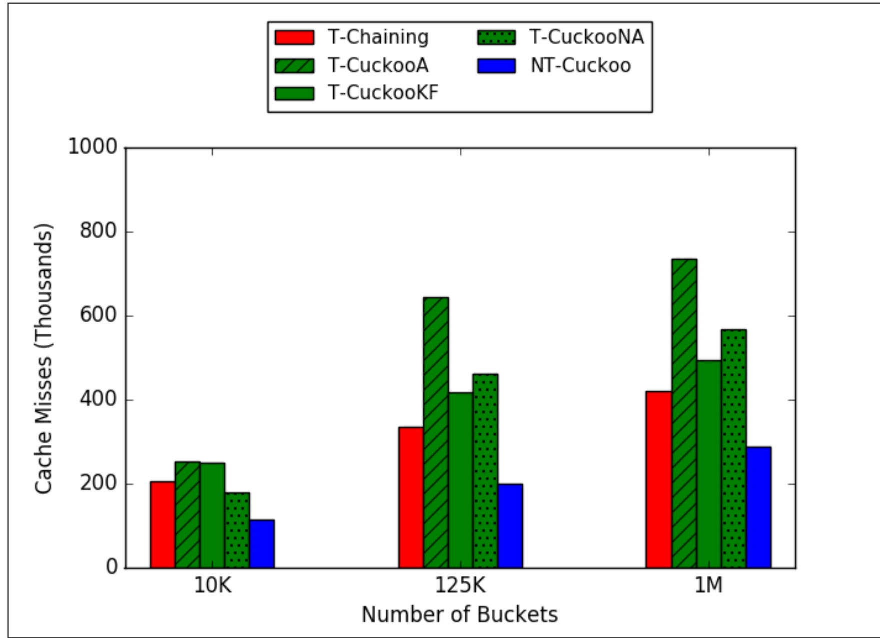
hashmaps and transactional chaining hashmaps mirrors the relative performance expected of their non-transactional counterparts.

5.2.3 Hypothesis 1

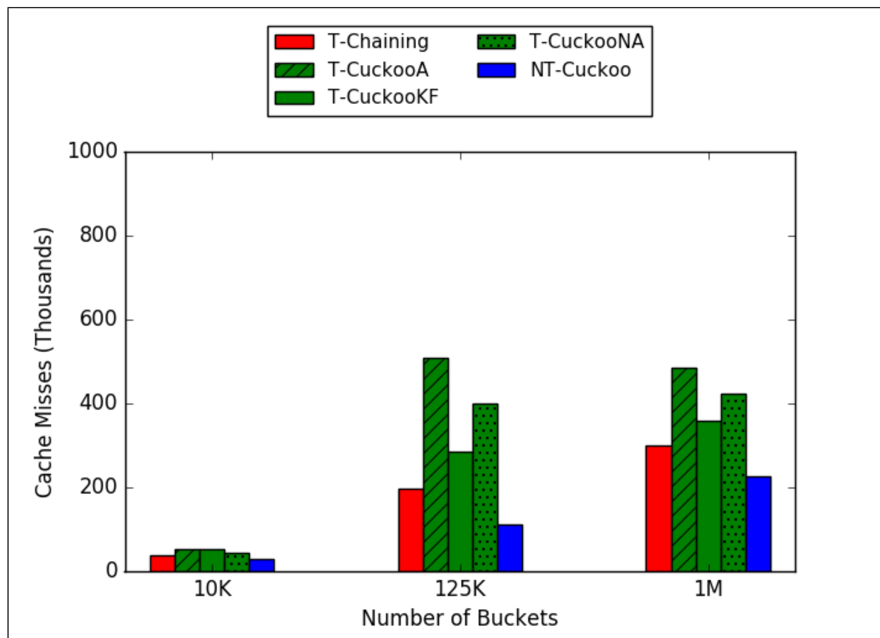
The non-allocating transactional cuckoo hashmap (T-CuckooNA) achieves the best cache usage out of all transactional hashmaps. (Not Supported)

The number of cache misses is influenced by the number of allocations, the patterns in which these allocations are accessed, and the patterns in which the hashmap buckets themselves are accessed. Our results in Figures 5.2.1, 5.2.2, and 5.2.3 demonstrate that, as expected, a pattern of 90% finds (5% inserts/5% erases) achieves better cache performance than a pattern of 33% finds/inserts/erases. An insert or erase does strictly more work than does a find, since both operations first invoke a find to detect if the key is present, and then must allocate or remove the element.

The non-transactional cuckoo hashmap experiences the least number of cache misses on all tests, which we expect given its lack of transactional instrumentation. Somewhat surprisingly, the non-allocating, transactional cuckoo hashmap does not achieve the best overall cache performance. As we see in Figure 5.2.1, T-Chaining has the best cache performance among the transactional hashmaps when the maximum fullness (i.e., the maximum load on the hashmap) is set to 5. This is likely due to the low number of pointer accesses when T-Chaining performs a find: the chains are relatively short, and more buckets are likely to be empty. In comparison, the cuckoo hashmaps must search through two entire bucket arrays on an absent find.

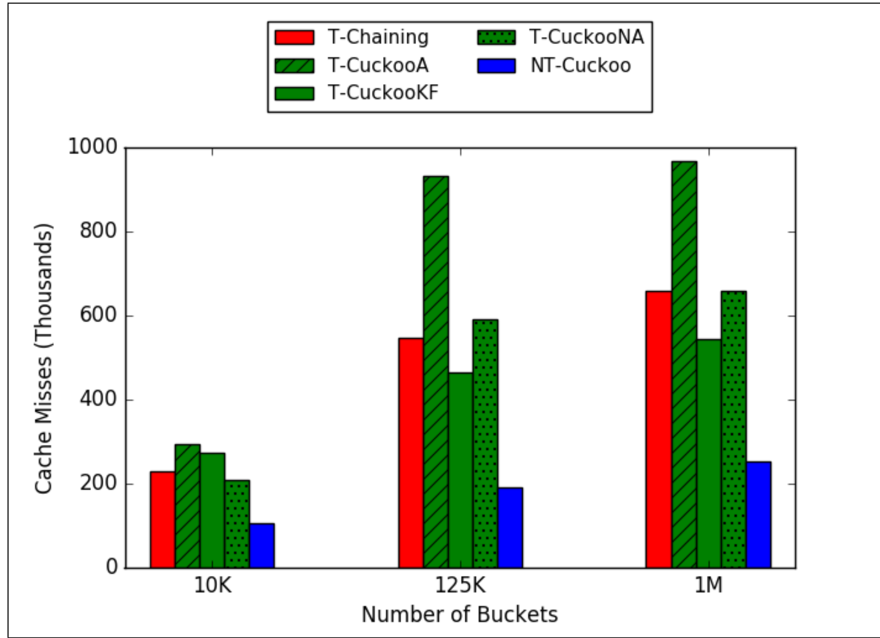


33%Find, 33%Insert, 33%Erase

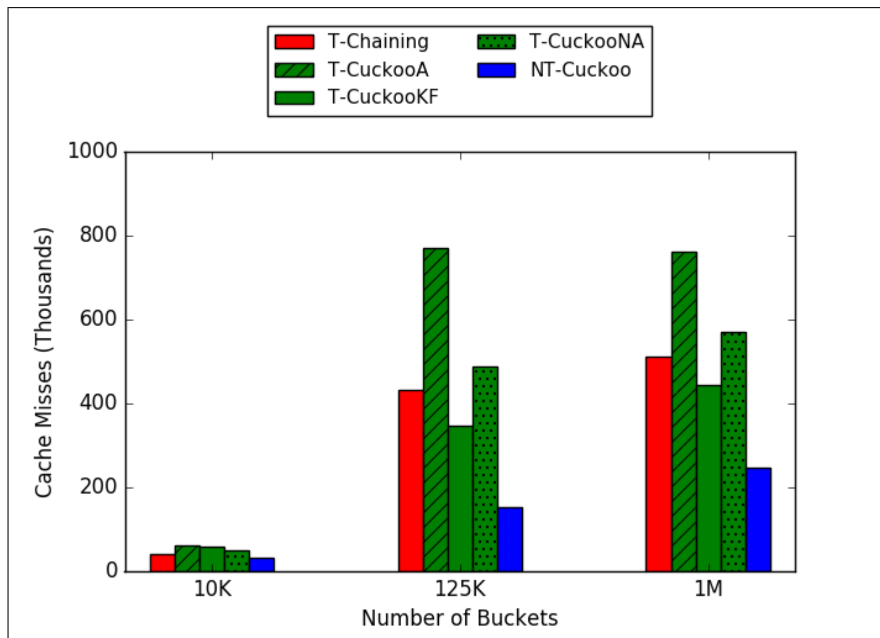


90%Find, 5%Insert, 5%Erase

Figure 5.2.1: Hashmap Cache Misses (Maximum Fullness 5): T-Chaining has the best cache performance of the transactional hashmaps.

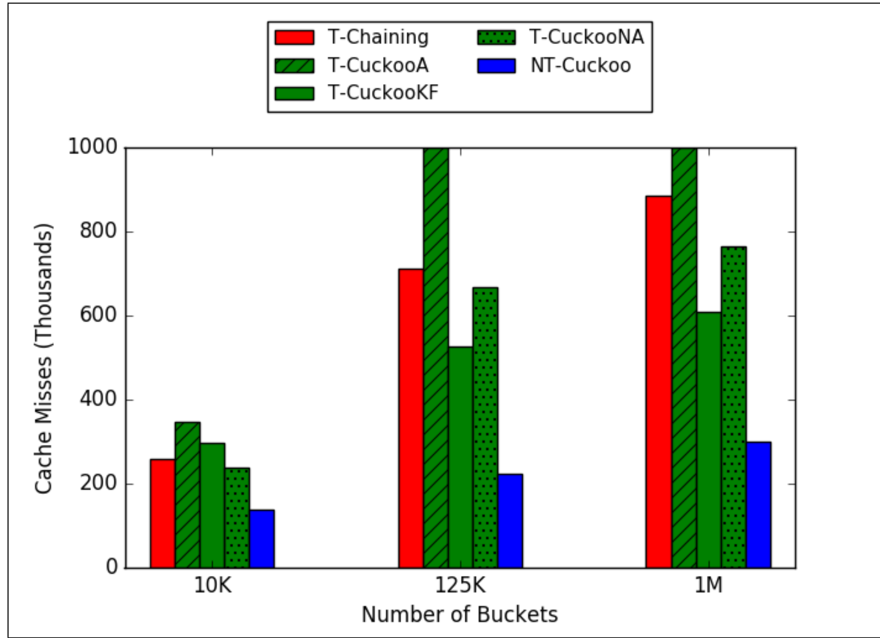


33%Find, 33%Insert, 33%Erase

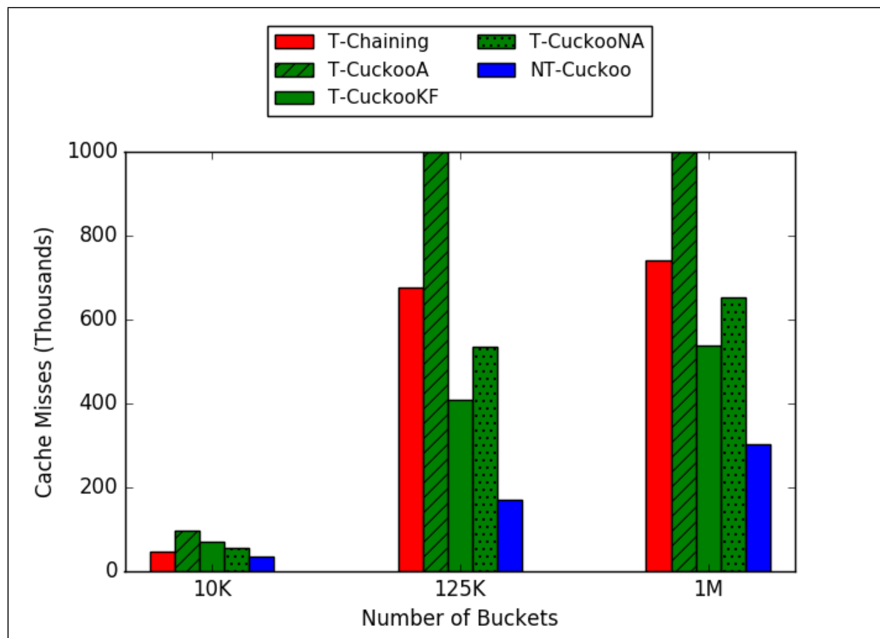


90%Find, 5%Insert, 5%Erase

Figure 5.2.2: Hashmap Cache Misses (Maximum Fullness 10): T-CuckooKF has the best cache performance of the transactional hashmaps, followed by T-CuckooNA.



33%Find, 33%Insert, 33%Erase



90%Find, 5%Insert, 5%Erase

Figure 5.2.3: Hashmap Cache Misses (Maximum Fullness 15): T-CuckooKF has the best cache performance of the transactional hashmaps, followed by T-CuckooNA. We see that the difference between T-CuckooKF's cache performance and T-Chaining's cache performance is increasing as fullness increases.

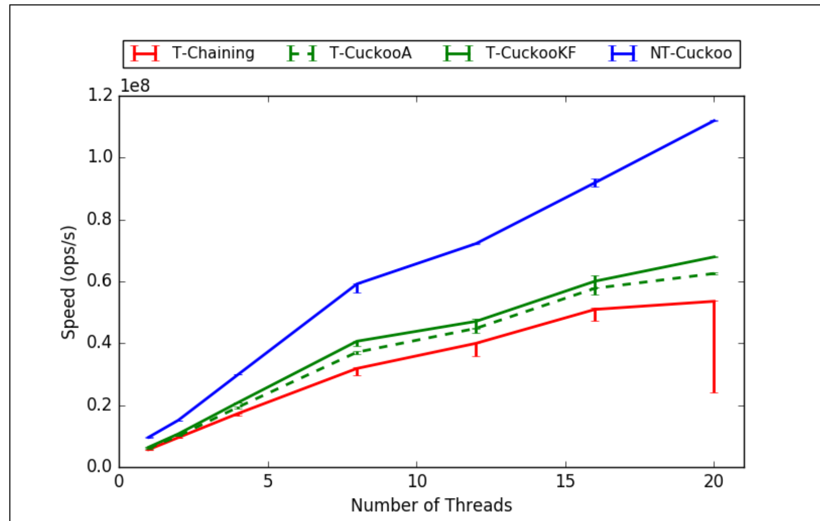
Figures 5.2.2 and 5.2.3 indicate that when the hashmap has longer chains or bucket sizes (maximum fullness 10 or 15) and cannot fit entirely in cache (i.e., the number of buckets is 125K or 1M), T-CuckooKF achieves the best transactional hashmap cache performance, followed by T-CuckooNA. T-Chaining and T-CuckooA experienced a lower cache performance because every find in T-Chaining and T-CuckooA requires following pointers. As the number of elements per bucket increases, T-CuckooA and T-Chaining follow an increasing number of pointers per find, and their cache performance worsens. We also note that the difference in the number of cache misses between T-Chaining and T-CuckooKF increases as maximum fullness increases. This occurs because, on average, T-Chaining's chains are longer and T-Chaining has to follow more pointers each time it performs a lookup.

A find will never follow pointers in the buckets of T-CuckooNA, and will not follow pointers in the buckets of T-CuckooKF if the element is absent. We conjecture that T-CuckooKF has fewer cache misses than T-CuckooNA because the additional memory accesses from following pointers is minimal (a present find follows only one pointer), and the additional lookups in T-CuckooNA mean that T-CuckooNA accesses approximately twice the number of buckets per operation compared to T-CuckooKF.

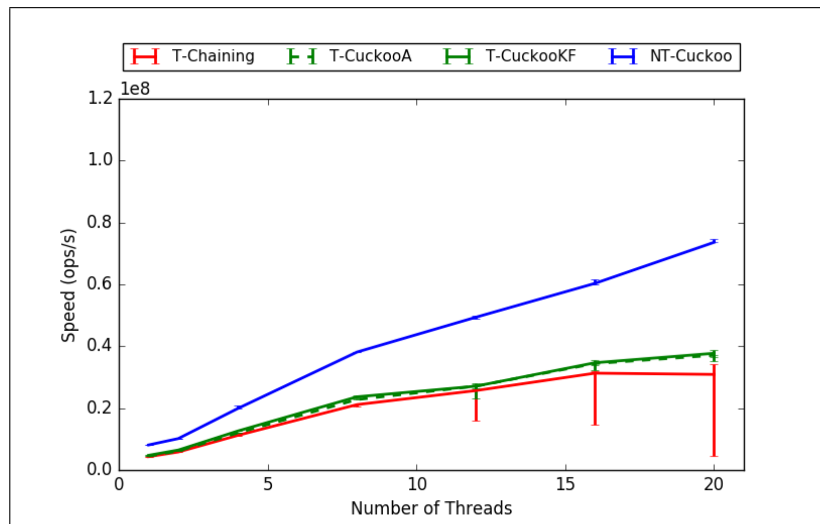
NOT SUPPORTED: T-CuckooNA has worse cache performance than T-CuckooKF when the hashmap has 125K or 1M buckets, and only slightly better cache performance when the hashmap has 10K buckets. At maximum fullness 5, T-Chaining has the best cache performance; at maximum fullness 10 and 15, T-CuckooKF has the best cache performance.

5.2.4 Hypothesis 2

When the hashmap can fit entirely or mostly in cache, the transactional cuckoo hashmaps outperform the transactional chaining hashmap. (Supported)



90F/5I/5E



33F/33I/33E

Figure 5.2.4: Hashmap Performance: 10K Buckets, Maximum Fullness 5. The transactional cuckoo hashmaps outperform T-Chaining.

To evaluate this hypothesis, we look at our results for a map with 10K buckets with a maximum fullness of 5, which fits entirely in cache (Figure 5.2.4).[1] For all hashmaps, performance with 90% finds (5% inserts/erases) is approximately 1.5× that of the 33% finds/inserts/erases. This is expected: a find is both faster to perform than an insert or erase, and a find requires only reading a bucket version and performs no writes. On both benchmarks, the cuckoo hashmaps achieve performance approximately 1.2× that of T-Chaining. There is little difference between the performance of T-CuckooA and the performance of T-CuckooKF, because cache performance does not play a significant role in these benchmarks.

We note that this result is consistent with the claim made in Section 5.1.2 that cuckoo hashmaps outperform chaining hashmaps when the hashmap can fit in cache.

SUPPORTED: The transactional cuckoo hashmaps outperform the transactional chaining hashmap when the hashmap can fit in cache.

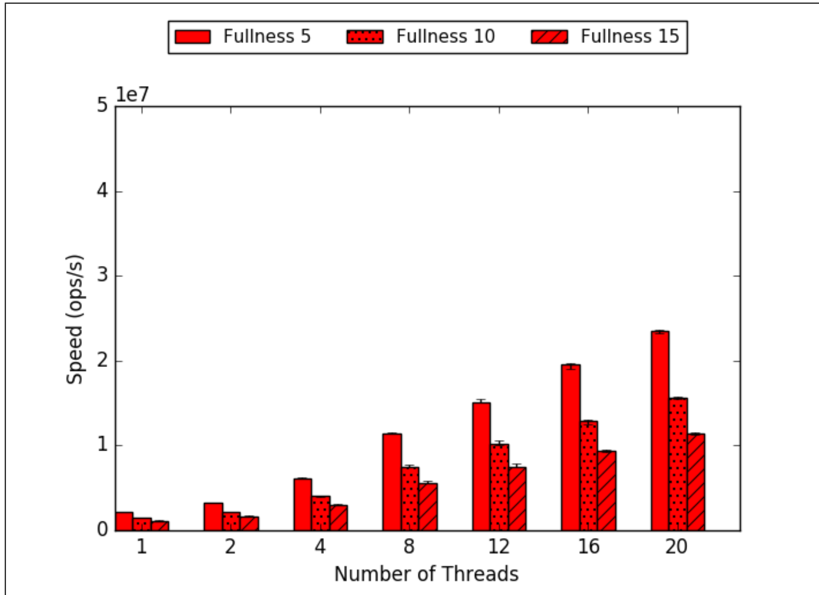
5.2.5 Hypothesis 3

As maximum fullness increases, T-CuckooKF outperforms T-Chaining by a greater margin. (Supported)

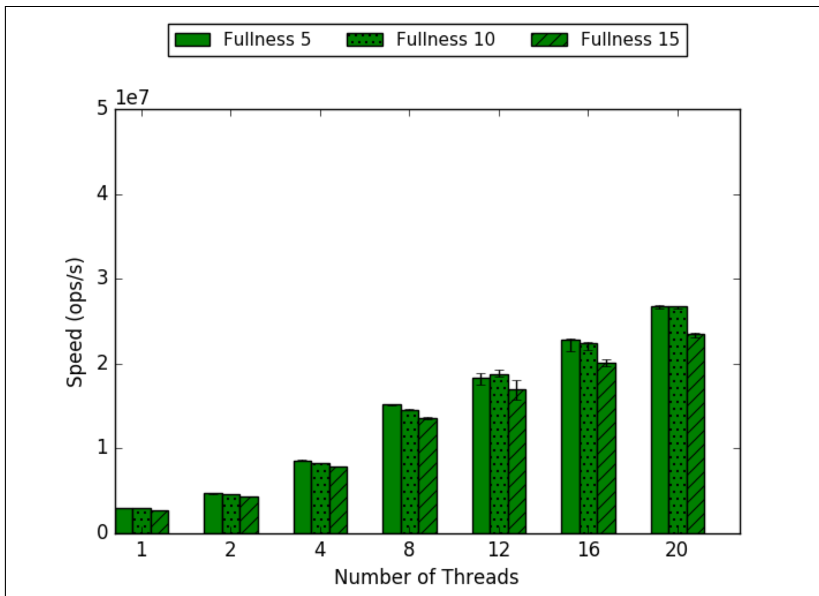
We analyze our results for hashmaps with various number of buckets, and compare their performance with a maximum fullness of 5 to that with a maximum fullness of 15. We show the results for hashmaps with 1M buckets (full results are in Appendix B).

We first consider the results from the 33% finds/inserts/erases test. Our results

[1] Our preliminary performance results on the Multi-thread Singletons Test show the same pattern as our cache performance results: T-CuckooNA performs better than T-CuckooA on average, and worse than T-CuckooKF. We omit these from the performance results presented here because the correctness of T-CuckooNA has not been confirmed.

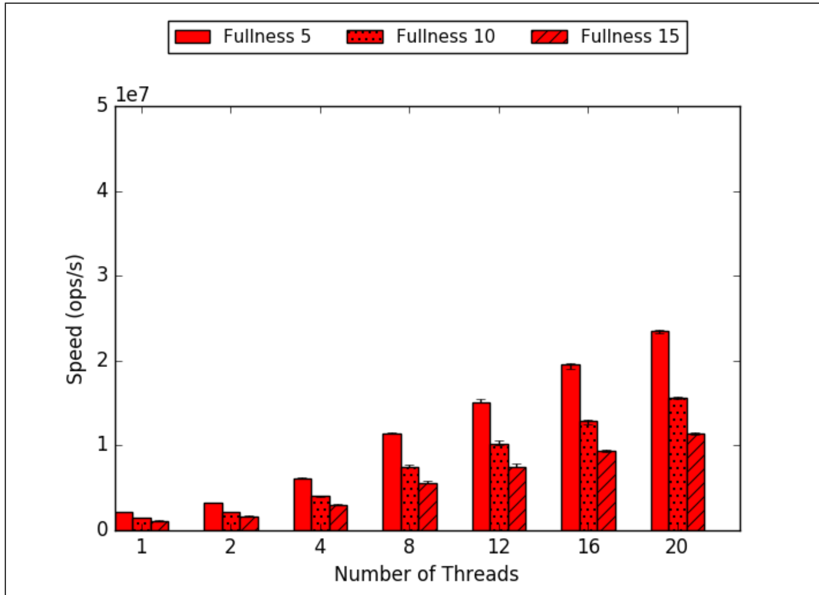


T-Chaining

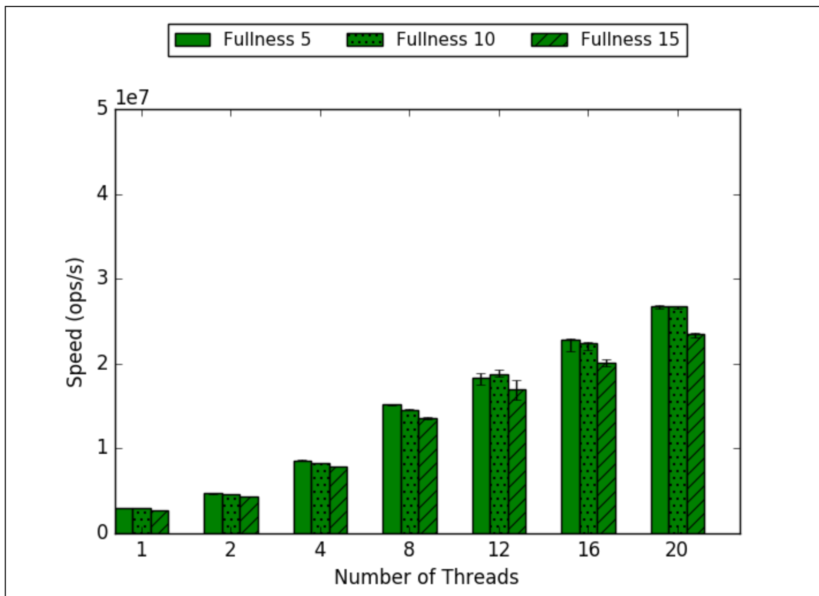


T-CuckooKF

Figure 5.2.5: T-Chaining vs. T-CuckooKF: Performance as fullness increases (1M Buckets, 33F/33I/33E). T-Chaining experiences a greater drop in performance than does T-CuckooKF as fullness increases. T-Chaining's performance drops over 50% as fullness goes from 5 to 15.



T-Chaining



T-CuckooKF

Figure 5.2.6: T-Chaining vs. T-CuckooKF: Performance as fullness increases (10K Buckets, 90F/5I/5E). T-Chaining experiences a greater drop in performance than does T-CuckooKF as fullness increases. T-Chaining's performance decreases by over 50% with 1M buckets as fullness increases.

show that T-CuckooKF achieves performance 1.5–2× that of T-Chaining as maximum fullness increases to 15, regardless of the number of buckets (Figure 5.2.5). This is caused by a decrease in the performance of T-Chaining as fullness increases: performance decreases by approximately 33% when the hashmap has 10K buckets, and by over 50% when the hashmap has 1M buckets (Figure 5.2.5). T-CuckooKF's performance, in contrast, decreases only slightly or remains essentially unchanged.

Our results from the 33% finds/inserts/erases test therefore support our hypothesis, regardless of the number of buckets in the hashmap: the difference between T-CuckooKF's performance and T-Chaining's performance increases as fullness increases.

We also consider our results from the 90% finds (5% inserts/erases) test to see if they support our hypothesis. Again, fullness seems to have a drastic impact on T-Chaining's performance, regardless of the number of buckets in the hashmap (performance drops by approximately 50%), and almost no impact on the performance of T-CuckooKF (Figure 5.2.6).

When the hashmap contains 10K buckets, T-CuckooKF's performance is approximately 1.2× that of T-Chaining at a fullness of 5. This difference increases with fullness: T-CuckooKF performs over 2× better than T-Chaining at a fullness of 15. We observe a similar phenomenon when the hashmap has 125K buckets, and when the hashmap has 1M buckets (Figure 5.2.6): T-CuckooKF's performance goes from approximately 2× the performance of T-Chaining at a fullness of 5, to over 3× at a fullness of 15.

This result is consistent with the claim made in Section 5.1.2 (i.e., that cuckoo hashmaps outperform chaining hashmaps when the average number of elements per bucket is high). Regardless of the number of buckets in the hashmap, T-Chaining experiences a much more drastic drop in performance than T-CuckooKF as fullness

increases, and the difference in performance also therefore increases.

SUPPORTED: T-CuckooKF outperforms T-Chaining by a greater margin as maximum fullness increases.

5.2.6 Hypothesis 4

An algorithm's cache usage most heavily affects performance on our benchmarks; abort rate is negligible and has little effect on performance. (Supported)

We pose this hypothesis to determine if the bottleneck in performance is the algorithm's cache usage or its abort rate. Our results for both tests discussed in Hypothesis 3 indicate that the number of cache misses is strongly correlated with performance. In general, the performance of all hashmaps decreases slightly (less than 20%) as hashmap size increases from 10K to 1M buckets. We attribute this to the increasing number of cache misses with increasing numbers of buckets.

The greater difference between T-CuckooKF's performance and T-Chaining's performance as fullness increases is also mirrored in the increased difference between T-CuckooKF's and T-Chaining's number of cache misses (discussed in Section 5.2.3). T-CuckooA's performance also drops more significantly than T-CuckooKF's as fullness increases (we observe, however, that T-CuckooA still performs at least as well as T-Chaining). T-CuckooA uses the same algorithm as T-CuckooKF, but has worse cache usage, which provides further evidence that cache usage plays a crucial part in performance.

Our results in Table 5.2.1 and in Appendix B.8 indicate that the abort rate remains relatively constant regardless of the fullness of the hashmap, but decreases as the number of buckets in the map increases. A greater number of buckets

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00130	0.00332	0.00765	0.01270	0.01802	0.02119
T-CuckooA	0.00035	0.00115	0.00216	0.00368	0.00558	0.00546
T-CuckooKF	0.00025	0.00077	0.00222	0.00398	0.00555	0.00677
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

10K Buckets

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00011	0.00034	0.00085	0.00126	0.00166	0.00212
T-CuckooA	0.00002	0.00009	0.00015	0.00028	0.00041	0.00056
T-CuckooKF	0.00002	0.00006	0.00013	0.00022	0.00037	0.00052
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

125K Buckets

Table 5.2.1: Hashmap Abort Rate (Maximum Fullness 10, 33% Finds/Inserts/Erases)

decreases the abort rates, since the probability that two threads will simultaneously attempt to read or modify the same bucket decreases. Abort rate is negligible in nearly all tests, with the chaining hashmap experiencing the highest abort rates (the median of 5 trials never exceeding 0.008%).

We see that cache performance affects overall performance more than the abort rate; as the number of buckets increases, performance drops even while the abort rate decreases. The major factor detracting from performance appears to be the significant increase in the number of cache misses.

SUPPORTED: The number of cache misses is strongly correlated with performance on our benchmarks, but abort rate is low and does not heavily affect performance.

5.2.7 Conclusion

Our evaluation demonstrates that the cuckoo hashmap does not experience crippling performance loss when integrated into a transactional setting, and that a scalable,

yet transactional cuckoo hashmap implementation exists. Furthermore, the relative behaviors of the transactional cuckoo and chaining hashmap mirrors the expected relative behaviors of their non-transactional counterparts: the transactional cuckoo hashmap outperforms the transaction chaining hashmap in scenarios in which cache performance is a bottleneck (high fullness), or when the time complexity and time to execute the operations is a bottleneck (when the hashmap can fit in cache).

We contrast our results with that of the strong transactional flat combining queue, and note that, unlike the strong flat combining queue, the cuckoo (and chaining) hashmaps scale in a transactional setting. We now look to the scalable commutativity rule, and the commutativity of our hashmap interface, to determine exactly why our transactional hashmaps can scale when our strong transactional queues cannot.

5.3 Commutativity of Transactional Hashmaps

In this section, we discuss the commutativity of non-transactional and transactional hashmap operations. We argue that synchronizing non-commutative transactions does not cripple the scalability of hashmaps implementing our hashmap interface, and that this occurs because the transactional setting adds few additional commutativity constraints to our hashmap interface. This is exemplified by our transactional chaining and cuckoo hashmap algorithms, which both scale in a transactional setting.

Our commutativity and scalability results with cuckoo and chaining hashmaps are in sharp contrast with our results from the strong flat combining queue. In Chapter 4, we saw how the flat combining technique relies on operation commutativity that is heavily reduced in a transactional setting, and that the

modifications to flat combining required to synchronize non-commuting transactions reduce the effectiveness of the flat combining technique and cripple its scalability. As we discover, the hashmap interface experiences only a slight loss of operation commutativity in a transactional setting; this allows transactional cuckoo and chaining hashing algorithms to scale.

In a non-transactional hashmap, insert and erase operations do not commute with finds, other inserts, or other erases when the argument keys are hashed to the same bucket, and at least one insert or erase out of the two operations succeeds. These scenarios are shown in Table 5.3.1. Both cuckoo and chaining hashmaps synchronize these non-commutative operations with per-bucket locks: an insert or erase in a chaining hashmap acquires a bucket-specific lock that protects access to the bucket during the duration of the insert or erase, while an insert or erase in a cuckoo hashmap locks both buckets in which the element may be placed. If the hashmap is large and the workload uses a large range of keys, it is very likely that two operations will commute because they will modify or read separate buckets. By the scalable commutativity rule, there exists a implementation of a hashmap algorithm that scales whenever insert or erase operations occur in separate buckets. Indeed, there are several examples of non-transactional chaining and cuckoo hashmap implementations that scale (with most reasonable hashmap workloads and a large enough map).

In a transactional setting, we have the added constraint that no two transactions commute if (a) any operation in one transaction observes that a key is present given one order of the transactions, but not the other; and/or (b) a key is present in the map immediately after both transactions commit in one ordering of the transactions, but not present in the map immediately after both transactions

Operations	H	H'
insert vs. erase	(T, M.insert(k,v), false) (T, M.erase(k), true)	(T, M.erase(k), true) (T, M.insert(k,v), true)
insert vs. find	(T, M.insert(k,v), true) (T, M.find(k,v), true)	(T, M.find(k,v), false) (T, M.insert(k,v), true)
erase vs. find	(T, M.erase(k), true) (T, M.find(k,v), false)	(T, M.find(k,v), true) (T, M.erase(k), true)
insert vs. insert	(T1, M.insert(k,v), true) (T2, M.insert(k,v), false)	(T2, M.insert(k,v), true) (T1, M.insert(k,v), false)
erase vs. erase	(T1, M.erase(k), true) (T2, M.erase(k), false)	(T2, M.erase(k), true) (T1, M.erase(k), false)

Table 5.3.1: Hashmap operations that do not commute. H is the original history; H' is the history with the order of the two operations exchanged. In the last two scenarios, note that it is important that the operations be performed by two different threads for the history to contain observably different results.

commit in the other ordering.

We can reason about commutativity in a transactional setting in terms of *sets* of buckets that overlap between two transactions, rather than individual buckets that are both modified or read by two operations. If an operation in a transaction modifies a bucket that is read or modified by an operation in another transaction, then the two transactions will not commute. Examples of larger transactions that fail to commute are shown in Table 5.3.2.

Once again, if the hashmap is large and the workload uses a large range of keys, it is very likely that two transactions will commute because they will modify or read separate sets of buckets. The intersection between the set of buckets modified in one transaction, and the set of buckets modified in the other, is likely to be empty or small (depending on the length of the transactions and the particular workload). We can therefore expect that an implementation of a cuckoo or chaining hashmap will still scale in a transactional setting in nearly all the same scenarios in

Example	H	H'
1.	(T1, START_TXN, ()) (T1, M.find(k1,v), false) (T1, M.insert(k1,v), true) (T1, M.insert(k2,v), true) (T1, COMMIT_TXN, ()) (T2, START_TXN, ()) (T2, M.find(k1,v), true) (T2, M.insert(k1,v), false) (T2, M.find(k2,v), true) (T2, COMMIT_TXN, ())	(T2, START_TXN, ()) (T2, M.find(k1,v), false) (T2, M.insert(k1,v), true) (T2, M.find(k2,v), false) (T2, COMMIT_TXN, ()) (T1, START_TXN, ()) (T1, M.find(k1,v), true) (T1, M.insert(k1,v), false) (T1, M.insert(k2,v), true) (T1, COMMIT_TXN, ())
2.	(T1, START_TXN, ()) (T1, M.erase(k1), false) (T1, M.erase(k2), true) (T1, M.insert(k2,v), true) (T1, COMMIT_TXN, ()) (T2, START_TXN, ()) (T2, M.insert(k1,v), true) (T2, M.insert(k2,v), false) (T2, COMMIT_TXN, ())	(T2, START_TXN, ()) (T2, M.insert(k1,v), true) (T2, M.insert(k2,v), false) (T2, COMMIT_TXN, ()) (T1, START_TXN, ()) (T1, M.erase(k1), true) (T1, M.erase(k2), true) (T1, M.insert(k2,v), true) (T1, COMMIT_TXN, ())

Table 5.3.2: Example hashmap transactions that do not commute. H is the original serial history; H' is the history with the order of the two transactions exchanged.

which individual operations commute.

To synchronize transactions that do not commute, both the cuckoo and chaining hashmaps must synchronize all buckets that are modified by both transactions, or that are modified by one transaction and observed by another. We note that the synchronization necessary to add transactional guarantees to the cuckoo hashmap is essentially equivalent to those necessary to add transactional guarantees to the chaining hashmap. The difference between the two synchronization mechanisms is that an operation that will require synchronizing one bucket in the chaining hashmap will instead require synchronizing two buckets in the cuckoo hashmap. Transactions are synchronized using the following method:

A transactional find in a cuckoo hashmap and a chaining hashmap adds a read of the bucket version(s) of the bucket(s) in which the key could be (or is) located. A transactional insert adds a read of the bucket version of the bucket in which the key is located only if the key is already present. A transactional erase adds a read of the bucket version(s) of the bucket(s) in which the key could be located if the key is absent, or a write of the bucket(s) if the key to erase is present. At commit time, any bucket versions in the read set are locked (therefore locking their corresponding buckets); the reads are then checked, and the writes installed.

We note that buckets are locked only at execution time, only at commit time, or at both commit and execution time, depending on which operation accesses the buckets. A transactional find, unsuccessful erase, or unsuccessful insert locks the appropriate buckets only at commit time. A successful transactional erase locks the appropriate buckets only at commit time, and a successful transactional insert locks the appropriate buckets only at execution time (because insertions are eager, a bucket into which an element is inserted does not need to be locked at commit time).

This algorithm allows for a implementation of a transactional hashmap that can scale in nearly all those scenarios in which an implementation of a non-transactional hashmap can scale. This is because the granularity of synchronization of the transactional hashmap is still per-bucket synchronization. The synchronization schemes of the cuckoo and chaining hashmaps rely on the fundamental assumption that operations performed on separate buckets will commute with each other. In a transactional setting, this assumption is still true for transactions whose operations modify or read disjoint sets of buckets. Because the scenarios that require extra synchronization to handle non-commuting transactions are rare, and these scenarios do not require synchronization beyond the level of buckets, both the transactional cuckoo and chaining hashmaps scale in nearly all

the same situations in which their non-transactional counterparts scale. This demonstrates that, unlike our queue operation interface, our hashmap operation interface (and the commutativity of operations in a transactional setting) allows for a scalable implementation of a transactional hashmap.

Furthermore, the cuckoo hashmap retains its non-transactional characteristics— better cache performance than the chaining hashmap, and asymptotic constant (or amortized constant) time bounds for its operations—even in the transactional setting. This is cuckoo hashing's fundamental aspects—hashing into multiple, fixed-size buckets, and cuckoo shuffling for inserts—are not modified when the cuckoo hashmap is made transactional. This result contrasts with our findings with the flat combining algorithm, which has to be fundamentally modified to support queue transactions.

6

Future Work and Conclusion

6.1 Future Work

One direction of future work is to specialize our data structures for singleton transactions. As we noted in Chapter 4, the commutativity between singleton transactions is equivalent to the commutativity between single operations. Special treatment of singleton transactions allows singletons to avoid the transactional overhead that is necessary to synchronize multi-operation transactions. However, singletons would need to be carefully handled if they interleave with multi-operation

transactions.

This work defines a class of concurrent data structure interfaces that do not suffer a high loss of commutativity in a transactional setting, and proposes that data structures implementing these interfaces will retain their scalability when modified to provide transactional guarantees. This claim can be tested further by implementing and benchmarking other data structures that fall into this class. Building these data structures will hopefully also result in additional transactional STO data structures that perform and scale well.

Another class of data structures interfaces defined by this work are those interfaces that suffer crippling performance and scalability loss in a transactional setting. This class includes, for example, the strong queue interface. For any concurrent data structure that falls into this class, alternative specifications for the data structure, such as the one we proposed for the queue, can be explored. These data structure specifications can be tuned to provide some useful guarantees in a transactional setting beyond that of a simple concurrent data structure, while still achieving high performance.

6.2 Conclusion

This thesis argues that retaining the performance benefits and scalability of highly-concurrent data structure algorithms within a transactional framework such as STO is contingent upon the amount of commutativity that is lost when transactions must be supported. The amount of commutativity between transactions determines the amount of independence between the synchronization strategy used by the highly-concurrent algorithm, and the transactional bookkeeping and mechanisms that must be added to provide transactional guarantees.

Our investigation into concurrent and transactional queue algorithms demonstrates that there is a large performance gap between our naively-concurrent transactional queues, and the best-performing non-transactional concurrent queue (the flat combining queue). However, the flat combining queue suffers crippling performance loss when moved into STO. This is because the flat combining technique relies on operation commutativity that is prohibited in a transactional setting; the fundamental principle of flat combining is that operations from different threads can be applied in an arbitrary order to the queue, which is no longer true when operations are performed within transactions. In order for flat combining to support transactions, it must be modified in ways that greatly reduce its effectiveness. By exploring an alternative queue specification that allows for greater operation commutativity in a transactional setting, we provide evidence that the effectiveness of the flat combining technique is dependent on operation commutativity.

As an example of the opposite phenomenon, in which a concurrent algorithm retains its performance benefits and scalability in a transactional setting, we look to cuckoo and chaining hashmap algorithms. Our hashmap interface experiences fewer added commutativity constraints in a transactional setting than does our strong queue interface. Consequently, both the transactional cuckoo and transactional chaining hashmaps retain their scalability. Furthermore, the beneficial properties of cuckoo hashmaps (such as good performance in a small hashmap with several values in each bucket) are present even in a transactional setting. This is because the cuckoo hashing synchronization algorithm can be implemented independently of the modifications necessary to support transactions. In other words, the lack of added commutativity constraints in the transactional setting means that the concurrent cuckoo hashmap algorithm can be made transactional without fundamentally

changing its behavior.

Our results provide a way to determine in advance whether a highly-concurrent, non-transactional data structure can be made transactional while still achieving high scalability and performance. By evaluating how much commutativity a particular data structure's interface loses when moving from a non-transactional to a transactional setting, we also evaluate the impact on the data structure's performance and scalability when it is modified to support transactions. This general method enables us to explain why and how different transactional data structures achieve the performance that they do. With this method, researchers can focus on data structure designs that have the potential for high performance in transactional settings, and avoid unfortunate surprises from data structures that are destined to underperform.

References

[1] Y. Afek, G. Korland, and E. Yanovsky. *Quasi-Linearizability: Relaxed Consistency for Improved Concurrency*, pages 395–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. Database Syst.*, 17(1):163–199, Mar. 1992.

[3] 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.

[4] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 56–67. ACM, 2007.

[5] 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.

[6] 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.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[8] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM sigplan notices*, volume 44, pages 155–165. ACM, 2009.

[9] A. Dragojević and T. Harris. Stm in the small: trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 1–14. ACM, 2012.

[10] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, pages 93–107, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] M. Gentili. Dynamically resizable cuckoo hashtable. https://github.com/mgentili/DRECHTs, 2014.

[12] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *ACM SIGPLAN Notices*, volume 50, pages 31–41. ACM, 2015.

[13] 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*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[14] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2nd edition, 2010.

[15] A. Hassan, R. Palmieri, and B. Ravindran. *On Developing Optimistic Transactional Lazy Set*, pages 437–452. Springer International Publishing, Cham, 2014.

[16] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. *SIGPLAN Not.*, 49(8):387–388, Feb. 2014.

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

[18] 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.

[19] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

[20] 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*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[21] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[22] 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.

[23] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 31:1–31:16, New York, NY, USA, 2016. ACM.

[24] R. D. (Intel). Web resources about intel® transactional synchronization extensions, Jan 2017.

[25] M. Khizhinsky. A c++ library of concurrent data structures. https://github.com/khizmax/libcds, 2015.

[26] H. F. Korth. Locking primitives in a database system. *J. ACM*, 30(1):55–79, Jan. 1983.

[27] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *ACM SIGPLAN Notices*, volume 46, pages 542–555. ACM, 2011.

[28] H. T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 116–126, New York, NY, USA, 1979. ACM.

[29] E. Ladan-mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[30] S. Lie. *Hardware support for unbounded transactional memory*. PhD thesis, Citeseer, 2004.

[31] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, Rochester, NY, USA, 1995.

[32] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Berkeley, CA, USA, 2014. USENIX Association.

[33] 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.

[34] K. A. Ross. Efficient hash probes on modern processors. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1297–1301. IEEE, 2007.

[35] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, Aug. 1984.

[36] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] A. Spiegelman, G. Golan-Gueta, and I. Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 682–696, New York, NY, USA, 2016. ACM.

[39] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, New York, NY, USA, 2001. ACM.

[40] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, Dec. 1988.

[41] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2013.

[42] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Larktm: Efficient, strongly atomic software transactional memory. Technical report, Technical Report OSUCISRC-11/12-TR17, Computer Science & Engineering, Ohio State University, 2012.

[43] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *Proceedings of the 2Nd International Workshop on Data Management on New Hardware*, DaMoN '06, New York, NY, USA, 2006. ACM.

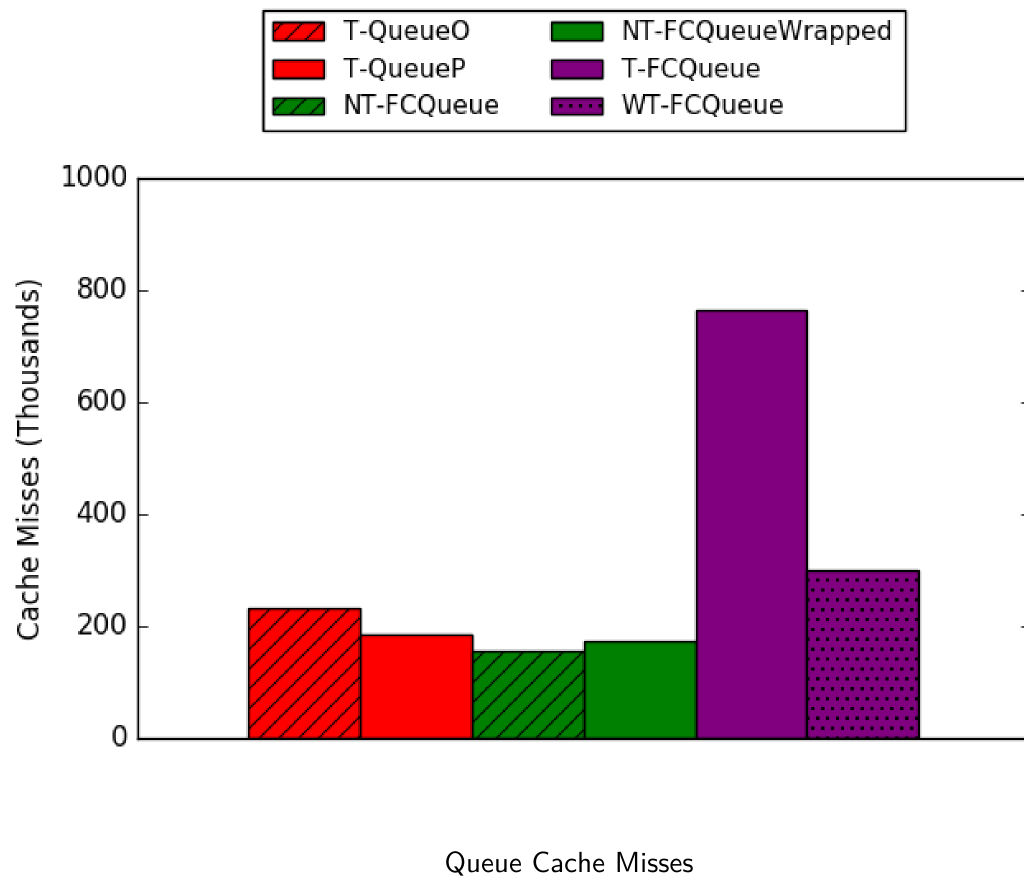
Appendices

Queue Results

All experiments are run on a 100GB DRAM machine with two 6-core Intel Xeon X5690 processors clocked at 3.47GHz. Hyperthreading is enabled in each processor, resulting in 24 available logical cores. The machine runs a 64-bit Linux 3.2.0 operating system, and all benchmarks and STO data structures are compiled with `g++-5.3`. In all tests, threads are pinned to cores, with at most one thread per logical core. In all performance graphs, we show the median of 5 consecutive runs with the minimum and maximum performance results represented as error bars.

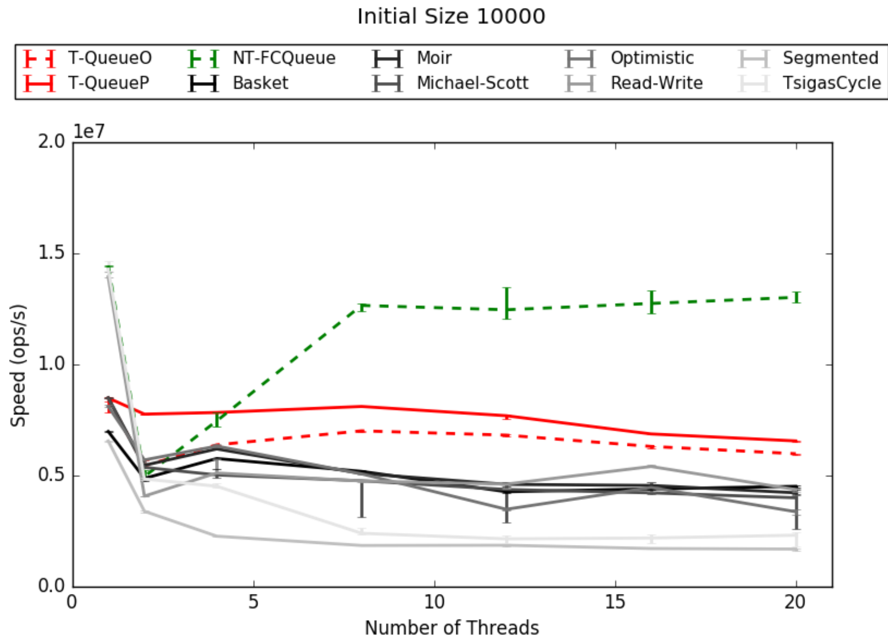
Cache misses are recorded by running the Multi-Thread Singletons Test benchmark with 8 threads, with each thread performing 10M transactions, under the profiling tool Performance Events for Linux (`perf`). The sampling period is set to 1000, meaning that every 1000th cache miss is recorded. We report the number of cache misses reported by perf (approximately 1/1000 of the actual number of cache misses).

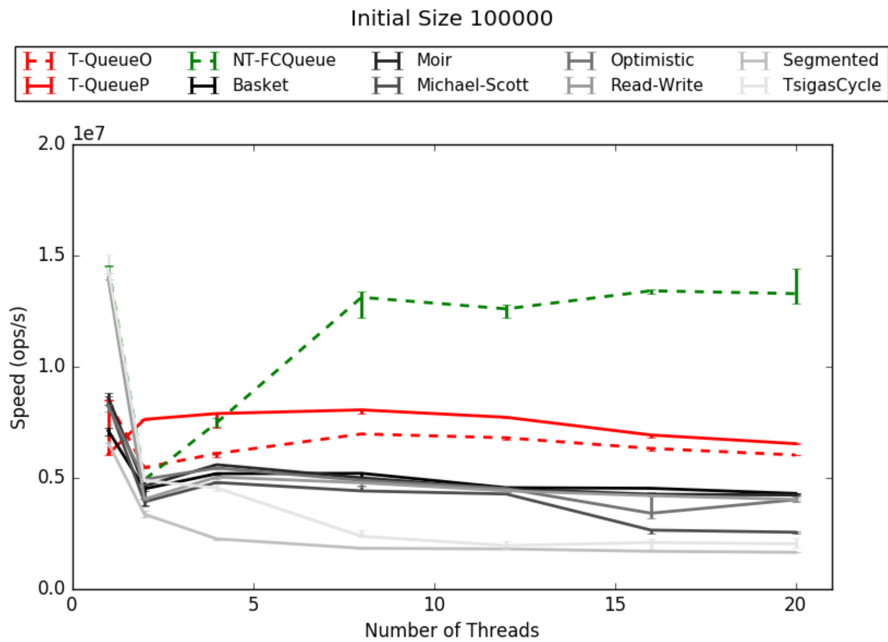
A.1 Cache Misses



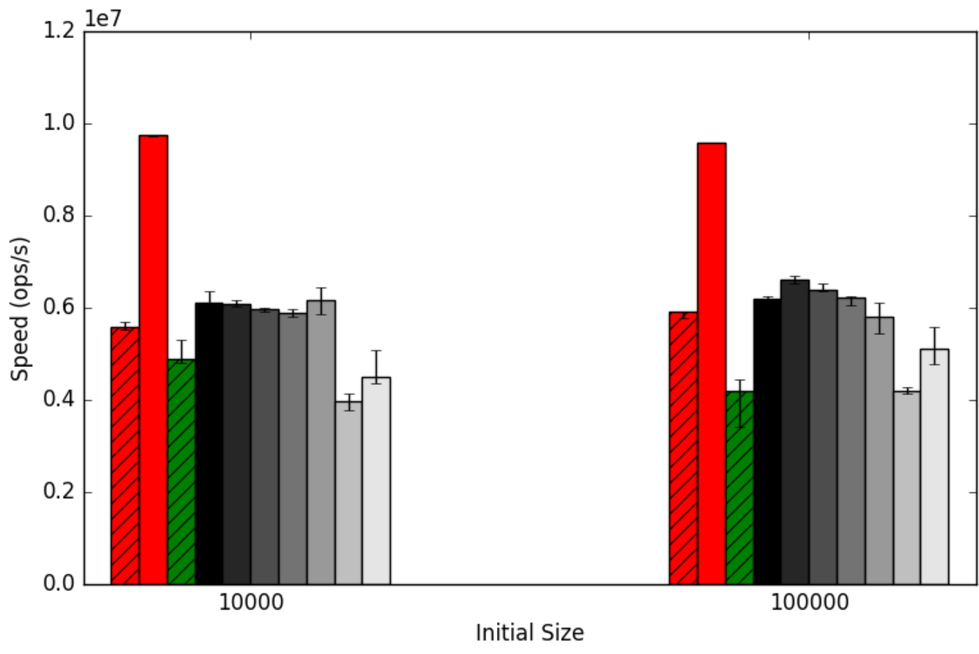
Queue Cache Misses

A.2 Performance of Non-transactional Concurrent Queues



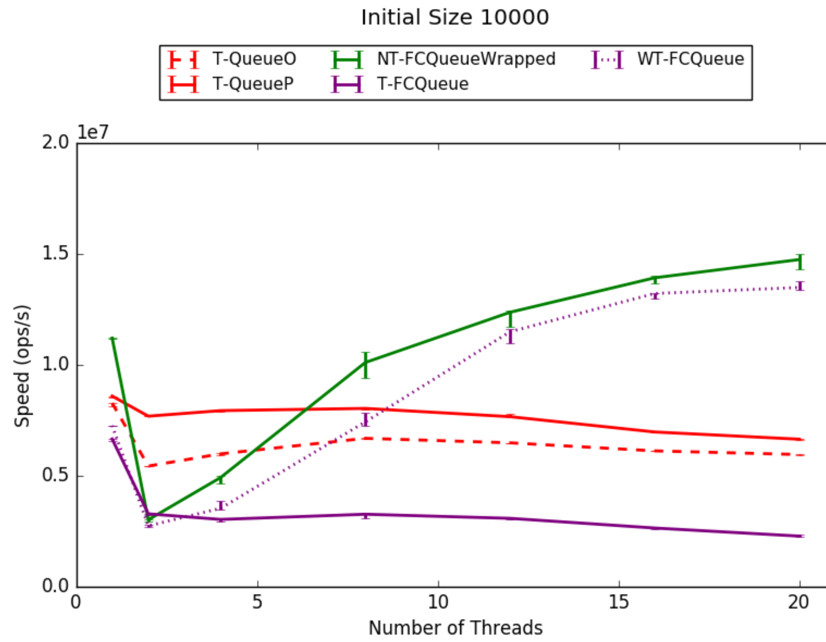


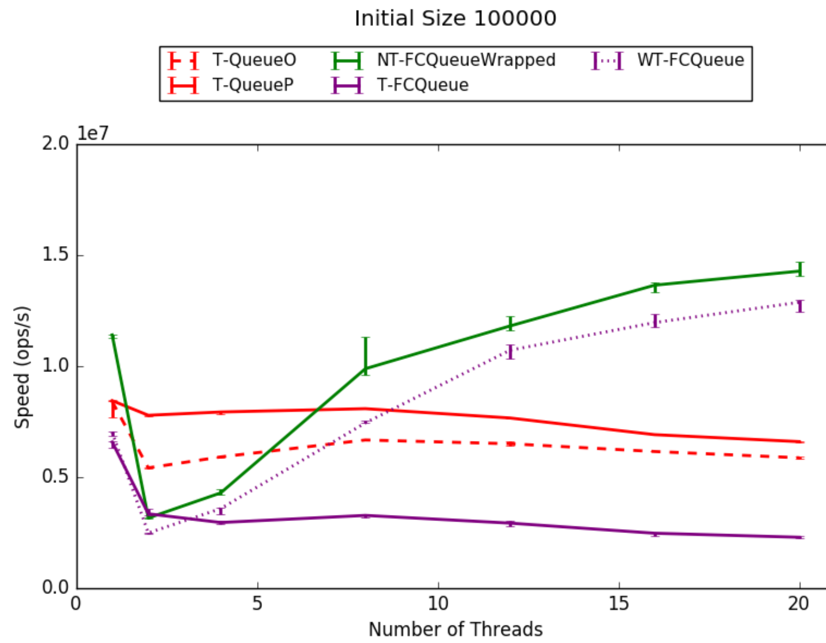
Multi-Thread Singletons Test



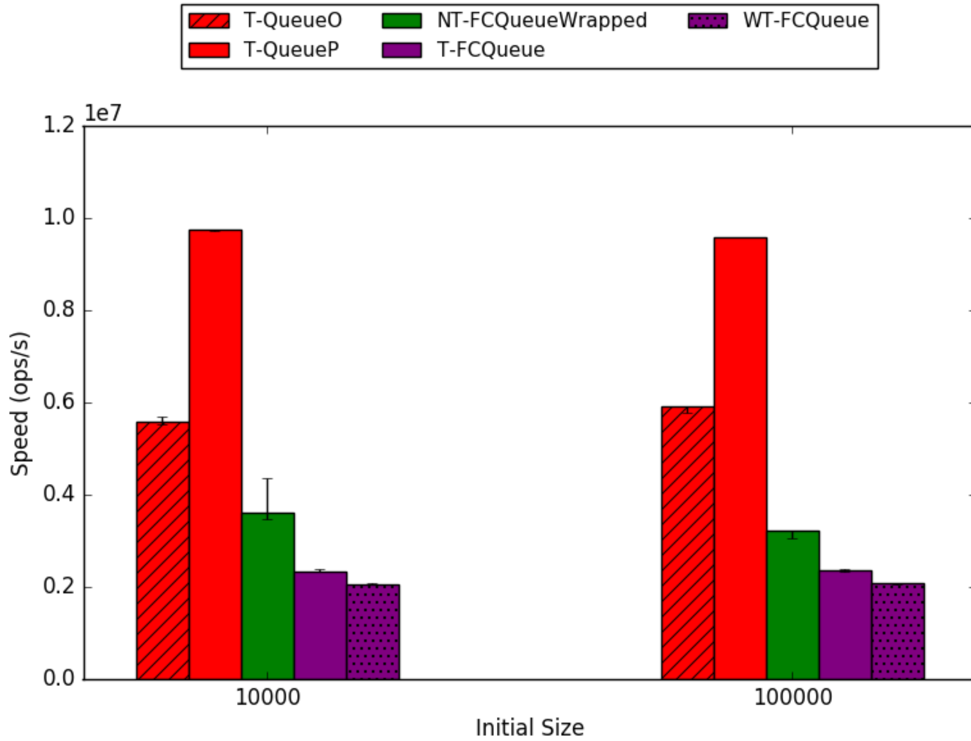
Push-Pop Test (2 threads)

A.3 Performance of Various Transactional Queues





Multi-Thread Singletons Test



Push-Pop Test (2 threads)

A.4 Push-Pop Test: Ratio of Pops to Pushes

Queue	Pops per 100 Pushes
T-QueueO	64
T-QueueP	28
T-FCQueue	57
NT-FCQueueWrapped	94
NT-FCQueue	110
Basket	42
Moir	62
Michael-Scott	54
Optimistic	50
Read-Write	84
Segmented	50
TsigasCycle	52

Ratio of pops to pushes when the push-only and pop-only threads are executing simultaneously

A.5 Abort Rate Results

Queue	Initial Size	
	10000	100000
T-QueueO	0.00001	0.00001
T-QueueP	1.54560	1.60886
T-FCQueue	0.62494	1.62620
WT-FCQueue	0.00000	0.00000

Push-Pop Test (2 threads)

Queue	#Threads					
	2	4	8	12	16	20
T-QueueO	7.51638	10.21210	7.62362	7.46284	6.71117	6.33827
T-QueueP	3.25357	2.84958	2.23814	2.26892	2.31778	2.32787
T-FCQueue	3.99773	6.08289	4.60842	4.78435	4.96070	5.33221
WT-FCQueue	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Multi-Thread Singletons Test: Initial Size 10000*

Queue	#Threads					
	2	4	8	12	16	20
T-QueueO	7.88767	10.67091	7.85545	7.54535	6.73918	6.48418
T-QueueP	3.07026	2.70427	2.12894	2.22341	2.22941	2.27197
T-FCQueue	4.05484	5.98340	4.51612	4.83815	5.08319	5.22679
WT-FCQueue	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Multi-Thread Singletons Test: Initial Size 100000*

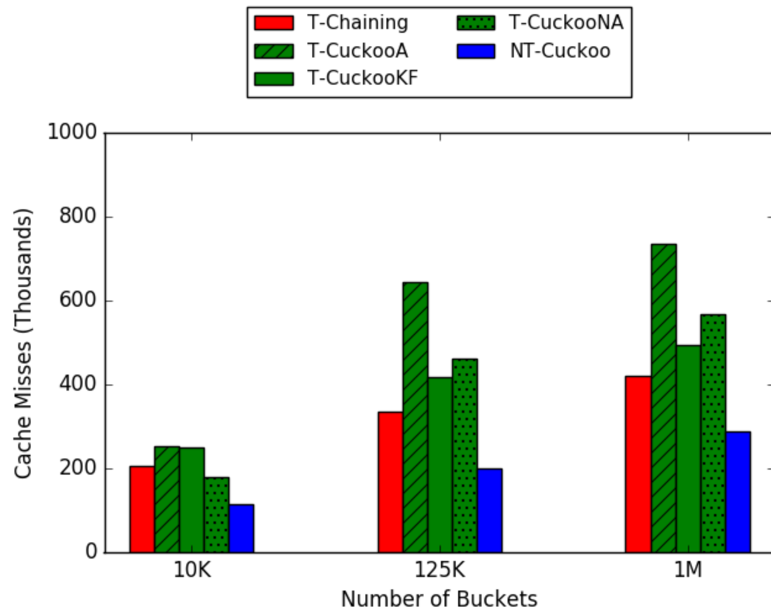
*We note that the abort rate appears to spike at 4 threads and decrease as the number of threads increases. One possible explanation may be that contention varies due to the spread of the threads among the CPU sockets.

B
Hashmap Results

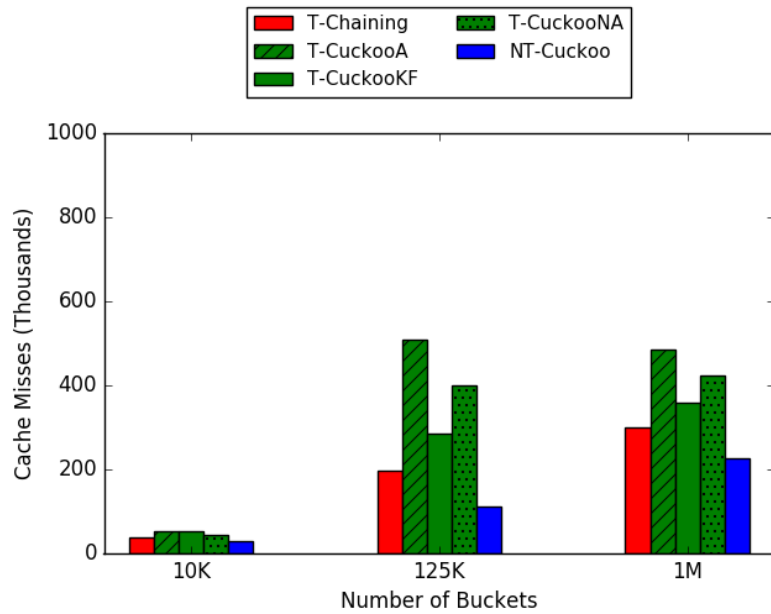
All experiments are run on a 100GB DRAM machine with two 6-core Intel Xeon X5690 processors clocked at 3.47GHz. Hyperthreading is enabled in each processor, resulting in 24 available logical cores. The machine runs a 64-bit Linux 3.2.0 operating system, and all benchmarks and STO data structures are compiled with `g++-5.3`. In all tests, threads are pinned to cores, with at most one thread per logical core. In all performance graphs, we show the median of 5 consecutive runs with the minimum and maximum performance results represented as error bars.

Cache misses are recorded by running the benchmark with 8 threads, with each thread performing 10M transactions, under the profiling tool Performance Events for Linux (`perf`). The sampling period is set to 1000, meaning that every 1000th cache miss is recorded. We report the number of cache misses reported by perf (approximately 1/1000 of the actual number of cache misses).

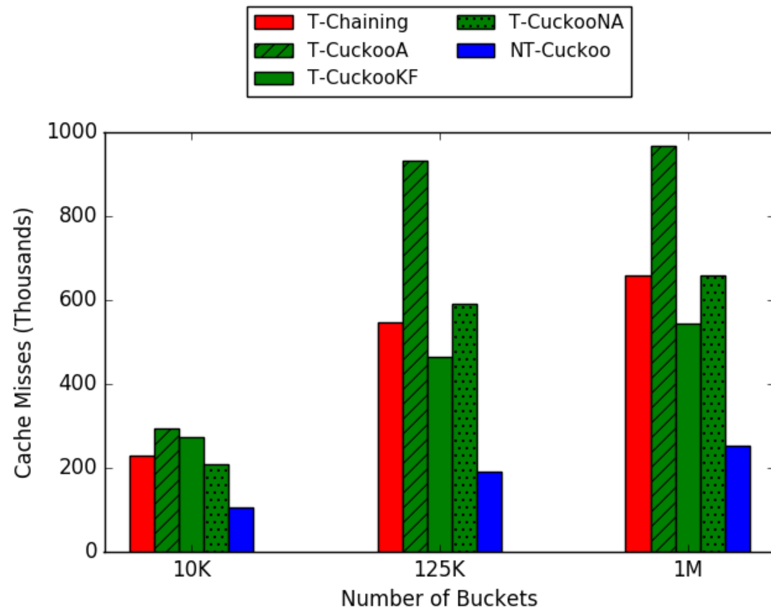
B.1 Cache Misses



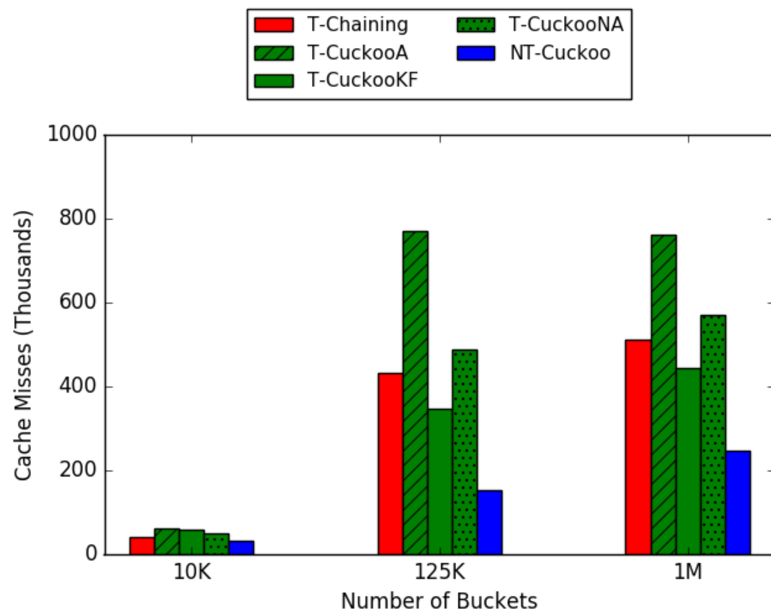
Max Fullness 5: 33%Find, 33%Insert, 33%Delete



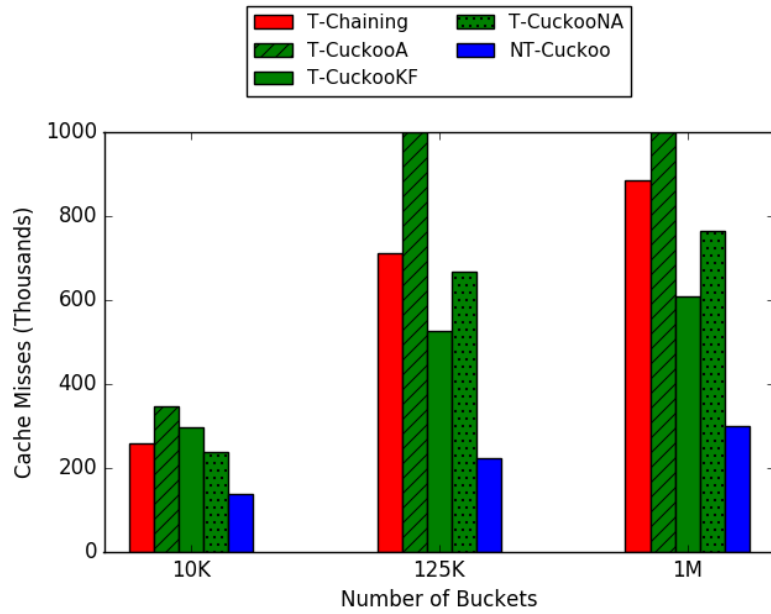
Max Fullness 5: 90%Find, 5%Insert, 5%Delete



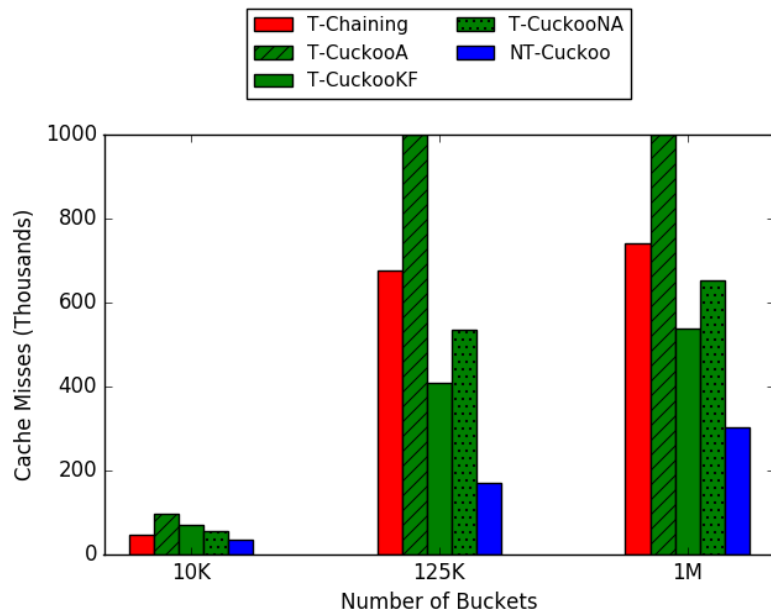
Max Fullness 10: 33%Find, 33%Insert, 33%Delete



Max Fullness 10: 90%Find, 5%Insert, 5%Delete

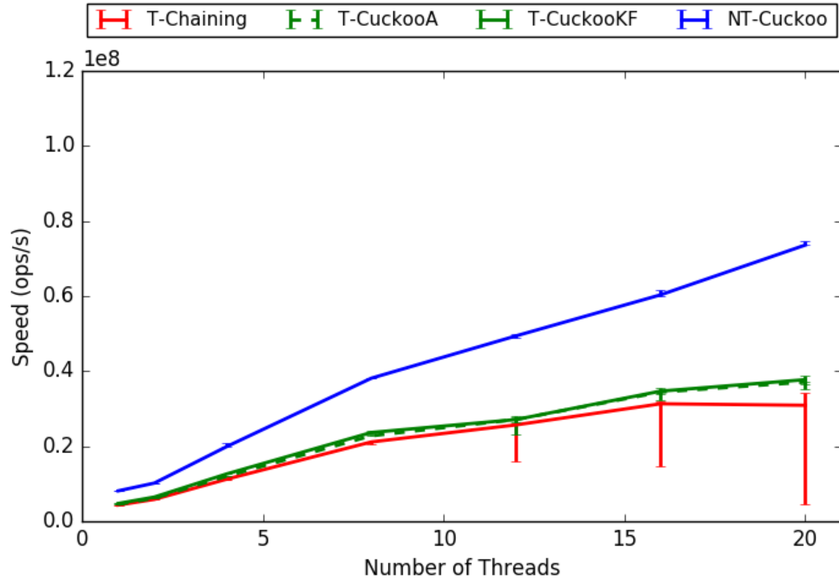


Max Fullness 15: 33%Find, 33%Insert, 33%Delete

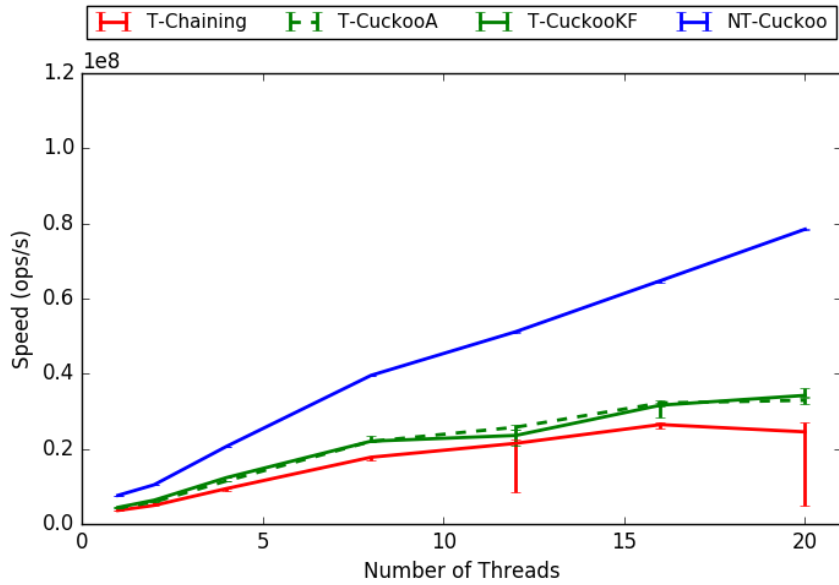


Max Fullness 15: 90%Find, 5%Insert, 5%Delete

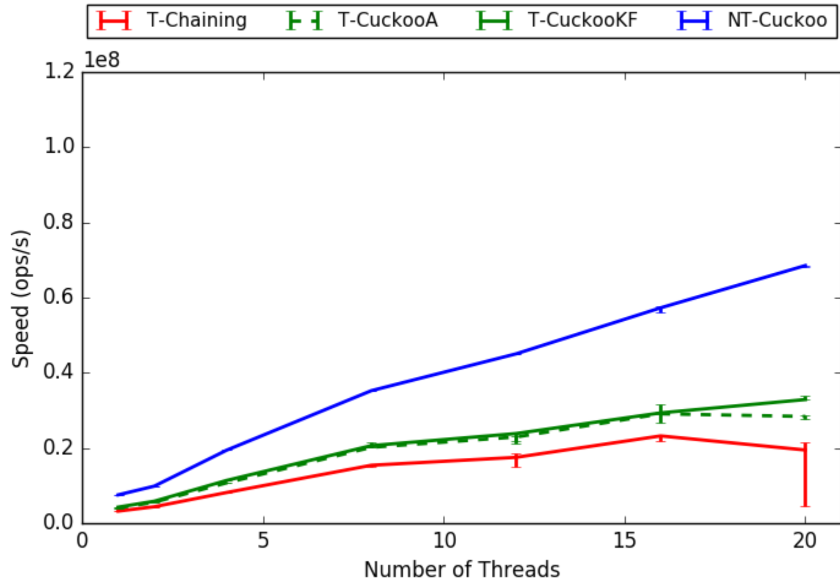
B.2 Performance: 10K Buckets, 33%Find, 33% Insert, 33% Erase



Maximum Fullness 5

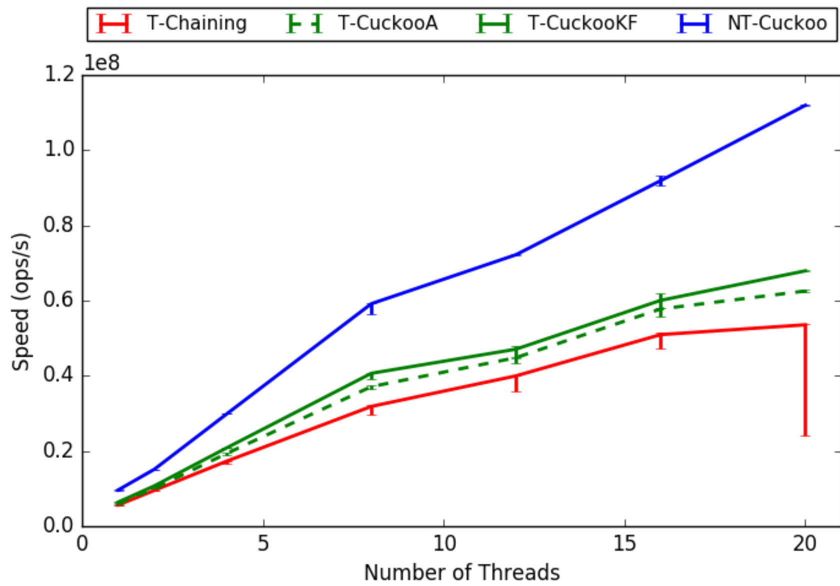


Maximum Fullness 10

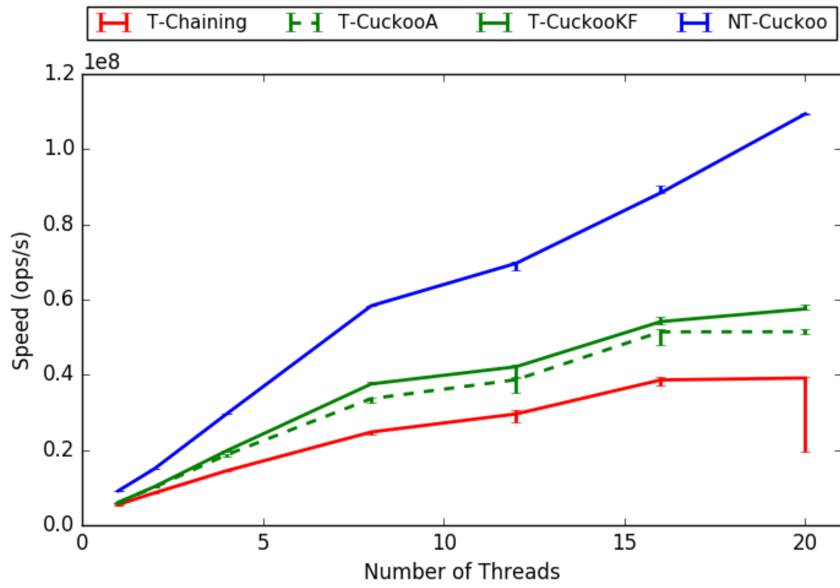


Maximum Fullness 15

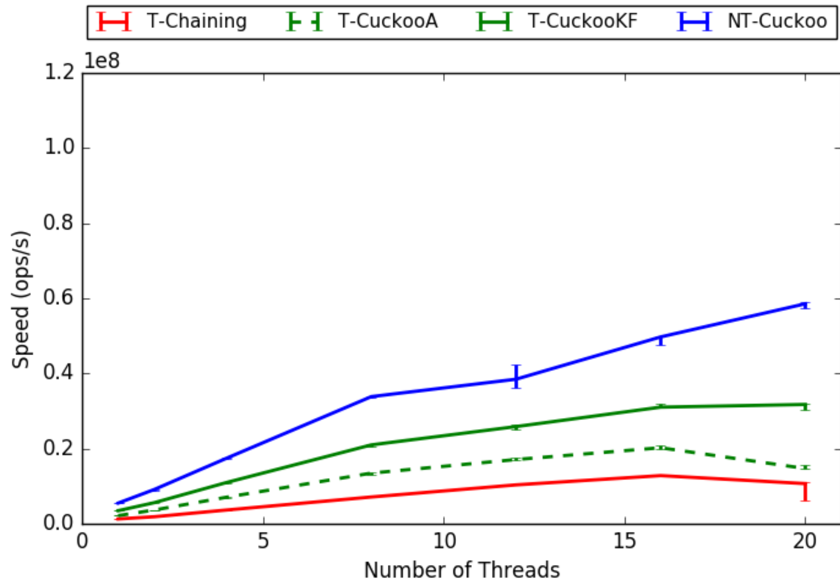
B.3 Performance: 10K Buckets, 90%Find, 5% Insert, 5% Erase



Maximum Fullness 5

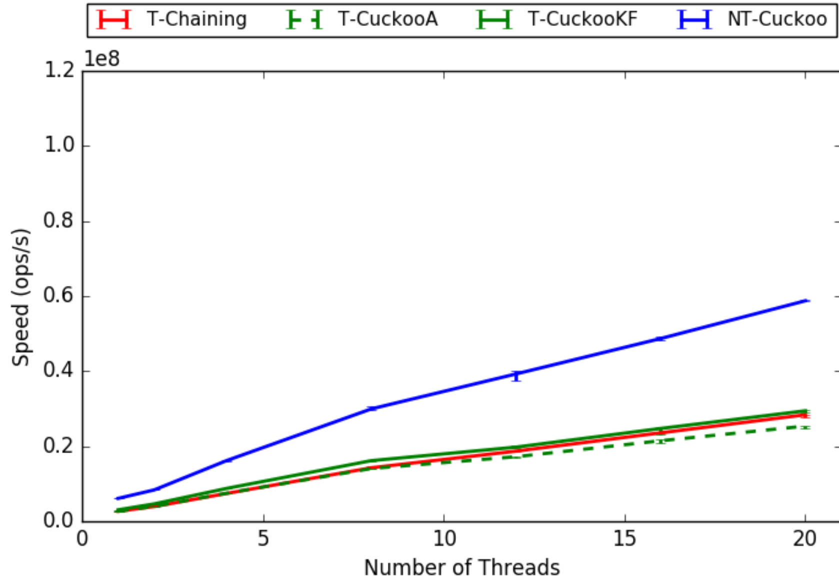


Maximum Fullness 10

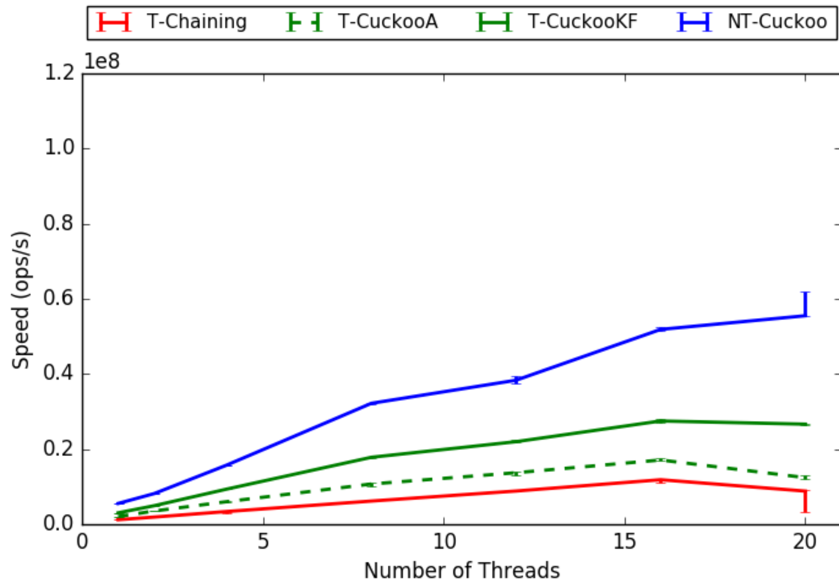


Maximum Fullness 15

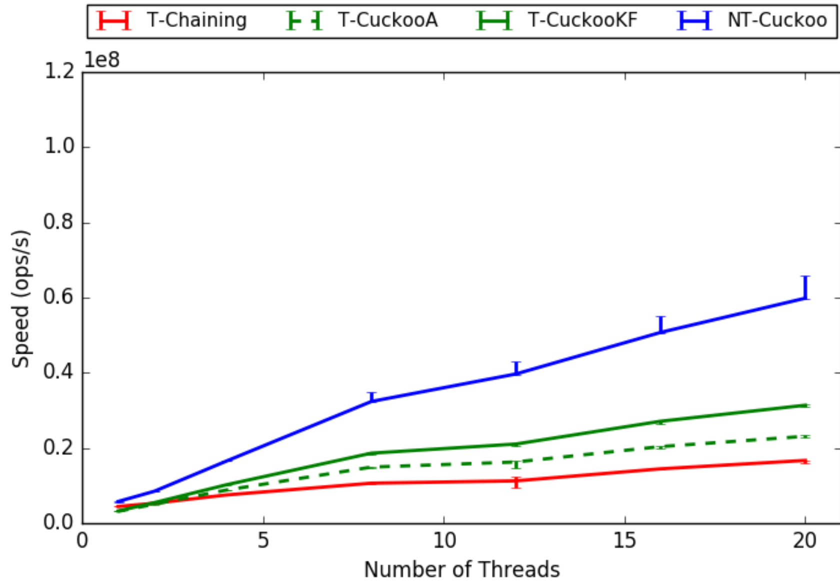
B.4 Performance: 125K Buckets, 33%Find, 33% Insert, 33% Erase



Maximum Fullness 5

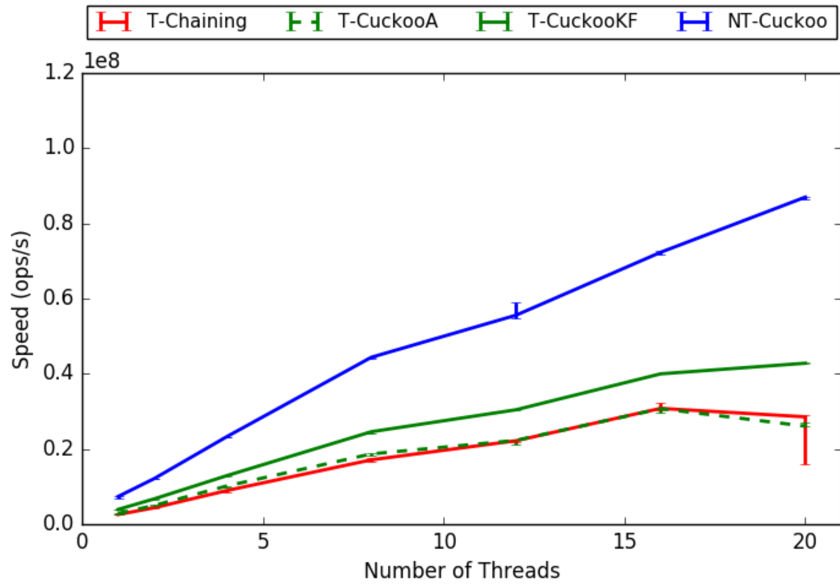


Maximum Fullness 10

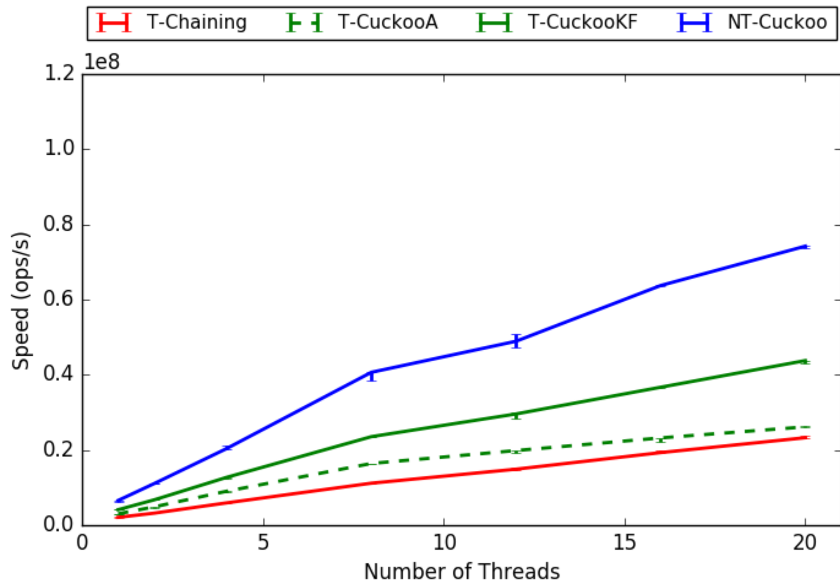


Maximum Fullness 15

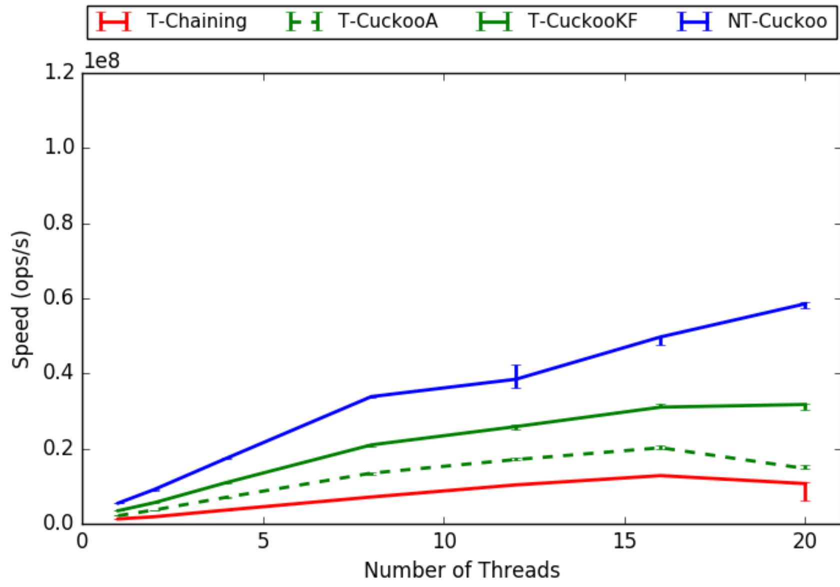
B.5 Performance: 125K Buckets, 90%Find, 5% Insert, 5% Erase



Maximum Fullness 5

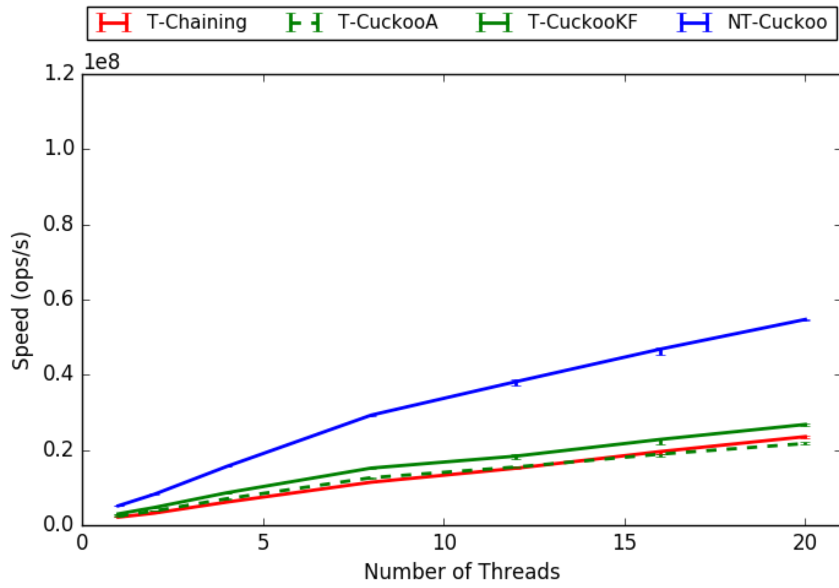


Maximum Fullness 10

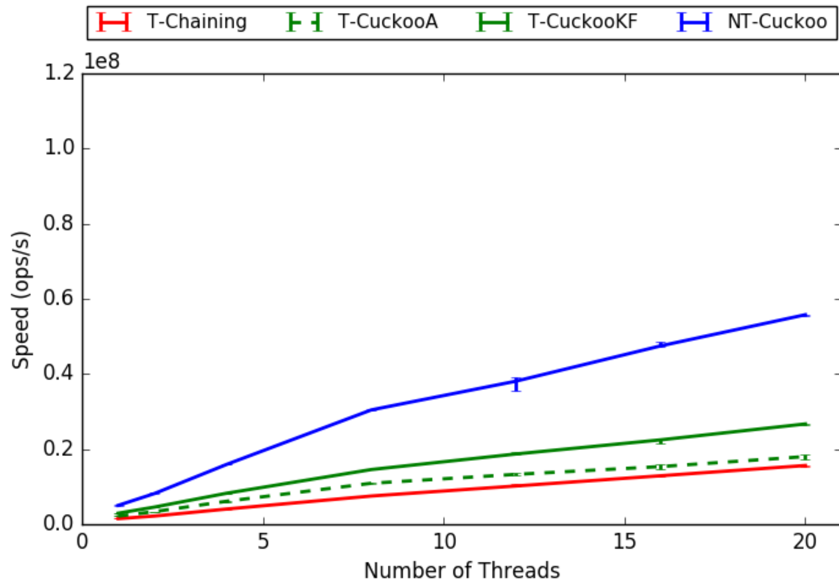


Maximum Fullness 15

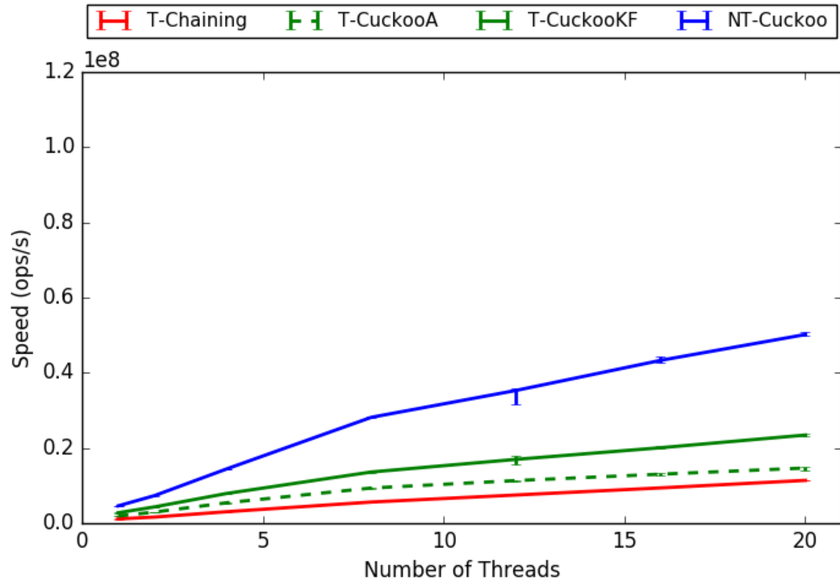
B.6 Performance: 1M Buckets, 33%Find, 33% Insert, 33% Erase



Maximum Fullness 5

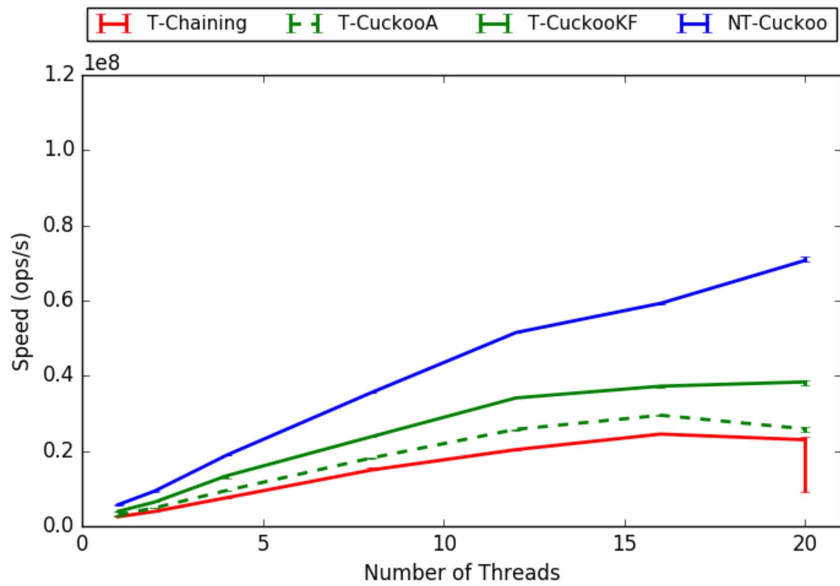


Maximum Fullness 10

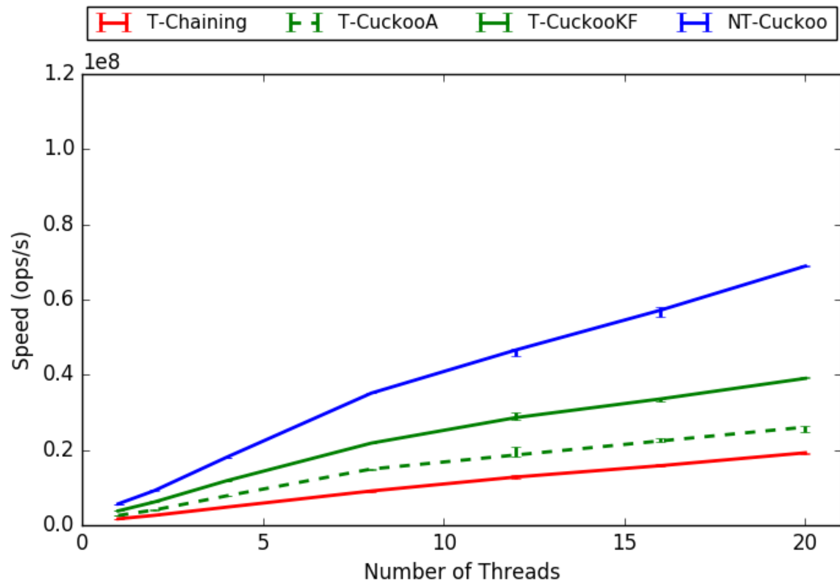


Maximum Fullness 15

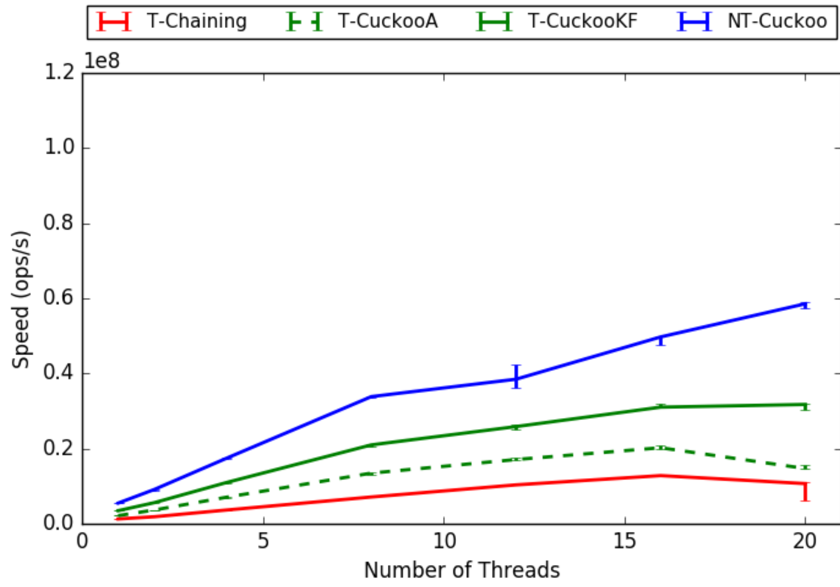
B.7 Performance: 1M Buckets, 90%Find, 5% Insert, 5% Erase



Maximum Fullness 5



Maximum Fullness 10



Maximum Fullness 15

B.8 Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00100	0.00325	0.00781	0.01291	0.01823	0.01843
T-CuckooA	0.00025	0.00108	0.00315	0.00487	0.00722	0.00773
T-CuckooKF	0.00025	0.00100	0.00286	0.00483	0.00664	0.00799
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00130	0.00332	0.00765	0.01270	0.01802	0.02119
T-CuckooA	0.00035	0.00115	0.00216	0.00368	0.00558	0.00546
T-CuckooKF	0.00025	0.00077	0.00222	0.00398	0.00555	0.00677
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00130	0.00375	0.00855	0.01460	0.01891	0.02273
T-CuckooA	0.00025	0.00097	0.00220	0.00389	0.00553	0.00528
T-CuckooKF	0.00020	0.00080	0.00185	0.00315	0.00473	0.00502
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

33F/33I/33E, 10K Buckets Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00030	0.00085	0.00187	0.00285	0.00367	0.00455
T-CuckooA	0.00005	0.00055	0.00089	0.00135	0.00162	0.00211
T-CuckooKF	0.00005	0.00038	0.00077	0.00118	0.00159	0.00215
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00050	0.00100	0.00229	0.00318	0.00409	0.00534
T-CuckooA	0.00020	0.00057	0.00106	0.00150	0.00178	0.00233
T-CuckooKF	0.00020	0.00057	0.00095	0.00140	0.00171	0.00243
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00030	0.00103	0.00266	0.00378	0.00449	0.00584
T-CuckooA	0.00025	0.00065	0.00115	0.00166	0.00205	0.00250
T-CuckooKF	0.00010	0.00068	0.00111	0.00163	0.00196	0.00262
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

90F/5I/5E, 10K Buckets Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00008	0.00028	0.00065	0.00101	0.00132	0.00174
T-CuckooA	0.00005	0.00014	0.00032	0.00051	0.00067	0.00091
T-CuckooKF	0.00002	0.00009	0.00026	0.00044	0.00065	0.00091
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00011	0.00034	0.00085	0.00126	0.00166	0.00212
T-CuckooA	0.00002	0.00009	0.00015	0.00028	0.00041	0.00056
T-CuckooKF	0.00002	0.00006	0.00013	0.00022	0.00037	0.00052
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00010	0.00047	0.00115	0.00163	0.00208	0.00267
T-CuckooA	0.00005	0.00015	0.00036	0.00056	0.00069	0.00084
T-CuckooKF	0.00005	0.00013	0.00028	0.00046	0.00052	0.00075
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

33F/33I/33E, 125K Buckets Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00001	0.00006	0.00016	0.00023	0.00030	0.00038
T-CuckooA	0.00001	0.00003	0.00008	0.00011	0.00021	0.00026
T-CuckooKF	0.00001	0.00003	0.00006	0.00011	0.00016	0.00026
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00003	0.00008	0.00016	0.00025	0.00032	0.00042
T-CuckooA	0.00001	0.00003	0.00009	0.00013	0.00018	0.00026
T-CuckooKF	0.00001	0.00002	0.00006	0.00009	0.00013	0.00020
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00004	0.00009	0.00020	0.00027	0.00037	0.00047
T-CuckooA	0.00001	0.00004	0.00009	0.00013	0.00018	0.00023
T-CuckooKF	0.00001	0.00002	0.00005	0.00007	0.00013	0.00018
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

90F/5I/5E, 125K Buckets Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00002	0.00004	0.00009	0.00012	0.00018	0.00022
T-CuckooA	0.00000	0.00001	0.00003	0.00004	0.00007	0.00010
T-CuckooKF	0.00000	0.00001	0.00002	0.00004	0.00006	0.00009
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00002	0.00004	0.00010	0.34599	0.24256	0.13496
T-CuckooA	0.00000	0.00001	0.00003	0.15493	0.10251	0.04467
T-CuckooKF	0.00000	0.00001	0.00002	0.09270	0.00520	0.06208
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00002	0.00005	0.14779	0.08940	0.00022	0.00027
T-CuckooA	0.00000	0.00001	0.03164	0.01825	0.00005	0.00008
T-CuckooKF	0.00000	0.00001	0.01399	0.00735	0.00005	0.00007
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

33F/33I/33E, 1M Buckets Abort Rate Results

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00000	0.00001	0.00003	0.00003	0.00004	0.00006
T-CuckooA	0.00000	0.00001	0.00001	0.00001	0.00002	0.00003
T-CuckooKF	0.00000	0.00000	0.00000	0.00001	0.00002	0.00003
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 5

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00001	0.00001	0.00003	0.00034	0.00074	0.00022
T-CuckooA	0.00000	0.00000	0.00001	0.00050	0.00035	0.00007
T-CuckooKF	0.00000	0.00000	0.00001	0.00026	0.00002	0.00003
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 10

Hashmap	#Threads					
	2	4	8	12	16	20
T-Chaining	0.00001	0.00001	0.00003	0.00636	0.00127	0.00016
T-CuckooA	0.00000	0.00000	0.00001	0.00001	0.00002	0.00037
T-CuckooKF	0.00000	0.00000	0.00001	0.00001	0.00002	0.00006
NT-Cuckoo	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Maximum Fullness 15

90F/5I/5E, 1M Buckets Abort Rate Results