Concurrency 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

Thesis advisor: Professor Eddie Kohler Lillian Tsai

Concurrency Algorithms in Transactional Data Structures

Abstract

This is my abstract.

Contents

List of Figures

List of Tables

This is the dedication.

Acknowledgments

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi commodo, ipsum sed pharetra gravida, orci magna rhoncus neque, id pulvinar odio lorem non turpis. Nullam sit amet enim. Suspendisse id velit vitae ligula volutpat condimentum. Aliquam erat volutpat. Sed quis velit. Nulla facilisi. Nulla libero. Vivamus pharetra posuere sapien. Nam consectetuer. Sed aliquam, nunc eget euismod ullamcorper, lectus nunc ullamcorper orci, fermentum bibendum enim nibh eget ipsum. Donec porttitor ligula eu dolor. Maecenas vitae nulla consequat libero cursus venenatis. Nam magna enim, accumsan eu, blandit sed, blandit a, eros.

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: ad-hoc 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—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 a separate, simultaneous transaction. Traditional word-STM tracks all words of memory read or written in a transaction, therefore incurring an enormous overhead[5]. Recent work at Harvard, however, has developed a novel type of STM, STO (Software Transactional Objects), that greatly improves upon the performance of traditional STM[22]. 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, increasing the overhead from transactional bookkeeping and introducing unnecessary conflicts: the transaction will abort if there is a concurrent update to the path, even though the result of the lookup is unaffected. STO allows datatypes to define datatype-specific abstract objects to track instead of memory words. This results in a reduced number of false conflicts and a tracking set that contains hundreds of times fewer objects than a word-STM.

1.2 Motivation

The focus of this work is to make STO as fast as the fastest concurrent programming patterns available, and when this is impossible, to precisely characterize why. Although STO outperforms traditional STM, STO's performance still falls far below that of other concurrent programming paradigms. STO's library of transactional

datatypes—datatypes exposing transactional operations—provide the interface by which programmers use to add transactional memory to their programs. Thus, a STO program is only as fast as its datatypes. The field of datatypes algorithms offers a natural point to focus our research: by defining the limits and potentials of transactional data structure algorithms, we learn how to maximize the performance of the STO system.

1.3 Overview

While the scope of our claims aims to be all the different datatypes supported by STO, this work focuses on a few core data structures: queues and hashmaps. We began by implementing the first version of these STO datatypes and, more broadly, developing design techniques for transactional data structures. These techniques defined 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 address our goal of performance maximization, 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 without the concern for transactional correctness. Thus, we benchmark the fastest concurrent datatypes currently available to set an upper bound for what performance STO data structures may reasonably hope to achieve. Benchmarks highlight which concurrent data structures are the highest-performing, and which parts of the STO data structure algorithms are bottlenecks and areas for improvement.

We first hypothesize that combining concurrent programming patterns with our transactional design patterns will produce transactional datatypes that greatly outperform our previous implementations. To evaluate our hypothesis, we take the fastest concurrent datatype algorithms and implement them within the STO framework. We discover that the algorithmic changes necessary to move highly-parallel data structures into STO results in a significant decrease in their performance; at times, they even underperformed the original STO data structure algorithms. Furthermore, our experience combining highly-concurrent, non-transactional algorithms with transactional ones leads to the conclusion that reasoning about scalability in transactional datatypes is inherently different than reasoning about scalability in traditional concurrent datatypes. In particular, reasoning about invariants regarding datatype state is essential to handle transactions: a transactional algorithm must maintain state across operations within a transaction. We formalize this argument as a commutativity argument, drawing on previous work regarding commutativity in transactional objects[38].

The stateful nature of transactions (and therefore reduced operation commutativity within transactions) leads us to revise our hypothesis; through experimental evidence, it seems clear that the reasoning that allows a high-concurrency data structure to achieve fast performance and scalability is intrinsically different than that required to create a transactional, highly performant data structures. We claim that certain high-concurrency algorithms for datatypes may be intrinsically non-transactional because the optimizations taken by these algorithm to achieve their high performances are incompatible with providing transactional guarantees. To evaluate our new hypothesis, we investigate how high-concurrency algorithms act when the transactional interface is modified. These interfaces allow us to better adapt concurrency datatypes algorithms for transactional settings and redefine the tradeoffs between the guarantees STO datatypes can provide and their performance.

This work as a whole is divided into 6 chapters. Chapter 2 describes background information on transactions and transactional memory as well as related work on transactional data structure algorithms. Chapter 3 is dedicated to discussing different queue algorithms, benchmarks, and performance results. Chapter 4 introduces a commutativity argument to claim that the highly-concurrent queue algorithms discussed in Chapter 3 are inherently non-transactional, and describes and evaluates an alternate transactional interface for the queue data structure to support this claim. Chapter 5 discusses the hashmap algorithms and results. Finally, Chapter 6 proposes future work and concludes.

2

Background and Related Work

2.1 Transactions

A transaction, or a group of operations, allows a programmer to more simply reason about concurrent access to shared state. The concept developed first in database theory and expanded to other domains with the development of hardware transactional memory in 1986 and software transactional memory (STM) in 1995. Transactions restrict how threads of execution can interact by providing certain guarantees: transactions are *serializable*, which means that one can find an ordering of committed transactions that satisfies the observed history of the execution. Operations within one transaction are never interleaved with operations in another transaction. Transactions are also *atomic*: if a transaction commits, all changes made by the transaction are instantly visible to other threads, and if a transaction aborts, no other thread sees any of the changes made by the transaction. Finally, transactions are *linearizable*[21]: 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.

An optimistic transactional system[14] enforces transactional guarantees in the following way: first, during a transaction's execution, the system tracks any intended changes to be made during the transaction in the transaction's *write set* and any state observed during the transaction in the transaction's *read set*. When the transaction attempts to commit, the system checks whether the observed values in the read set are still valid. If so, the system performs the changes in the transaction's write set and the transaction is marked committed. Else the transaction aborts and the system ensures that the transaction leaves no visible effects.

A pessimistic transactional system instruments every read and write such that if a conflict is detected when the transaction is executing a read or write prior to commit time, then the transaction either aborts or stalls until the conflicting transaction completes. Software transactional memory systems—the transactional systems concerned in this work—vary from fully pessimistic to fully optimistic.

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

2.2 Transactional Memory

Transactional memory[14][20] guarantees transactional properties in shared memory. TM can be implemented in both hardware and software. Although hardware transactional memory (HTM) naturally outperforms STMs, 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 cache used to track read/writes of the transaction is restricted in size. In addition, HTM lacks flexibility (granularity of reads/writes is at the word level) and fails to be truly composable[39]. Nevertheless, with the increasing support for HTM in computer hardware, many STMs, including STO, look to improve performance by integrating STM with HTM.

The STO system[22] used by this work is one of several developed software transactional memory (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, non-transactional API STMs which change the transactional API for performance gains, and abstraction-based STMs which track items based on abstract datatypes.

TL2[6] and LarkTM[40] 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.

Non-transactional APIs, such as open nesting[31], elastic transactions[10], transactional collection classes[4], early release[19], and SpecTM[9]), have also been heavily researched. These allow for programmers to reduce bookkeeping costs and false conflicts between transactions, but also require programmers to be experts in the system to implement such optimizations. This complicates, rather than simplifies, concurrent programming. For example, transactional collection classes remove unnecessary memory conflicts in data structures which are due to choice of implementation instead of semantic necessity; however, this requires designing multi-level, open-nested transactions, a much more complicated framework than an STM such as STO that allows data structures to be designed specifically to support transactions.

STO is one of several STM systems that use abstraction to improve STM performance [18][16][13][3]. These systems expose a transactional API to programmers in the form of concurrent, transactional data structures which are written on top of an STM. However, systems beside 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 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 that allow programmers to work with transactions.

2.3 Abstraction-based STMs

STO, the STM used in this work, tracks abstract operations of a transactional abstract datatype. 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 datatypes already implemented (ranging from queues to red-black trees), STO provides a transactional framework that allows programmers to easily add transactional support to other datatypes based on their particular semantics. STO allows datatypes to add datatype-specific *items* to their read or write sets, thereby reducing bookkeeping costs by exploiting the semantics of the particular datatype. Furthermore, datatypes can choose 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/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. The specifics of the commit protocol are implemented as datatype callbacks. This allows a datatype to use pessimistic strategies for certain operations (e.g. by needing to acquire a lock) while using optimistic strategies for other operations. STO is therefore a flexible hybrid of optimistic/pessimistic and eager/lazy versioning strategies.

Boosting[18] is a method to convert high-concurrency (non-transactional) data structures into transactional data structures. Like STO, boosting determines conflicts between transactions by relying on particular data structure semantics. Instead of allowing each datatype to define read- and write-set items, boosting maps a data structure'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 of the abstract locks must be acquired. Thus, the granularity of synchronization of the original linearizable data structure is only achievable in a boosted data structure for commutative operations. Because a transaction may abort, boosting practices undo logging and requires that each operation has an inverse. These constraints limit the applications of boosting to all data structures, but make it particularly useful for ones like sets. Unlike boosting, STO allows us to work with data structures whose operations have no inverse. Boosting is an inherently pessimistic strategy practicing 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 writes 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 not execute operations until commit. This adds support for operations that may not have an inverse. Optimistic boosting has not been shown to be effective in practice, and STO provides a more flexible hybrid transactional framework and achieves larger performance gains.

Automated locking[13] takes a similar approach to boosting by pessimistically acquiring abstract locks corresponding to each operation. It differs from boosting because it also takes a commutativity specification of conditions in which operations commute which an abstract datatype compiles into symbolic sets (locking modes) that are used to lock operations. These locking modes allow two commutative operations to run concurrently. Thus, this approach optimizes and automates the creation of abstract locks for an abstract data structure. A similar approach to automated locking may help STO data structures choose which abstract read/write items to track during a transaction to avoid false conflicts.

Predication[3] is a technique that maps an element to a predicate and uses an STM to manage access to the predicate. For example, an element in a set would be associated with an `in_set?` predicate and the STM would manage reads and writes of the predicate when elements are removed/added to the set. Unlike boosting, transactional predication achieves semantic conflict detection without keeping an undo log. However, predication must create predicates for absent as well as present elements, causing a garbage collecting problem that STO and other systems do not face. Transactional predication also focuses on making transactional data structures perform equally as well as high-concurrency 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[36], like STO, aims to produce libraries of transactional data structures. Similar to STO, TDSL allows for both pessimistic/optimistic and eager/lazy strategies and customizes strategies for each individual data structure. This allows for optimizations that rely on the 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 the example TDSL contains only a queue and skiplist, STO has implementations of many other data structures in transactional settings, including a queue, 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.

[1] This thesis investigates the integration of several highly-concurrent (non-transactional) data structures with STO. In particular, we test and modify different concurrent queue [29][7][27][1][37][17] and concurrent hashmap[34][28][11][32] algorithms, which we describe in more detail in Chapters 3 and **??**. Similar work has been done with lazy sets[15], in which transactional support is added to a lazy concurrent set. We extend their work to other data structures and investigate further into how to achieve performance close to highly concurrent (non-transactional) data structure.

[1]LYT: not sure wh

2.3.1 Commutativity in Transactional Data Structures

STO, along with the above methods of integrating concurrent abstract datatypes with STM, builds upon the ideas introduced by Weihl in the late 1980s[38]. Weihl defines optimal local atomicity properties of datatypes that allow for transactions to be serialized, therefore providing 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 implementation of these properties.

Schwarz and Spector[33] introduce a theory to order concurrent transactions based on the semantics of shared abstract datatypes and 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 specifies 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[24] that show when two operations commute (have no inter-dependencies), the order in which the operations are ordered do 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 properties[26]. Weaker ordering properties allow a datatype to fully exploit its operation semantics. 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 do not concern themselves with explicit concurrency algorithms for the queue. Our work in Chapter 4 with the weakly-transactional queue builds off this idea, and we provide experimental evidence that operation semantics can be usefully exploited only by providing weaker transactional guarantees.

Badrinath and Ramamritham[2] investigate operation commutativity to define recoverability, a weaker notion that commutativity that can achieve enhanced concurrency. Their key observation is that recoverability, unlike commutativity, takes into account the *removal* of operations from the execution history. If an operation is recoverable with respect to another transaction's operation, then the operation can be eagerly applied to the data structure without the other transaction's operation aborting or committing. This forces an ordering at commit time, but prevents issues such as cascading aborts: at least one of the transactions will be able to commit. Commutativity has also played a role in network consistency algorithms and CDRTs[35], as well as in the database community (where the idea of transactions originated and has been heavily researched).

Kulkarni, et.al.[25] 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.

Finally, commutativity work has played a large part in optimizing distributed transactions. Mu, et.al.[30] 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 and delay all transaction execution until commit time. When a transaction commits, 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.

3

FIFO Queue Algorithms and Analysis

This chapter investigates different concurrent and transactional algorithms for queues 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 unsuited to be converted for use as a fully transactional queue algorithm; the reasoning which allows for the optimizations taken by these algorithms directly conflicts with the reasoning necessary to provide transactional guarantees.

3.1 Transactional Queue Specification [1]

A concurrent queue must adhere to the following specification, in which all operations can be ordered such that:

- A value is popped off the queue only once (no duplicate pops).
- A value is pushed onto the queue only once (no duplicate pushes).
- 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 pops pop consecutive values in the queue starting from the head of the queue .
- Any two pushes push consecutive values at the tail of the queue.

To satisfy these invariants, transactional data structures must support *read-my-writes*. This is when a thread sees and modifies or returns the value from a previous operation in the transaction.

[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 those used for pop.

3.2 Naive Synchronization Queue Algorithms

STO provides two transactional FIFO queues that adhere to the interface exposed by the `C++` standard library queue. These transactional queue algorithms are designed with transactional correctness in mind, and concurrency as a secondary concern.

These two algorithms enforce transactional correctness using *versions*. A version can act as a lock on the data structure: in order to update the data structure, a thread must first lock the version. A version also tracks changes to the data structure because it monotonically increases when a thread modifies the data structure. Thus, any version seen by a thread is equivalent to some previous or current state of the data structure. The first instance of the version observed by a thread during a transaction is checked when the transaction commits. This ensures that all observations are valid. Note that we cannot update the read version to an instance of the version observed later in the transaction. This is because we need to validate the first time we see the version in the transaction.

3.2.1 T-Queue1

The T-Queue1 is the first implementation of the transactional queue data structure using STO's framework. It implements a circular, fixed-size transactional queue.

The queue is implemented using optimistic concurrency control (OCC), which is a transactional algorithm that optimistically assumes that no other transaction will conflict with a thread's transaction. No thread prevents another thread from operating simultaneously on the queue during a transaction, which means multiple threads can add read/writes of the same parts of the queue during their transactions. Contention only occurs during commit time, when the thread must necessarily lock the queue so that it can safely verify and modify the queue's values without parallel modifications by other threads. A thread can only realize that another thread has "beaten" it to modifying the queue at commit time.

There are two versions for the queue that can potentially invalidate a transaction if either changes: a headversion, and a tailversion. The headversion tracks the location of the head of the queue, and the tailversion tracks the location of the tail of the queue.

The queue supports three transactional operations: push, pop, and `front`. A push within a transaction adds to an internal `write_list_item`. This `write_list` holds thread-local list of values to be pushed onto the queue, which are added to the tail of the queue during commit time (ensuring all values are added consecutively). We observe that all transactions comprised of only pushes will always commit because pushes do not observe any property of the queue: one transaction's pushes and pops do not affect the outcome of another's pushes. (Note, however, that the opposite is not true). During commit time, the thread locks the tailversion so that no other thread's push can push onto the queue. The items on the `write_list` are added to the queue, and the tailversion incremented.

A pop within a transaction first checks if the queue is empty. If the queue is empty, then the thread reads the tailversion to ensure that no other transaction has committed a push before this thread commits. Suppose another thread successfully installs a push before this thread commits. Then this pop should have read that pushed value instead of seeing an empty queue. This forces the thread to abort the transaction. If there have been items added to the `write_list` from previous pushes within the transaction, then the pop will "pop" an item off the `write_list`, performing an instance of *read-my-writes*. Such a modiicatio is allowable during execution time because the `write_list` is thread-local. If the queue is nonempty, then the thread reads the headversion to ensure that no other transaction has committed a pop before this thread commits. The thread then finds the item to pop off the queue by iterating through the queue from the head until it finds an item that has not yet been popped off within this transaction. The thread adds a write to this item so it knows during commit time and during future pops that it intends to pop this item.

3.2.2 T-Queue2

The T-Queue2 is also a circular, fixed-size queue, with operations push, pop, and front. The T-Queue2 algorithm is a hybrid design integrating the T-Queue1 algorithm with another transactional algorithm: pessimistic locking. This takes inspiration from the transactional queue from the Transactional Data Structures Library[36] as described in previous work. Their pessimistic transactional queue appears to achieve better performance in their benchmarks than the T-Queue1, and the algorithm is simpler to implement and describe.

Pessimistic locking entails locking the queue when any naturally-contentious operation (e.g., pop) is invoked. The queue is then only unlocked after the transaction is complete. This ensures that no other thread will execute an operation that may invalidate a pop within this thread's transaction. However, operations such as "push" that can operate without any wait do not require locking during execution. Therefore, a push follows the same protocol as in the T-Queue1.

Because pop locks the queue, there are no conflicts at commit time. 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 queuelock.

3.3 Flat Combining Queue Algorithms

Given the relatively slow performance of our STO queues, we looked to find a highly-concurrent (non-transactional) queue algorithm that would be promising to integrate with STO's transactional framework. After running several benchmarks (see Figure **??**), 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 the STO queues.

3.3.1 Flat Combining Queue (Non-Transactional)

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 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 the queue) and achieves better cache performance by leveraging the single-threaded access patterns during data structure design.

The data structure design includes a sequential implementation of the data structure, a global lock, and per-thread records that are linked together in a global publication list. A record allows a thread 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 do an operation O:

1. T writes the opcode and parameters for O to its local record. Specifically for the queue, the thread writes `<PUSH, value>` or `<POP, 0>` to its local record.

2. T tries to acquire the global lock.

   (a) T acquires the lock and is now the "combiner" thread. T applies all thread requests in the publication list to the data structure in sequence, and writes both the result and an `<OK>` response to each requesting thread's local record.

   (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>`.

In the context of the queue, flat combining proves to be an effective technique to handle the contention caused by parallel access on the head and tail of the queue. In addition, their choice of queue implementation uses "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 the T-Queue1 and T-Queue2 suffer from the contention and cache performance issues pointed out in the flat combining paper, leading us to believe that flat combining's alternative synchronization paradigm might improve the performance of a transactional queue as much as it does for a concurrent queue.

3.3.2 Flat Combining Queue (Transactional)

Recall that, in addition to the requirements for a correct concurrent queue, a transaction queue must guarantee that there must be 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.

In order to add transactional guarantees to the flat combining queue, the order in which threads' requests are applied to the queue becomes important. For example, let a transaction in thread T1 be `pop, pop` and a transaction in thread T2 be pop. The combiner thread sees `T1:pop` and `T2:pop`, and applies both operations to the queue. T1 second pop request will violate the transactional specification because the two popped values in T1's transaction will not be consecutive. T1 must now abort, which means T2's pop is now invalid: it does not represent a pop at the head of the queue.

Addressing the scenario described above requires two important changes to flat combining (we describe the rational for these changes in Chapter 4):

1. pushes cannot be applied to the queue during a transaction's execution, and must instead be performed when a transaction commits
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 algorithm must track which values in the queue are going to be popped within the transaction. This prevents duplicate pops and detects if the queue will be "empty" given how values will be popped.
   (b) The algorithm does not execute flat combining requests from other threads until the transaction has committed or aborted. This can be implemented either through making other threads' transactions abort, or by causing the threads to block or spin.

We now describe the new algorithms for push and pop. We change the types of request 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, 0>`. 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 to be popped by this `thread_id`
- `<DEQ, thread_id>` : dequeue all values in the queue that are marked by this `thread_id`
- `<EMPTY?, thread_id>` : check if the queue, after popping all items marked by this `thread_id`, is empty
- `<CLEANUP, thread_id>` : unmark all values that are marked with this `thread_id`

As with the T-Queue1, a push within a transaction adds to an internal `write_list_item`. At commit time, the thread will invoke the `<PUSH, list>`, 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>` command. The combiner thread, upon seeing a MARK_POP command, 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.

If the value is not marked, the combiner thread marks the value with the caller's `thread_id` and returns `<OK>`. Else the value is marked by the calling thread's `thread_id`. Note that in this scenario, no other thread can have marked values in the queue, since they will abort when seeing the head value marked by the calling thread's `thread_id`. The combiner thread iterates sequentially through the queue values 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 removes the popped value from the queue.

If the queue is either empty or there are no values not marked with the caller's `thread_id`, the 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.

Note that this algorithm does not allow pops to read the values pushed within the same transaction. To do so would require passing in the thread's `write_list` in addition to the `thread_id` as arguments to the combiner thread. During our evaluation, we leave this part of the transactional queue specification unimplemented (and expect that adding this will only decrease performance).

The `<EMPTY?, thread_id>` request is posted when a thread attempts to commit a transaction that observed an empty queue at some point in its execution. This happens when the thread receives an `<EMPTY>` response to a `<MARK_POP>` request during the transaction. If the response to `<EMPTY?>` is true, then the thread knows that no other thread has pushed onto the queue between the time of its `<MARK_POP>` seeing an empty queue and commit time. Else another thread has pushed items onto the queue, invalidating this thread's pop result, and this thread must abort.

If a thread ever sees an empty queue when executing a pop *and* subsequently performs a push within the same transaction, the thread must prevent another transaction from committing between the time of the empty check and the installation of its pushed value. This requires adding what is essentially a lock of the tail of the queue. This is implemented via additional machinery in the combiner thread, which signals whether or not a transaction has locked the queue, and prevents any other thread's pushes from being installed until the "lock" is released.[2] [2]LYT: TODO ACT

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

3.4 Evaluation

3.4.1 Microbenchmarks

All queues are evaluated on a set of microbenchmarks to demonstrate their scalability and performance. The controlled nature of these microbenchmarks allow us to easily 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 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. This is because the benchmark tests do not manipulate the values they push/pop and the queue algorithms are agnostic to the actual values being placed in the queue.
- Initial Queue Size: We run several tests with different initial fullness of the data structure. 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.
- Transaction Size: We modify the number of operations per transaction in different benchmarks. For some benchmarks, the number of operations in a transaction is set to 1 (i.e., the transactions are singleton transactions). This provides a more fair evaluation of transactional data structures against concurrent data structures: by keeping a transaction as short as possible, we minimize the performance hit from transactional overhead. In order to support multiple-operation transactions, STO adds overhead which includes support for multiple items in read/write sets, read-my-writes, and an increased number of aborts and retries. With single operation transactions, we observe an upper bound on the best performance our data structures can achieve.
- Data Structure Opacity: If opacity is enabled, a transaction will abort immediately if any inconsistent state is detected. This requires keeping track of a global transaction ID (TID). This global TID must be accessed when a transaction commits and when items are added to the read set during a transaction's, making transactions overall more expensive. For our benchmarks, we consider only queues without opacity enabled (as to measure its maximum achievable performance).[3]

[3]LYT: do we need

Tests

1. 2-Thread Push-Pop Test: This test has one thread that performs only pushes and another thread that performs only pops. Unless the queue is empty, the two threads should never be modifying the same part of the data structure, and will never conflict (abort rate should be near 0). We use this test to measure the speed of push/pops on the queue when contention is not an issue. We expect that our transactional queues should perform as well, if not better, than most of the high-concurrency queues: while their algorithms are optimized for multi-threaded access, our simpler implementation should be just as fast with low contention and low abort rates.
2. 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. This test is run with different initial queue sizes and different numbers of threads, which each thread performing singleton transactions. This test allows us to benchmark performance under variable amounts of contention (by increasing the number of threads) and increased abort rates. We expect that our T-Queue1/T-Queue2 transactional queues will perform significantly worse once the number of threads is increased and our naive concurrency algorithms underperform concurrency algorithms optimized for contentious situations.

3.4.2 Results and Discussion

We describe the results of these benchmarks for different sets of queues. We first compare the T-Queue1 and T-Queue2, then a set of high-concurrency, non-transactional queues, and lastly different variants of the flat-combining queue (transactional and non-transactional). The latter two sets are measured along with our T-Queue2 as a baseline reference.

We refer to the several variants of the flat combining queue that we benchmark as follows:

- NT-FCQueue: the non-transactional flat combining queue.
- T-FCQueue: the fully-transactional flat-combining queue.
- NT-FCQueueWrapped: the non-transactional flat combining queue that invokes STO `start_transaction` and `commit_transaction` calls, but does not do any of the transactional bookkeeping necessary to provide transactional guarantees.

All results for the Push-Pop Test can be found in Appendix A.2, and all results for the Multi-Thread Singletons Test can be found in Appendix A.3. Within this discussion, we reference particular figures that provide more detailed statistics and results than we present in words here.

T-Queue1 vs. T-Queue2

The comparative performance of the T-Queue1 and T-Queue2 (Figure A.3.2 and Figure A.2.2) measures the effectiveness of a pessimistic approach to the pop operation. We use the better performing queue, the T-Queue2, as the baseline reference in all future benchmarks. This also verifies that a pessimistic approach to contentious operations (such as pop) benefits performance.

Concurrent, Non-transactional Queues

We benchmark a set of the best-performing high-concurrency queue algorithms against the better performing STO queue implementation, the T-Queue2. This helps determine which high-concurrency algorithm would be best suited to integration with STO. We look for both the most scalable and the highest-performing queue.

Our implementation of the flat combining queue modifies the implementation of the flat combining queue from the authors of the flat combining paper[17]. Our implementation of the other high-concurrency queues are taken from the Concurrent Data Structures (CDS) library implementations online[23]. The performance of these implementations on our tests the performance results given in the papers that originally proposed the algorithms.[4] [4]LYT: CHECK

Results for the Push-Pop Test can be found in Figure A.2.1, and results for the Multi-Thread Singletons Test can be found in Figure **??**.

Out of the concurrent data structures tested, the Moir queue[7] consistently performs best on the 2-Thread Push-Pop Test. On the Multi-Thread Singletons Test, the flat combining queue achieves performance over 2.5× greater than any other concurrent, non-transactional queue as the number of threads increases above 2.

The T-Queue2 outperforms all queues on the 2-thread Push-Pop test. This test incurs the least contention and transactional overhead to track simply how fast the data structure can handle pushes and pops. It is unsurprising that, on this test, a simple synchronization strategy, such as that used in the STO queues, outperforms the majority of high-concurrency algorithms which are optimized for scalability. The Multi-Thread Singletons Test shows the performance benefits of the flat combining queue: as contention and transactional overhead (abort rate) increases, the flat combining queue reaches performance approximately 1.75× greater than that of the STO queues. In addition, the flat combinig queue is the only queue that scales. All the other high-concurrency algorithms perform worse than our T-Queue2 regardless of number of threads accessing the queue or initial queue size.

By benchmarking transactional queues with naive concurrency algorithms (T-Queue1 and T-Queue2) against various high-concurrency algorithms, we demonstrate that a simple implementation of a naive algorithm can consistently outperform more complex concurrent queue implementations even given the additional STO overhead. This indicates that the overhead added from STO does not cripple performance if used carefully—our transactional data structures can compete with several high-concurrency, non-transactional data structures. However, we see by comparing to the non-transactional flat combining queue that our algorithms 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 focus our work on the flat combining queue.

NT-FCQueueWrapped Performance

The relative performance of the NT-FCQueueWrapped to the NT-FCQueue indicates how much of the overhead added by the STO system is unavoidable (without modifications to STO itself). The STO wrapper calls (`start_transaction` and `commit_transaction`) allow a thread to mark which operations should occur together in the same transaction. After invoking the `start_transaction` call, the thread can collect items in its read- and write-sets; when `commit_transaction` is invoked, the commit procedure is run (validation and installation of items in the read- and write-sets). The NT-FCQueueWrapped adds no items to the read- and write-sets after invoking `start_transaction`, and thus incurs the minimum amount of overhead necessary to use STO: the commit procedure has zero items to validate or install. The NT-FCQueueWrapped therefore represents the upper bound on what performance we can expect from a fully transactional flat combining queue, the T-FCQueue. Results are shown in Figure A.2.3 and Figure A.3.3.

The STO wrapper calls can lead to a loss of performance ranging from 0% at twenty threads to 40% at four threads (comparing against the performance of the vanilla non-transactional flat combining queue). Once contention increases and becomes the bottleneck, the difference in performance becomes negligible. The NT-FCQueueWrapped scales nearly equally as well as the NT-FCQueue, and matches performance at twelve threads when run with a smaller initial queue size. This comparison of the NT-FCQueueWrapped and the NT-FCQueue demonstrates that STO introduces negligible necessary overhead. Even with the wrapper calls, our results indicate it can still be possible to achieve performance up to approximately 1.75× greater than that of our original T-Queue1 and T-Queue2 as the number of threads increases.

T-FCQueue Performance

We compare the T-FCQueue against the NT-FCQueueWrapped and the T-Queue2 to measure how the flat combining transactional approach described in Section 3.3.2 performs.

In the Push-Pop Test (Figure A.2.4, the T-Queue2 outperforms both flat combining variants, an unsurprising result given our results from the concurrent queues benchmark in Figure A.3.3. We also note that only the T-Queue2 experiences aborts (at approximately 1.2% abort rates). We hypothesize this is due to a "no-starvation" aspect of the flat combining algorithm: the T-Queue2's pop or push operations acquire a global queue lock. This means that the push-only or pop-only thread may continuously succeed in acquiring the lock, leading to a large sequence of pops or pushes. This can lead to the queue reaching an empty state more often, which is the only state that can cause commit-time checks to fail and the transaction to abort. The flat combining algorithm, however, applies the operations of both requesting threads during one combiner pass. Since one thread only pushes and the other only pops, no thread aborts due to seeing a marked pop, and the queue rarely reaches an empty state since both one push and one pop are applied during each pass.

The Multi-Threaded Singletons Test (Figure A.3.4 shows the T-Queue2 performing approximately 2× better than the T-FCQueue, regardless of initial queue size. Both queues do not scale, and the performance ratio remains constant regardless of the number of threads. The T-FCQueue also experiences abort rates 1.5-2× of that of the T-Queue2 as the number of threads increase. Analysis with the `perf` tool indicates that the majority of the overhead is incurred 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 nearly 7× greater. We see these results because of two reasons:

1. *Higher Quantity*: 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*: each flat combining call requires executing instructions, which makes each operation request more expensive.

We conclude that the flat combining technique, while perhaps near-optimal for a highly-concurrent data structure, is no better in a transactional setting than a naive synchronization technique such as that used in the T-Queue1 and T-Queue2. This is because the flat combining algorithm must track the transaction state (e.g., going to perform two pops, one of which observes an empty queue) in order to provide transactional guarantees. This requires modifying the flat combining algorithm itself, reducing the performance benefits from the algorithm's optimizations. In the next chapter, we formalize this argument using a commutativity discussion and claim that the higher quantity of more complex flat combining calls is necessary for flat combining to be used in a transactional setting: the flat combining technique depends on operation commutativity present in only a non-transactional setting to achieve its high performance.

4

Commutativity and Weakly Transactional Queues

In this chapter, we first describe the commutativity of queue operations in high-concurrent, non-transactional settings and compare it to the commutativity of queue operations in a fully-transactional setting. We then argue that the flat combining technique, while perhaps near-optimal for a highly-concurrent data structure, is no better for performance than a naive synchronization technique in a transactional data structure: this is because the flat combining algorithm's high performance comes from exploiting the greater operation commutativity present in a non-transactional setting. To demonstrate this point, we propose a transactional specification that allows greater operation commutativity with the expectation that the flat combining technique can achieve better performance under this specification. Our experimental results illustrate that the commutativity of operations in this new setting is essential for the effectiveness of the flat combining technique.

4.1 Terminology

We first introduce some basic terminology (as defined in [33] and [38]) that will occur in our discussion.

4.1.1 Histories

Definition. A *history* is a sequence of `(transaction, operation, result)` tuples that represent an interleaving of operations of all committed transactions. 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 and operation return values.

Example.

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

Definition. A history H' is *consistent* with H if H' contains the same tuples as H: the same transactions were executed with the same return values for all operations within the transactions.

Definition. A history H is *serial* if all tuples are ordered as if all transactions were executed in a serial order.

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

Example. H is a serializable history given H'. H'' represents a serial but inconsistent history with H.

```
          H                        H'                       H''
(T2, Q.push(a), ())      (T2, Q.push(a), ())      (T1, Q.pop(), false)
(T1, Q.pop(), true)      (T2, Q.push(c), ())      (T1, Q.pop(), false)
(T2, Q.push(c), ())      (T1, Q.pop(), true)      (T2, Q.push(c), ())
(T1, Q.pop(), true)      (T1, Q.pop(), true)      (T2, Q.push(a), ())
```

Definition. A history H is *strictly serializable* and therefore *valid* if it is both serializable and all operations are linearizable. Informally, this means that the serial order of transactions corresponds to the real time at which the transactions commit.

[5] [5]LYT: examples?

4.1.2 Dependencies

Note that all operations can be classified as sets of reads and/or writes (as we do in Table 4.2.1). We therefore define dependencies abstractly as reads and writes of particular objects in our definitions.

Definition. Transaction T_2 is *dependent* on transaction T_1 if one of the following relations holds:

- R-R: T_1 reads an object later read by T_2
- R-W: T_1 reads an object later written by T_2
- W-R: T_1 writes an object later read by T_2
- W-W: T_1 writes an object later written by T_2

Definition. An operation P of transaction T_1 is *commutative* with operation Q of transaction T_2 when ordering P before Q results in a *R-R* dependency or no dependency at all.

4.1.3 Results

These results are well-known in the literature about transactional data structure scalability: we repeat them here for reference.

Theorem 4.1.1. *A history H is serializable if there are no cycles in R-W, W-R, or W-W dependencies between any two transactions (see [33] for proof).*

Corollary 4.1.1.1. *When operations commute, they can be freely ordered in the history without affecting serializability.*

Corollary 4.1.1.2. *The number of valid histories of a transactional data structure is dependent on the commutativity of its operations.*

Corollary 4.1.1.3. *Commutativity of data structure operations determines scalability: the greater number of valid histories, the lower the contention between transactions, and the greater the scalability of the transactional data structure.*

[6] [6]LYT: examples?

4.2 Commutativity of Concurrent and Transactional Queues

For generality, we reduce each queue operation to a read or write of a particular semantic object: the head, the tail, or the empty? predicate of the queue. This allows for our reasoning to be applied to operations that differ from our current specification of pop or push. For example, we can imagine an alternative pop operation that returns `void` regardless of whether the queue was empty, which would perform no visible reads. We summarize what reads and writes our specifications for pop and push perform in Table 4.2.1.

Operation	Read	Write
pop	empty?	head, empty?
push		tail, empty?

Table 4.2.1: Read and writes of queue operations

4.2.1 Concurrent Non-Transactional Queues

The guarantees of concurrent, non-transactional queues are *nearly* equivalent to that of singleton transactions. A history of singleton transactions is automatically serializable, since the history corresponding to the ordering of operation execution is a serial ordering of transactions. The atomicity of transactions is guaranteed by the correctness properties of the concurrent data structure. However, single operations are not always linearizable: depending on the implementation of the data structure, the effects of a operation P that has "committed" (i.e., has returned), may not be visible to an operation that is performed after P returns.

The flat combining technique provides serializable, atomic, *and* linearizable singleton transactions[17]. Therefore, we can reason about the commutativity and therefore scalability of a concurrent, non-transactional flat combining queue using notions of transactional dependencies.

We start by defining the dependency relations between single operations (Table 4.2.2). A concurrent, non-transactional queue cannot have any cyclical dependencies, since such dependencies necessarily require that there exists a transaction with more than one operation. Therefore, synchronization is only necessary to ensure correctness when two threads attempt to write (W-W) or simultaneously write/read (W-R or R-W) the same object: the performance bottleneck is only caused by concurrent access synchronization. As we have shown (**??**), the flat combining approach works particularly well to minimize the overhead of this synchronization cost.

4.2.2 Transactional Queues

With a fully-transactional queue, cyclical dependencies can occur, thus reducing the number of valid histories. All possible interleavings of two transactions that generate cyclical dependencies are shown in Table 4.2.3. Note that most of the interleavings that result in cyclical dependencies, and therefore invalid histories, occur only when the queue becomes empty; only when two transactions both perform pushes or both perform pops do we see cyclical dependencies in a nonempty queue.

Object	Pop-Pop	Pop-Push	Push-Pop	Push-Push
head	W-W			
tail				W-W
empty	W^e-R	R-W^e	W^e-R, W^e-W^e	

X-Y represents an operation X performed by one thread and an operation Y performed by another thread.
^{e} indicates that the operation modifies the empty status of the queue.
R-R relations are not shown.

Table 4.2.2: Dependencies of pairs of queue operations

```
1)  // Q.size() > 1
    (T1, Q.pop(), true) // Q empty
    (T2, Q.pop(), true) // W-W
    (T1, Q.pop(), *)    // W-W
```

```
4)  // Q.size() >= 0
    (T1, Q.push(a), ())
    (T2, Q.push(a), ()) // W-W
    (T1, Q.push(a), ()) // W-W
```

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

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

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

Interleavings that create no dependencies are left out.

Table 4.2.3: Operation interleavings generating dependency cycles.

A transactional queue prevents these interleavings from occurring through delaying push operation execution and using a pessimistic or optimistic approach upon encountering an empty queue.

To prevent interleavings 4 and 5, all pushes are delayed until commit time. These interleavings can only occur if T_1's first push is visible to T_2 prior to T_1 committing. If we delay pushes until commit time, T_2 will not detect the presence of a pushed item in the queue.

Because pop operations immediately return values that depend on the state of the queue (empty or nonempty), interleavings 1, 2, and 3 cannot be prevented by delaying pop operations until commit time. Instead, we can take one of two approaches:

1. Optimistic: Abort T_1 during commit time if T_2 has committed an operation that would cause an invalid interleaving.
2. Pessimistic: Prevent T_2 from committing any operation until after T_1 commits or aborts the transaction containing a pop.

ge see the optimistic method implemented in the T-Queue1, where the `tailversion` and `headversion` are used to determine at commit time if the empty status of the queue has been modified by another, already committed transaction. The pessimistic approach is implemented in our T-Queue2, 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.

In order to support transactions, the flat combining approach must do either approach (1) or (2). Here we argue that the flat combining approach cannot do either without introducing overhead that reduces its performance to below that of the T-Queue1 or T-Queue2.

If we take approach (1), a pop cannot be performed at execution time because no locks on the queue are acquired: other transactions may commit pops of an invalid head if this transaction later aborts. Thus, in order to determine if a pop should return true or if it should return false, a transactional pop flat combining request requires much more complexity than a nontransaction one: the thread must determine how many elements the queue holds, how many elements the current transaction is intending to pop, and if any other thread intends to pop (in which case the transaction aborts). The transactional push flat combining request is also significantly 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 the pop is performed at execution time, then the thread must acquire a global lock on the queue after a pop and hold the lock until commit: this prevents another thread from observing an inconsistent state of the queue. If a pop should execute and remove the head of the queue prior to commit and the transaction aborts, the popped elements must be re-attached to the head of the queue. However, the cleanup procedure during an abort does not lock the queue, meaning that any other thread can commit a transaction that pops off the incorrect head of the queue. To remedy this issue, a global lock must be acquired by any transaction that performs a pop to allow for the case in which 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) where if a transaction T_1 executes a pop, we disallow any pops from other transactions (using the equivalent of a global lock) but allow other transactions containing only pushes to commit prior to T_1 completing. This approach prevents interleavings 1 and 3, but requires performing a check of the queue's empty status, as in approach (1), if the queue is seen empty during a pop. This is because another transaction may have committed a push between the time of T_1's pop and T_1's completion. This mixed approach outperforms both approach (2) and approach (1), and is the approach described as the flat combining algorithm in Chapter 3.

As noted previously, all possible approaches to prevent interleavings 1, 2, and 3 rely on additional flat combining calls and increased 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. 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 to be necessary in order to provide transactional guarantees. The original flat combining algorithm exploits the property that any correct history of operations in data structures supporting only singleton transactions (i.e., a normal non-transactional data structure) is valid. The combiner thread is allowed to immediately apply all thread's operation requests in arbitrary order. However, this property that makes flat combining so performant disappears as soon as the algorithm has to deal with invalid histories. In the next section, we demonstrate how ignoring invalid histories leads to a version of flat combining that can outperform our T-Queue1 and T-Queue2 algorithms.

4.3 The Weakly Transactional Queue

The Weakly Transactional Queue (WT-FCQueue) demonstrates how the flat combining technique's performance is dependent upon the number of invalid histories. This queue implements a weaker transactional specification, which provides all invariants of a concurrent queue, but provides the following guarantees instead of the transactional ones listed earlier (Chapter 3):

- Instead of the normal pops, the queue executes *LazyPops* with the following specification:
  - Any two pops within the transaction do not need to pop consecutive values off the queue.

  - A pop's return value *cannot* be accessed during the transaction. The pops are applied and their return values determined only at commit time (hence the name LazyPop).

- A pop cannot remove an uninstalled value (i.e., a value pushed earlier in the same transaction).
- The *atomic* property thus becomes: a transaction with multiple pops and multiple pushes guarantees that the operations will either all occur or that none will occur.

Given this specification, interleavings 1, 2, and 3 in Table 4.2.3 are allowable if the return value of the pop is not used within same transaction. This is because two pops in a transaction do not need to pop consecutive values off the queue. Interleaving 4 is prevented by installing all pushes in the same transaction together at commit time, and interleaving 5 is allowed because T_1's pop cannot see its earlier pushed value.

The queue under this specification retains all the fairness properties of a concurrent queue (no value remains in the queue forever, since values are still removed in the order in which they are added). Like Schwarz[33], we see uses for this transactional queue as a buffer between producer and consumer activities, in which exact ordering of values in the buffer are unimportant. Our specification differs from that of Schwarz[33], however, by preventing a pop from seeing a push within the same transaction, and by preventing access to the return value of a pop until after the transaction commits. This allows us to utilize the flat combining approach to its full potential: 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.

4.3.1 Algorithm

For comparison, we implement two weakly transactional queues. This allows us to determine if changes in performance are due to the changes in the transactional specification, or are instead caused by differences in synchronization algorithms.

WT-Queue

The WT-Queue uses the naive synchronization strategy from the T-Queue1 and T-Queue2. The headversion (for pops) or tailversion (for pushes) is locked prior to actually performing the push or pop at commit time. A call to pop or push at execution time does not require any access to the queue state, but merely returns a LazyPop (pop) or adds an item to the `write_list` (push).

WT-FCQueue

The WT-FCQueue uses the flat combining synchornization algorithm. We modify the nontransactional, flat combining technique as follows:

- *Pops*: Executing a pop returns a LazyPop value. It does not generate a flat combining request or access the queue itself. At commit time, all LazyPops are instantiated with values: for each LazyPop, the thread makes a `<POP>` flat combining request. This request is completed using the vanilla, concurrent `<POP>` flat combining implementation, which simply pops an item off the queue.
- *Pushes*: Executing a push merely adds the value onto a `write_list_item` and does not access the queue. Pushes from the same transaction are installed together, using the `<PUSH, list>` flat combining implementation from the transactional flat combining queue that takes the entire list and pushes each value onto the queue.

4.3.2 Evaluation and Results

We evaluate the weakly transactional flat-combining queue on the same benchmarks described in Section 3.4.1 to compare against the fully transactional flat-combining queue (T-FCQueue), the T-Queue2, and the WrappedNT-FCQueue. Full results are shown in Figure A.2.5 and Figure A.3.5.

While the weakly-transaction, flat combining queue (WT-FCQueue) does not perform as well as its nontransactional counterpart, NT-FCQueue, the performance of the WT-FCQueue exceeds that of the T-Queue1, the T-Queue2, and the T-FCQueue, which all provide full-transactional guarantees. We see gains in performance over the T-Queue2 up to 1.5× as the number of threads accessing the queue increases to 18; the WT-FCQueue begins to outperform the STO queue as the number of threads increases past 7. The WT-FCQueue outperforms the T-FCQueue starting at 4 threads and achieves performance up to about 3× by 18 threads.

The WT-FCQueue does not experience any aborts. This is due to the flat combining algorithm, which does not require that any locks be held in the weakly transactional setting in order to ensure correctness. A transaction can only abort if the result of a pop is accessed during the transaction's execution.

Because of the lack of aborts, the WT-FCQueue outperforms its counterpart, the WT-Queue; this demonstrates the effectiveness of the flat combining technique in the weakly transactional setting. The WT-Queue performs worse than all queues measured because of its high abort rate at commit time (100% of aborts occur at commit time). This is caused by contention on the headversion and tailversion locks. While the actual pop function called during a transaction's execution is much simpler than in the T-Queue1 or T-Queue2 algorithms because it does not access the queue to check if the queue is empty, the installation procedure becomes more complicated because it requires instantiating all LazyPops, which also nearly triples the number of cache misses. The WT-FCQueue incurs approximately 1.5× more cache misses than the NT-FCQueue, but is not crippled from LazyPop-caused cache misses because the flat combining algorithm optimizes for efficient cache usage. Unlike the WT-FCQueue, the WT-Queue holds a global lock during installation at commit time, causing other transactions attempting to commit to abort.

5 hashmap Algorithms and Analysis

This chapter investigates different concurrent and transactional algorithms for hashmaps. We begin with an overview of concurrent and transactional hashmap specifications and algorithms. We then evaluate how these hashmaps perform on several microbenchmarks, and discuss why and how particular high-concurrency hashmap algorithms, modified to provide transactional guarantees, can outperform our current transactional algorithms.

5.1 Algorithms

We present the concurrency and transactional algorithms we analyzed, created, and implemented in our work. Some general terminology: an *element* refers to the key-value pair inserted into the hashmap. A *bucket* is a container of elements, and a hashmap consists of a set of buckets. The algorithms differ in the methods used to place elements in buckets and track how buckets and elements are modified.

5.1.1 STO chaining hashmap

The STO chaining hashmap (ChainingHashmap) 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. Thus, the worst case lookup/delete is $O(n)$. Inserts are always constant time and require allocating an element. Each bucket is associated with a *bucketversion* that increments upon any committed addition or removal from the bucket. The bucketversion is used to verify that no thread has added an element that was absent during a transaction's find. In addition, each bucket has a lock that synchronizes access to the bucket. Each inserted element is associated with an *elementversion* that tracks if the value to the element has been modified or if the element has been removed.

Elements are inserted at execution time but marked as *phantom*, allowing another transaction that sees ones of these uninstalled elements to realize it is viewing an inconsistent state of the map and therefore abort. If a transaction containing insertions aborts, these phantom elements are removed from the map. ELse thephantom mark is erased during commit. An alternative approach would be to insert all elements at commit time.However, this requires either relying on the bucketversion to determine if another transaction has inserted the same element (which would result in false aborts since the bucketversion 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. Deletions are delayed until commit time (an optimistic approach). This requires careful handling of cases of *read_my_writes*, such as deleting an element inserted in the same transaction,

5.1.2 Non-Transactional Cuckoo Hashmap

The Non-Transactional Cuckoo Hashmap (CuckooNT) implements a concurrent, non-transactional cuckoo hashing algorithm (implementation modified from [12]).

Each element is placed in one of two buckets; these buckets are determined by two different hash functions. A bucket has a fixed size of elements. This means that lookups and deletes only require executing two hash functions and checking the contents of two buckets (an $O(1)$ operation).

Inserts run in amortized time $O(1)$ but 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 alternative bucket by hashing e with the second hash function. If both buckets are full, cuckoo shuffling occurs. This process kicks out an element e' in one of e's buckets and places e' in e''s alternative bucket. If e''s alternative 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, reinserting all elements, but for simplicity, we keep the algorithm 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.

5.1.3 Transactional Cuckoo Hashmap

The transactional cuckoo hashmap comes in three flavors: allocating, allocating with key-fragments, and non-allocating. All flavors 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 STO chaining hashmap, each bucket has a *bucketversion* and lock and each element has an *elementversion*. Because insertions can occur due to cuckoo shuffling as well as an external insertion call, the bucketversion 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 deletions are delayed until commit time.

The allocating transactional cuckoo hashmaps allocate elements upon insertion. One variant (CuckooIE) contains buckets containing pointers to the elements, allowing STO to track elements by their memory address to verify elementversions at commit time. The key-fragments variant (CuckooKF) expands buckets to contain both an array of keys and an array of element pointers. This enables a lookup or delete for an absent item to skip following the pointer to the allocated internal element itself, and can reduce the number of cache line accesses depending on the workload.

The non-allocating transactional cuckoo hashmap consists of buckets consisting of a fixed-sized array of wrapped elements. STO tracks elements by their keys. Therefore, to verify if an elementversion 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 elementversion. Although this reduces the number of allocations, elementversions can now move between buckets, and the values in their previous locations invalidated. This complicates correctly checking and synchronizing the reads of elementversions.

5.2 Evaluation

5.2.1 Microbenchmarks

As with the queue, all hashmaps queues are evaluated on a set of microbenchmarks to demonstrate their scalability and performance. The controlled nature of these microbenchmarks allow us to easily compare particular aspects of each algorithm, such as transactional overhead introduced by STO. 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 graphs, we show the median of 5 consecutive runs with the minimum and maximum performance results represented as error bars.

Parameters

- Proportion of Finds/Inserts/Deletes: The ratio of inserts:deletes is kept at 1 to ensure that the hashmap does not always become empty or only grow in size. It is expected that half the inserts will succeed and half the deletes will succeed, since both are drawing key values from the same range. Tests of 5% inserts, 5% deletes, and 90% finds simulate the most likely use cases for hashmaps[34]. Tests of equal proportion (33%) of all operations investigate how the hashmap reacts to an increased rate of inserts and deletes.
- Operations per transaction: We choose to run all tests comparing transactional to non-transactional (parallel-only) 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: Both the cuckoo hashmaps and the chaining hashmap statically set the number of buckets in the data structure. The number of items allowed in one bucket of the cuckoo hashmaps is fixed at a particular value, which we will call the *maximum fullness*. With the Chaining hashmap, the number of items in one bucket can grow arbitrarily large because each bucket represents a linked list of values. The *capacity* (number of buckets $\times$ number of items per bucket) of the two map datatypes differs because the cuckoo hashmaps have a fixed size bucket, and therefore a finite capacity, while the Chaining hashmap The number of buckets and the size of each bucket affects 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: The ratio equivalent to (number of keys : number of buckets). This determines the average number of items to be found per bucket. The tests are implemented such that at steady state, fullness is expected to be 75% of the maximum fullness of the cuckoo hashmap to avoid an out-of-space exception. 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 fixed point.

Note that the initial size of the data structure should not affect performance as the test proceeds for a longer period of time and reaches a steady state. Therefore, we do not include the initial size as a benchmark parameter.

Multi-Thread, Variable-Capacity Singletons Test

This test is run with different numbers of threads and with different proportions of finds/inserts/deletes. Each thread runs 5 million singleton transactions. The test is run twice, once with a probability of 33% insert, 33% delete, and 34% find, and again with a probability of 5% insert, 5% delete, and 90% find. The steady-state final size is 50% maximum load.

5.2.2 Results and Discussion

We measure all hashmap's performance in terms of operations per second, abort rates, and cache performance (number of cache misses). The full results presented as graphs and tables can be found in Appendix B.

Cache Misses

The number of cache misses of each datatype is influenced by the number of allocations per data structure and the patterns in which these allocations are accessed. As expected, the non-transactional cuckoo hashmap experiences the least number of cache misses: this is because all other hashmaps' buckets contain a pointer to an allocated item inserted into the map. We expect that the non-allocating, transactional cuckoo hashmap will perform better than either of the allocating types [7]. As expected, the map of size 10K buckets performs best for all algorithms regardless of the maximum fullness allowed: this is because the table and all its elements fit entirely in cache.

[7]LYT: TODO?

iterations require looking at all of bucket vs. short chain All figures and data are in Appendix Section **??**.

- Proportion of Finds/Inserts/Deletes: Increasing the number of finds, as compared to insertions and deletions, increases the performance of all hashmaps by nearly 100% in certain cases. This is expected, since a find only performs a read of a bucket and can abort a transaction only if an insertion or deletion of an element in the same bucket simultaneously occurs.
- Load: As load increases, the performance of all hashmaps on both the 33%Find/33%Insert/33%Erase test and the 90%Find/5%Insert/5%Erase test decreases for all size hashmaps. Performance drops most during an increase from maximum load 5 to maximum load 10. This is expected since increased load results in increased numbers of cache misses [8], a constant multiplier on the time it takes to look up an element in the cuckoo hashmaps, and a greater possibility for longer chains in the chaining hashmap.

[8]LYT: numbers?

- Number of Buckets: Increase in the number of buckets decreases the abort rates, since the probability that two threads will simultaneously attempt to read/modify the same bucket decreases. However, performance is more heavily affected by the number of cache misses, as we see the performance drop as size increases and the number of cache misses also increases.

The relative performance of the different hashmaps remains consistent as load changes, with the

The chaining hashmap is least affected by changes in size, whereas the cuckoo hashmap variants are heavily influence by cache performance. This is due to the differences in the algorithms: [9]

[9]LYT: TODO

We note that the transactional cuckoohashmaps still perform as expected compared to the STO chaining hashmap: they outperform the chaining hashmap when run on large loads with a small hashmap. Thus, integrating the cuckoo hashmap into a transactional setting did not cripple the cuckoohashmap's performance as with the flat combining queue. This is because the cuckoohashmap algorithm's optimizations still pertain in the transactional setting: the speedup of the cuckoohashmap algorithm results from the constant time lookups and deletes, and amortized constant time insertions. These asymptotic time bounds still hold for our transactional algorithms for cuckoohashmap operations.

Furthermore, the overhead incurred by adding transactional guarantees to the cuckoohashmap is no more than the overhead incurred by adding the same guarantees to the chaining hashmap. A transactional find in a cuckoohashmap and a chaining hashmap adds to the read set an object representing the bucket in which the object should be (or is) located. A transactional insert requires adding a read (and write, depending if the inserted key is already present) of the bucket in which the object should be located. A transactional erase requires adding a read (and write, depending if the key to erase is present) of the bucket the object should be located. Thus, the

granularity of conflicts is still per-bucket, and the core optimizations which the cuckoohashmap takes to achieve better performance than the chaining hashmap in particular scenarios are still applicable.

6 Conclusion and Future Work

6.1 Queues

Our work with the flat combining queues show that

6.2 Hashmaps

6.3 Future Work

To further increase performance, we can specialize for singleton transactions...

We can also further test the idea that we can increase the performance of transactional STO data structures by incorporating high-concurrency algorithms only if the data structure experiences little to no reduction of operation commutativity in a transactional setting as compared to a concurrent one.

[10] For data structures in which operation commutativity crippled the optimizations in order to provide transactional guarantees (such as the FIFO queue or priority queue), we can explore further different, weaker transactional specifications that may provide some useful guarantees beyond that of a simple concurrent data structure and allow optimizations by concurrency algorithms to increase performance.

[10]LYT: todo

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

[7] S. Doherty, L. Groves, V. Luchangco, and M. Moir. *Formal Verification of a Practical Lock-Free Queue Algorithm*, pages 97–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[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] S. Feldman, P. LaBorde, and D. Dechev. Concurrent multi-level arrays: Wait-free extensible hash maps. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 155–163, July 2013.

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

[13] 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.

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

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

[30] 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.

[31] 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.

[32] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

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

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

Appendices

A

Queue Results

A.1 Cache Misses

The results are collected by evaluating the Multi-Thread Singletons Test with each thread running 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.

In these figures, we report the number of cache misses reported by perf (approximately 1/1000 of the actual number of cache misses.)

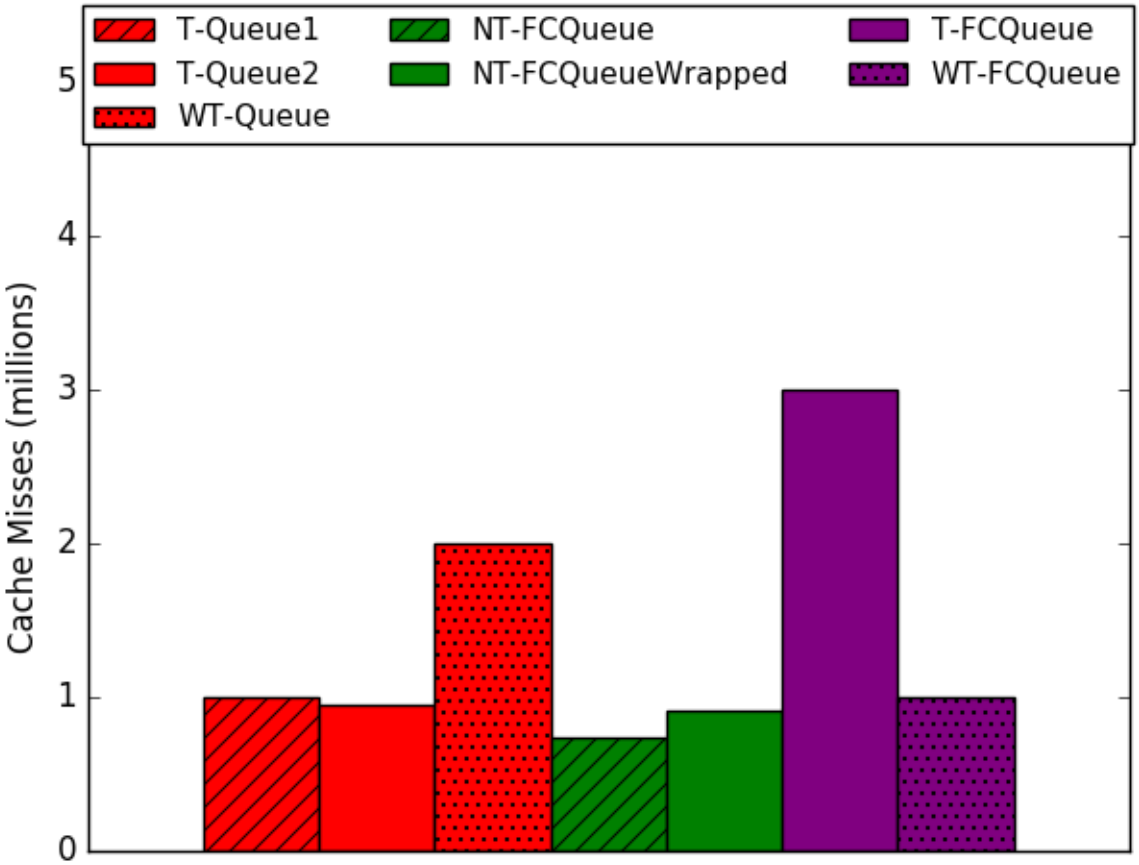


Figure A.1.1: Queue Cache Misses

A.2 Push-Pop Test Results

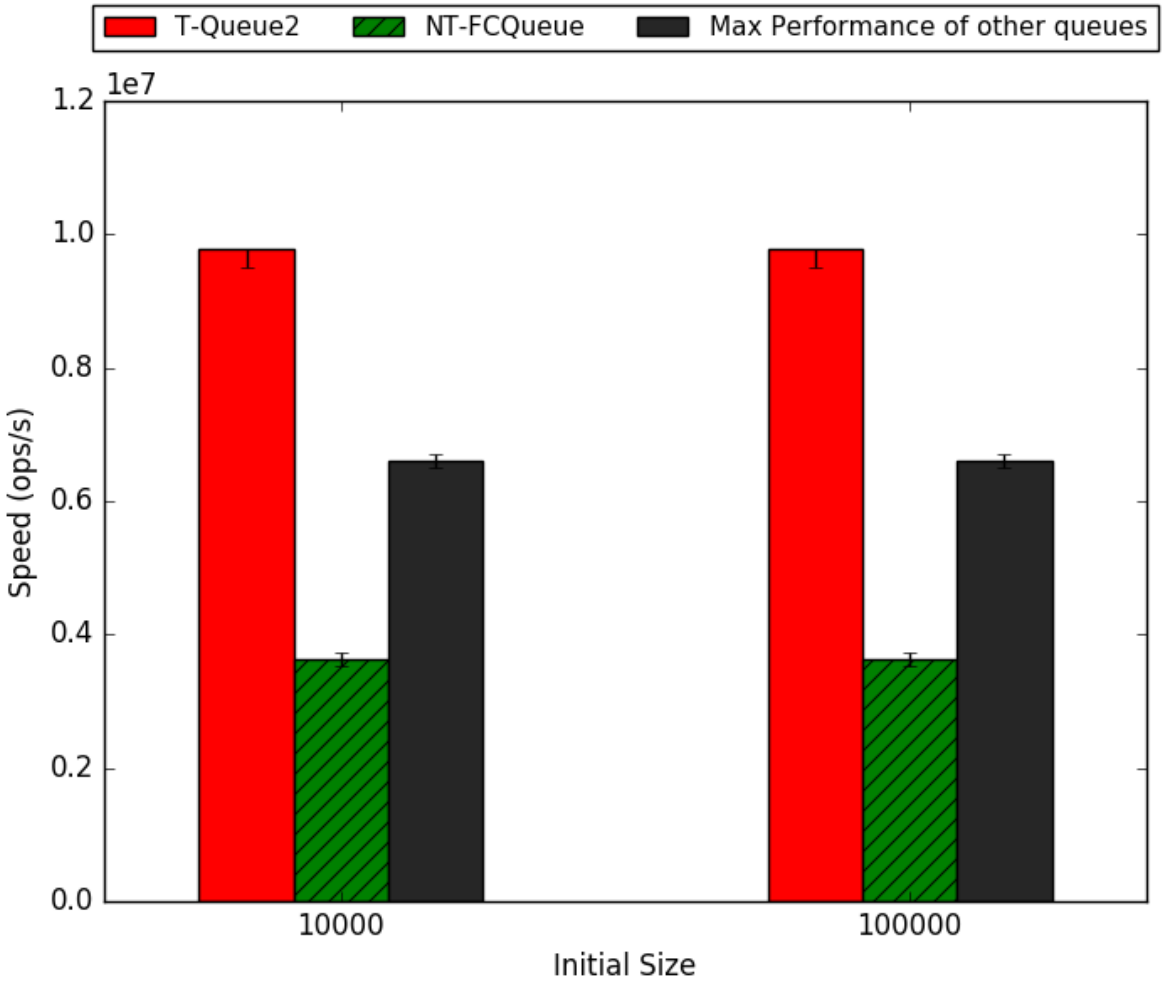
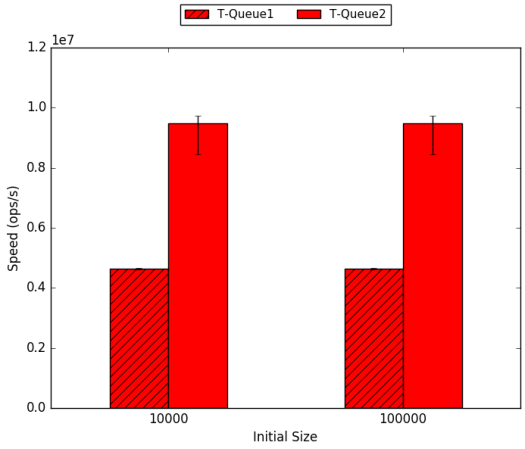
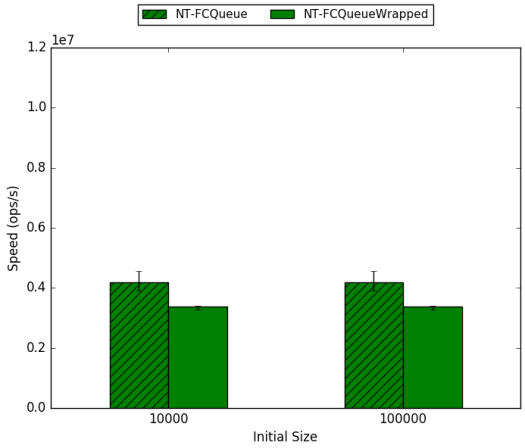


Figure A.2.1: Push-Pop Test: Concurrent, Non-transactional Queues



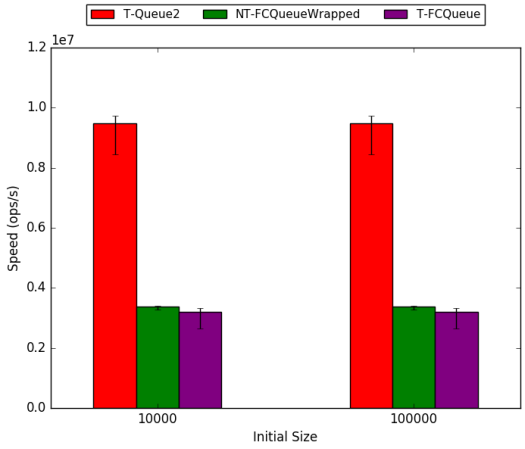
Queue	Initial Size Abort Rate (%)	
	10000	100000
T-Queue1	0.394	0.394
T-Queue2	1.454	1.454

Figure A.2.2: Push-Pop Test: T-Queue1 vs. T-Queue2



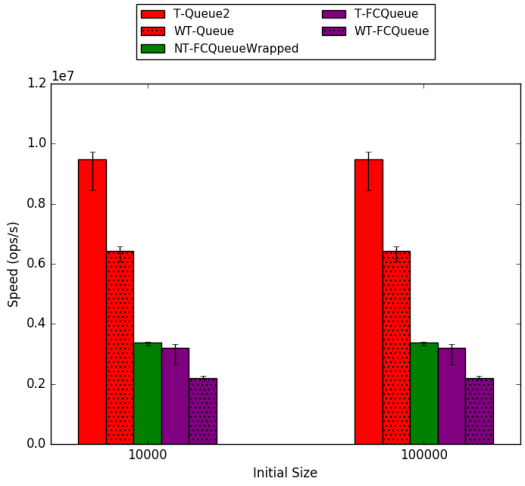
Queue	Initial Size Abort Rate (%)	
	10000	100000
NT-FCQueue	0.000	0.000
NT-FCQueueWrapped	0.000	0.000

Figure A.2.3: Push-Pop Test: NT-FCQueueWrapped vs. NT-FCQueue



Queue	Initial Size Abort Rate (%)	
	10000	100000
T-Queue2	1.454	1.454
NT-FCQueueWrapped	0.000	0.000
T-FCQueue	0.000	0.000

Figure A.2.4: Push-Pop Test: T-FCQueue



Queue	Initial Size Abort Rate (%)	
	10000	100000
T-Queue2	1.454	1.454
WT-Queue	0.000	0.000
NT-FCQueueWrapped	0.000	0.000
T-FCQueue	0.000	0.000
WT-FCQueue	0.000	0.000

Figure A.2.5: Push-Pop Test: WT-FCQueue

A.3 Multi-Thread Singletons Test Results

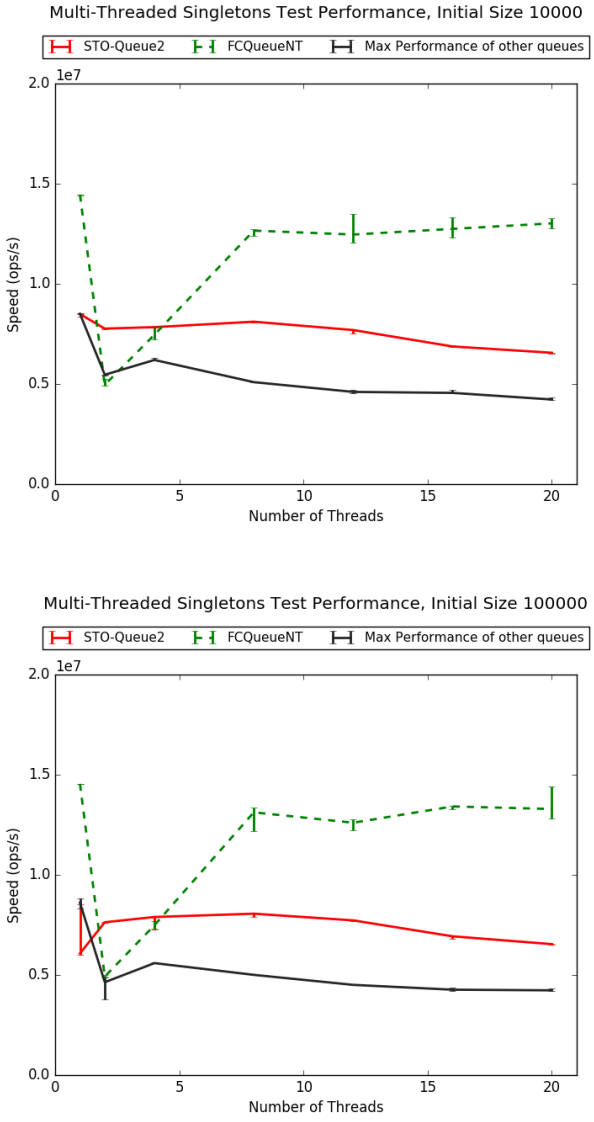
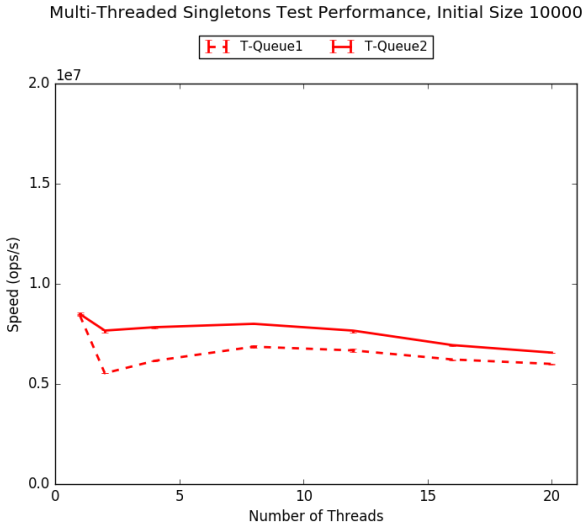
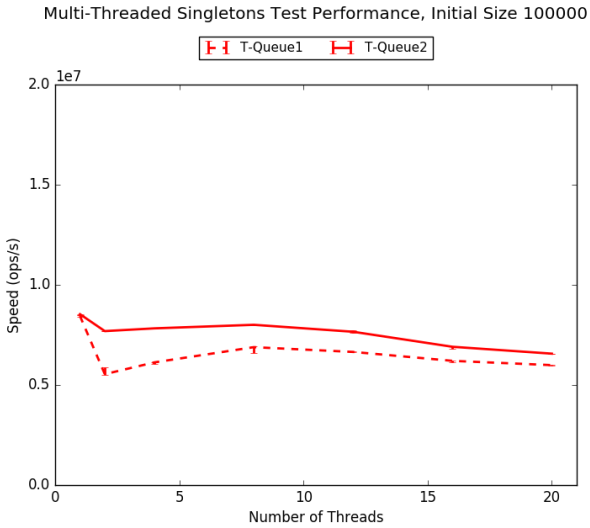


Figure A.3.1: Multi-Thread Singletons Test: Concurrent, Non-transactional Queues

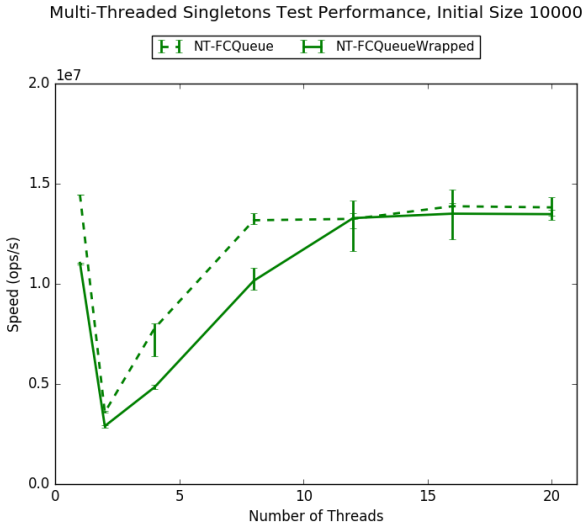


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue1	7.474	9.550	6.885	6.775	6.428	6.381
T-Queue2	3.292	2.875	2.233	2.296	2.379	2.462

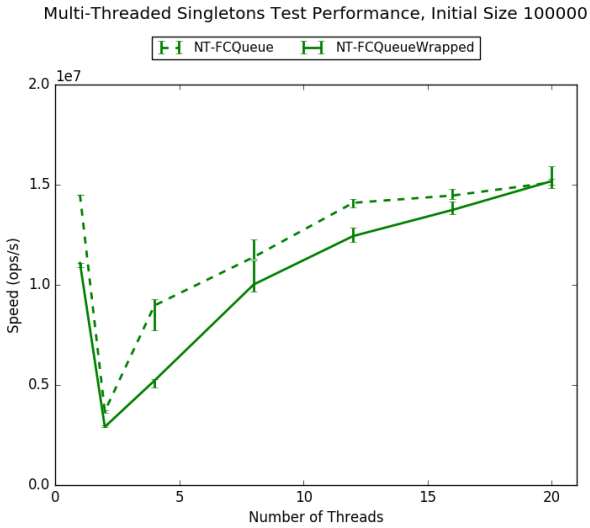


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue1	6.984	9.405	7.075	6.841	6.461	6.303
T-Queue2	3.077	2.809	2.162	2.324	2.380	2.446

Figure A.3.2: Multi-Thread Singletons Test: T-Queue1 vs. T-Queue2

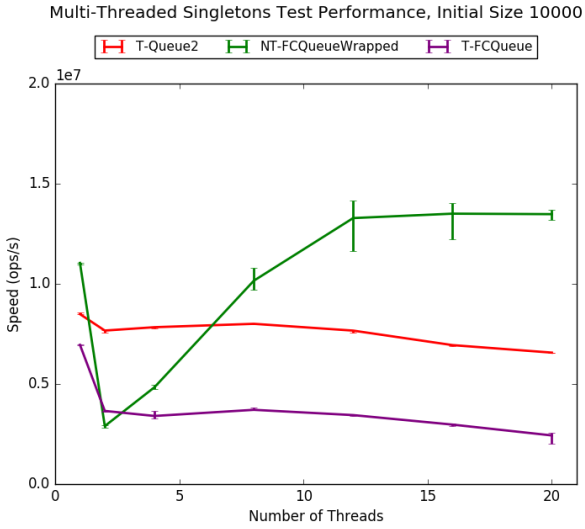


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
NT-FCQueue	0.000	0.000	0.000	0.000	0.000	0.000
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000

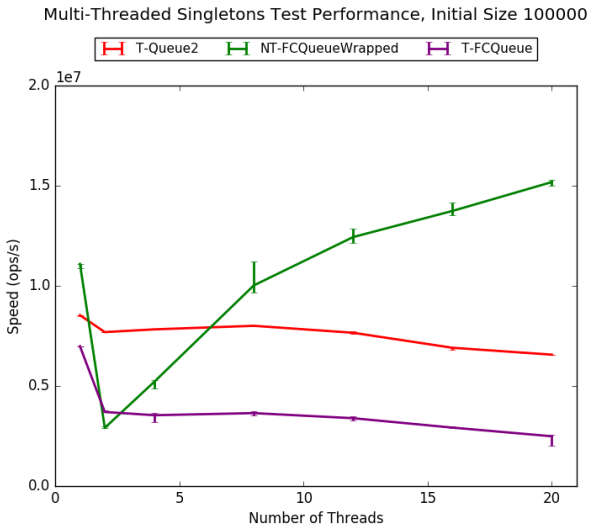


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
NT-FCQueue	0.000	0.000	0.000	0.000	0.000	0.000
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000

Figure A.3.3: Multi-Thread Singletons Test: NT-FCQueueWrapped vs. NT-FCQueue

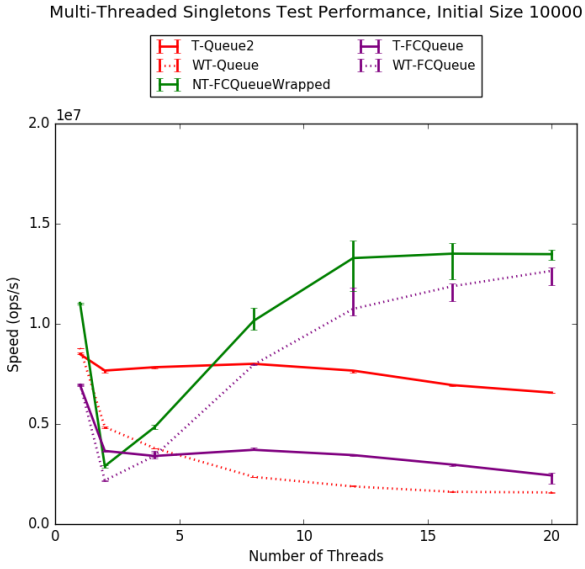


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue2	3.292	2.875	2.233	2.296	2.379	2.462
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000
T-FCQueue	3.364	4.629	3.394	3.468	3.741	4.288

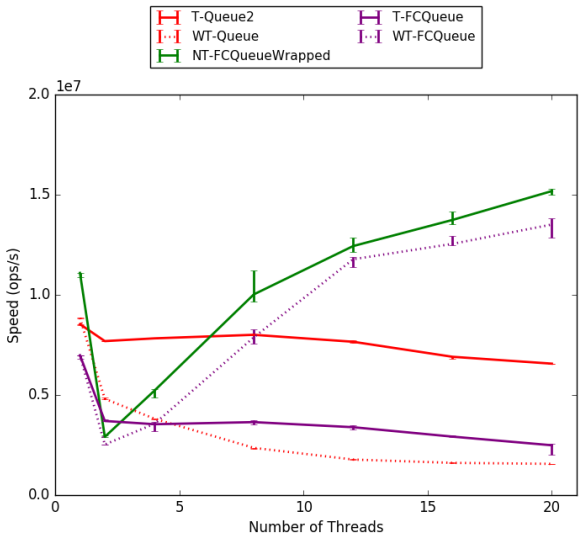


Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue2	3.077	2.809	2.162	2.324	2.380	2.446
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000
T-FCQueue	3.177	4.436	3.407	3.521	3.774	4.207

Figure A.3.4: Multi-Thread Singletons Test: T-FCQueue



Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue2	3.292	2.875	2.233	2.296	2.379	2.462
WT-Queue	2.767	17.016	45.886	57.891	66.785	71.006
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000
T-FCQueue	3.364	4.629	3.394	3.468	3.741	4.288
WT-FCQueue	0.000	0.000	0.000	0.000	0.000	0.000



Queue	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Queue2	3.077	2.809	2.162	2.324	2.380	2.446
WT-Queue	2.716	16.600	45.607	59.998	66.679	71.140
NT-FCQueueWrapped	0.000	0.000	0.000	0.000	0.000	0.000
T-FCQueue	3.177	4.436	3.407	3.521	3.774	4.207
WT-FCQueue	0.000	0.000	0.000	0.000	0.000	0.000

Figure A.3.5: Multi-Thread Singletons Test: WT-FCQueue

B
Hashmap Results

B.1 Cache Misses

The results are collected by evaluating the Multi-Thread Singletons Test with each thread running 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.

In these figures, we report the number of cache misses reported by perf (approximately 1/1000 of the actual number of cache misses.)

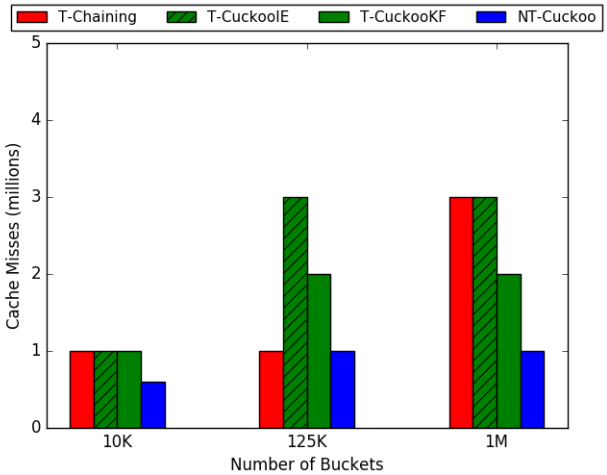


Figure B.1.1: Hashmap Cache Misses: Maximum Fullness 5

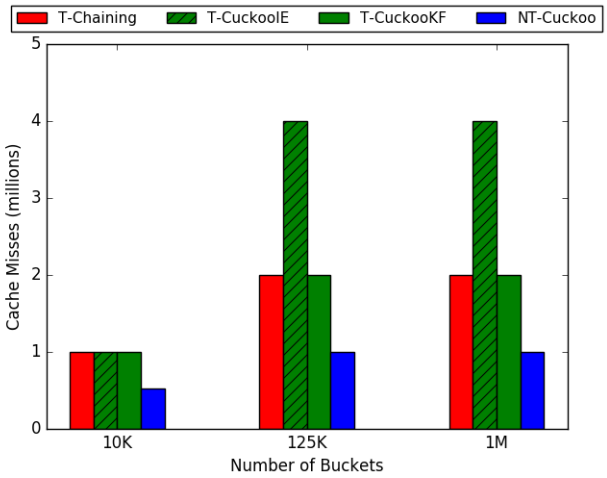


Figure B.1.2: Hashmap Cache Misses: Maximum Fullness 10

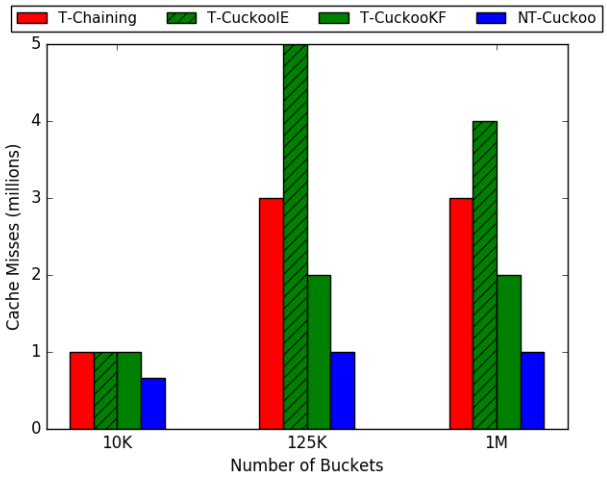
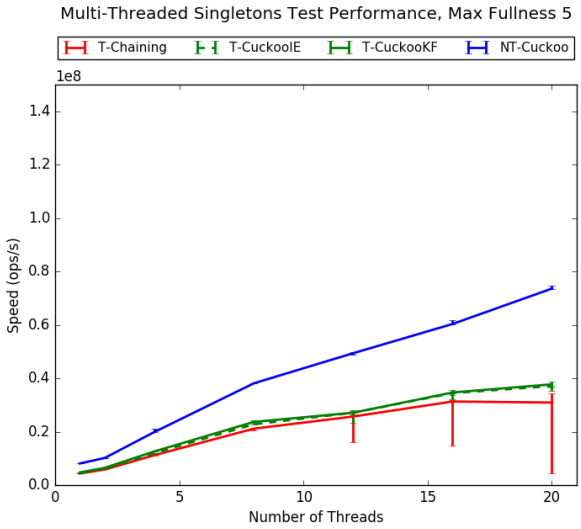


Figure B.1.3: Hashmap Cache Misses: Maximum Fullness 15

B.2 Multi-Thread Singleton Test Results: 33% Find, 33% Insert, 33% Erase

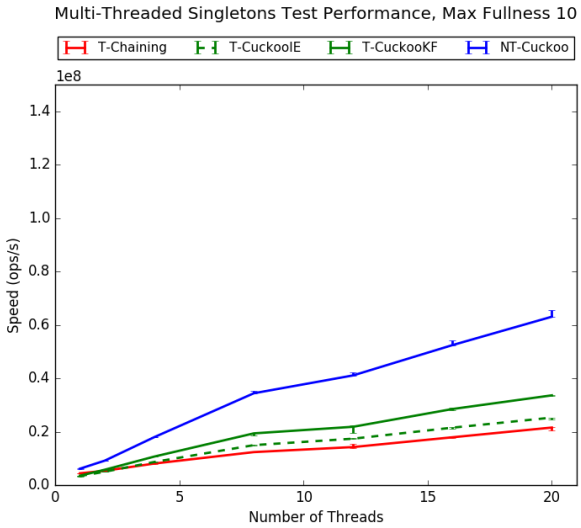
B.2.1 Maximum Fullness 5

Figure B.2.1: Hashmap Performance (33F/33I/33E): 10K Buckets, Maximum Fullness 5



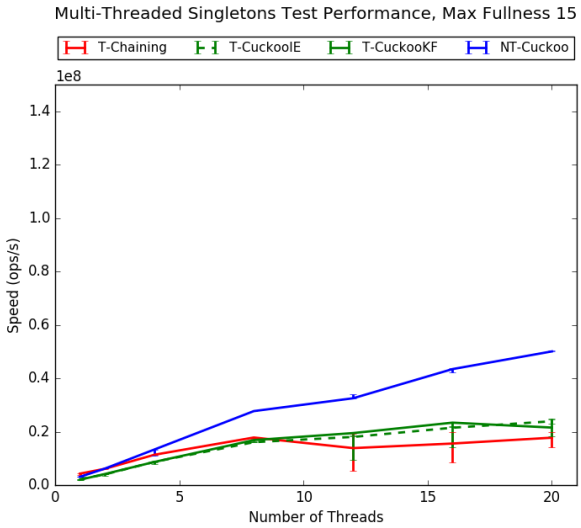
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.001	0.003	0.008	0.013	0.018	0.018
T-CuckooIE	0.000	0.001	0.003	0.005	0.007	0.008
T-CuckooKF	0.000	0.001	0.003	0.005	0.007	0.008
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.2.2: Hashmap Performance (33F/33I/33E): 125K Buckets, Maximum Fullness 5



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.001	0.001	0.001	0.002
T-CuckooIE	0.000	0.000	0.000	0.001	0.001	0.001
T-CuckooKF	0.000	0.000	0.000	0.000	0.001	0.001
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

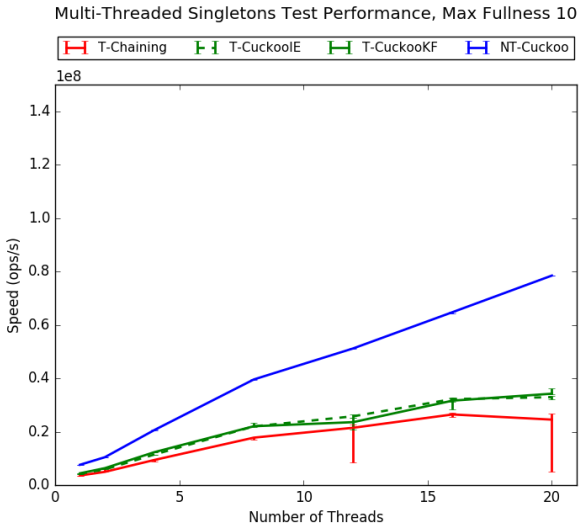
Figure B.2.3: Hashmap Performance (33F/33I/33E): 1M Buckets, Maximum Fullness 5



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

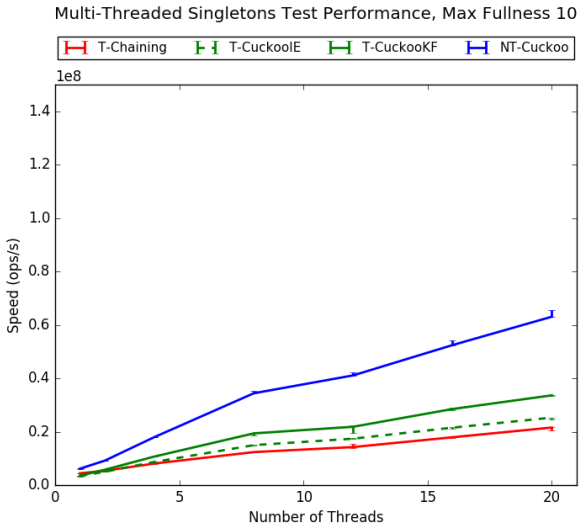
B.2.2 Maximum Fullness 10

Figure B.2.4: Hashmap Performance (33F/33I/33E): 10K Buckets, Maximum Fullness 10



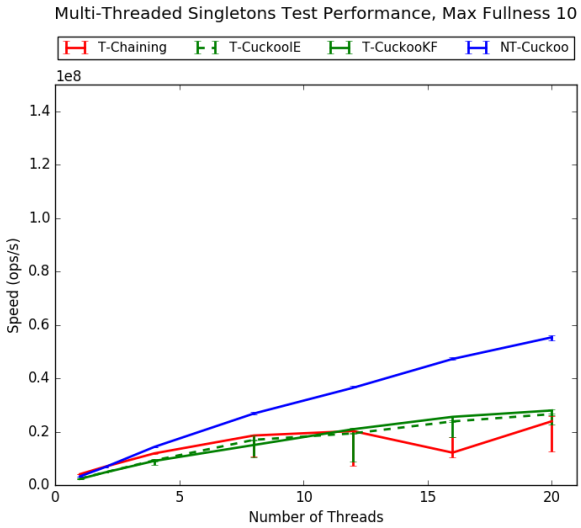
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.001	0.003	0.008	0.013	0.018	0.021
T-CuckooIE	0.000	0.001	0.002	0.004	0.006	0.005
T-CuckooKF	0.000	0.001	0.002	0.004	0.006	0.007
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.2.5: Hashmap Performance (33F/33I/33E): 125K Buckets, Maximum Fullness 10



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.001	0.002	0.002	0.002
T-CuckooIE	0.000	0.000	0.000	0.000	0.001	0.001
T-CuckooKF	0.000	0.000	0.000	0.000	0.001	0.001
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

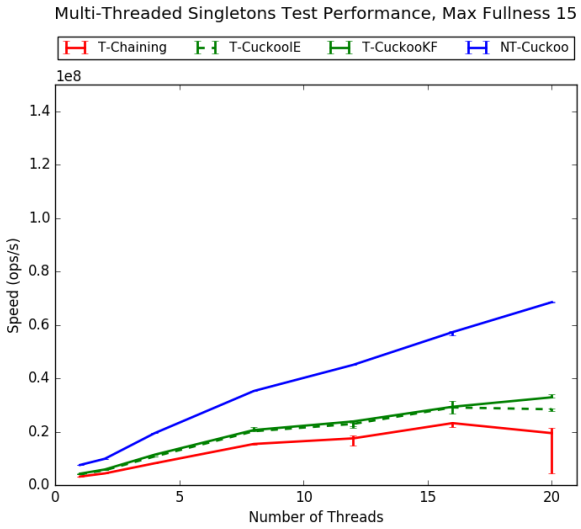
Figure B.2.6: Hashmap Performance (33F/33I/33E): 1M Buckets, Maximum Fullness 10



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.075	0.000
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

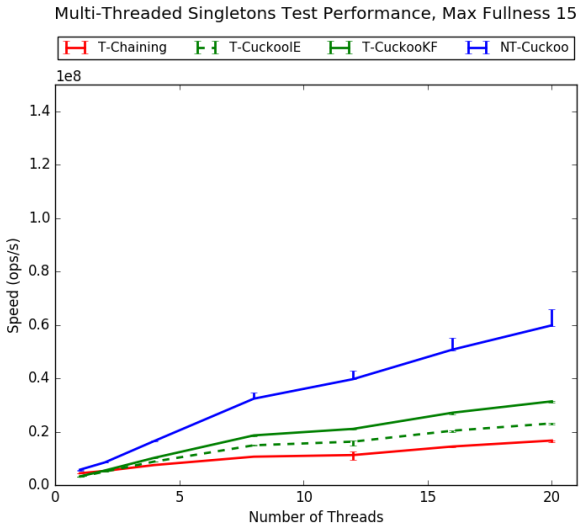
B.2.3 Maximum Fullness 15

Figure B.2.7: Hashmap Performance (33F/33I/33E): 10K Buckets, Maximum Fullness 15



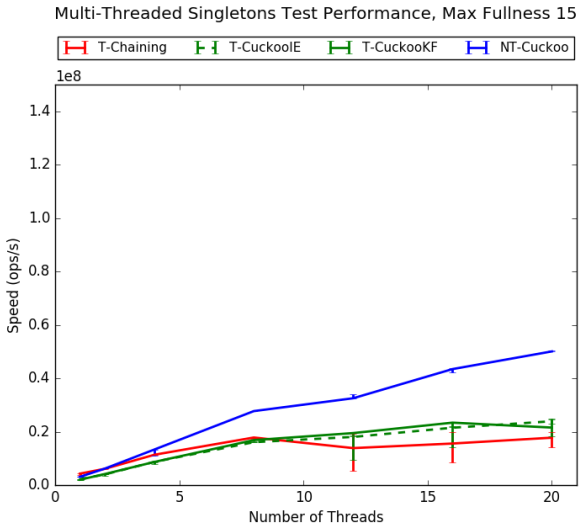
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.001	0.004	0.009	0.015	0.019	0.023
T-CuckooIE	0.000	0.001	0.002	0.004	0.006	0.005
T-CuckooKF	0.000	0.001	0.002	0.003	0.005	0.005
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.2.8: Hashmap Performance (33F/33I/33E): 125K Buckets, Maximum Fullness 15



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.001	0.002	0.002	0.003
T-CuckooIE	0.000	0.000	0.000	0.001	0.001	0.001
T-CuckooKF	0.000	0.000	0.000	0.000	0.001	0.001
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.2.9: Hashmap Performance (33F/33I/33E): 1M Buckets, Maximum Fullness 15

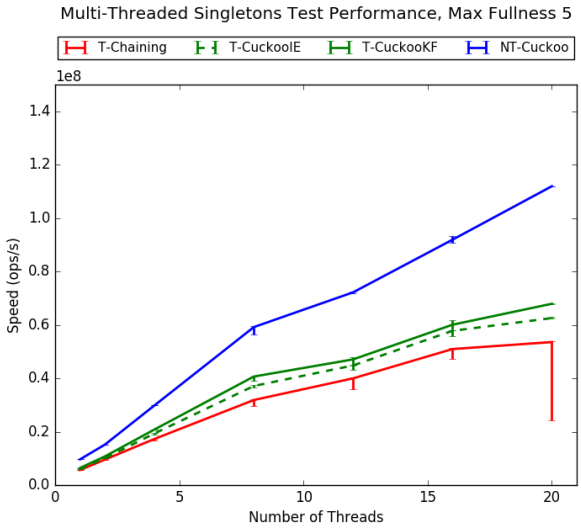


Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.001	0.001
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

B.3 Multi-Thread Singleton Test Results: 90% Find, 5% Insert, 5% Erase

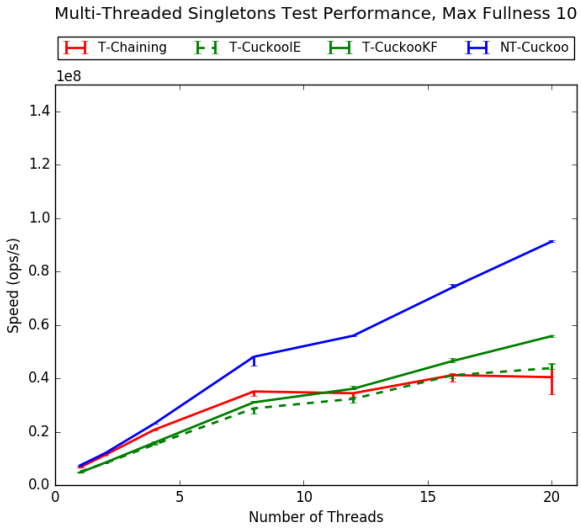
B.3.1 Maximum Fullness 5

Figure B.3.1: Hashmap Performance (34F/5I/5E): 10K Buckets, Maximum Fullness 5



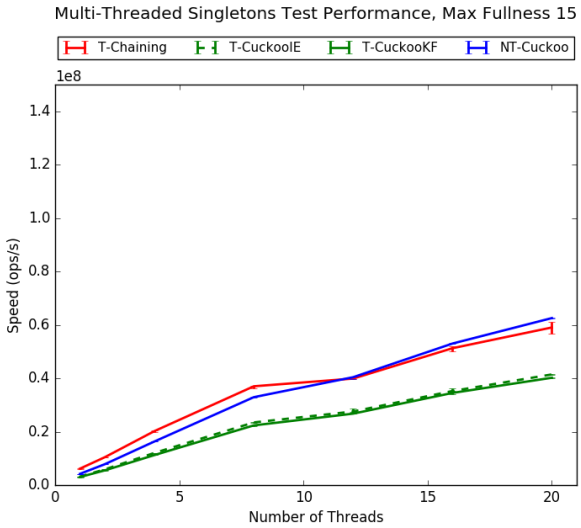
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.001	0.002	0.003	0.004	0.005
T-CuckooIE	0.000	0.001	0.001	0.001	0.002	0.002
T-CuckooKF	0.000	0.000	0.001	0.001	0.002	0.002
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.3.2: Hashmap Performance (34F/5I/5E): 125K Buckets, Maximum Fullness 5



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.001	0.001
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

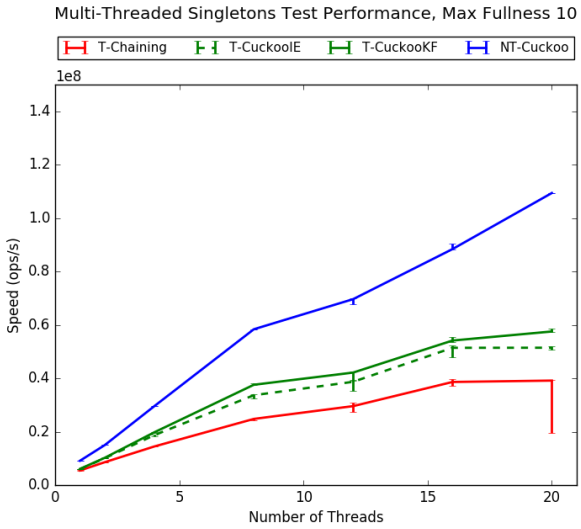
Figure B.3.3: Hashmap Performance (34F/5I/5E): 1M Buckets, Maximum Fullness 5



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

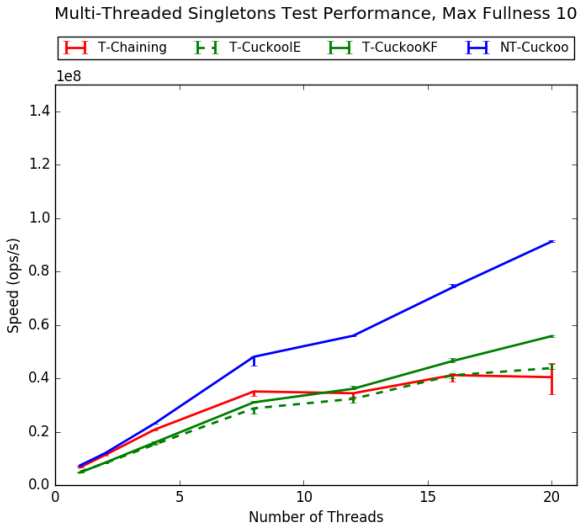
B.3.2 Maximum Fullness 10

Figure B.3.4: Hashmap Performance (90F/5I/5E): 10K Buckets, Maximum Fullness 10



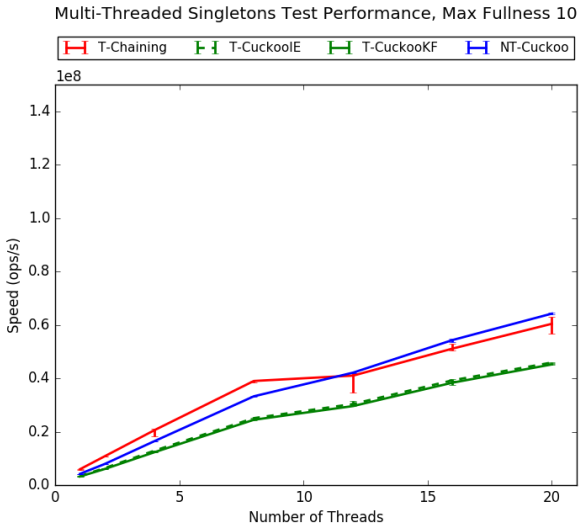
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.001	0.001	0.002	0.003	0.004	0.005
T-CuckooIE	0.000	0.001	0.001	0.002	0.002	0.002
T-CuckooKF	0.000	0.001	0.001	0.001	0.002	0.002
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.3.5: Hashmap Performance (90F/5I/5E): 125K Buckets, Maximum Fullness 10



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.001	0.001
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.001
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.001
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

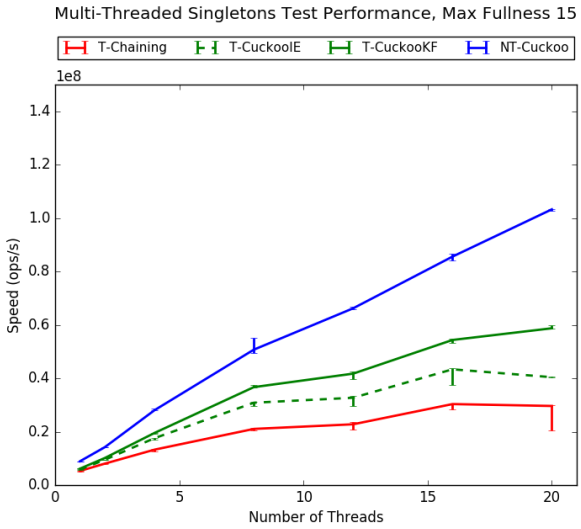
Figure B.3.6: Hashmap Performance (90F/5I/5E): 1M Buckets, Maximum Fullness 10



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

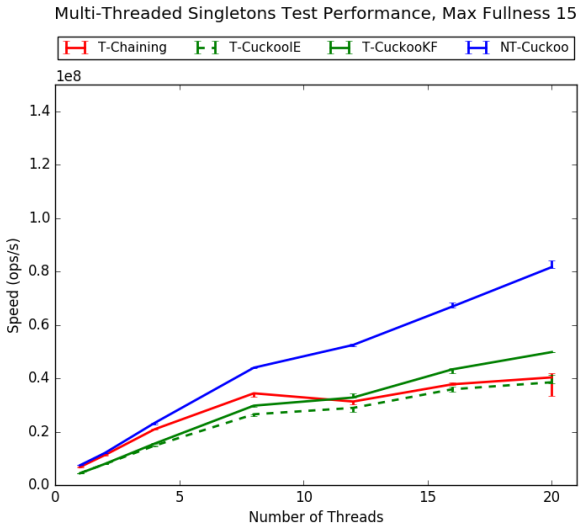
B.3.3 Maximum Fullness 15

Figure B.3.7: Hashmap Performance (90F/5I/5E): 10K Buckets, Maximum Fullness 15



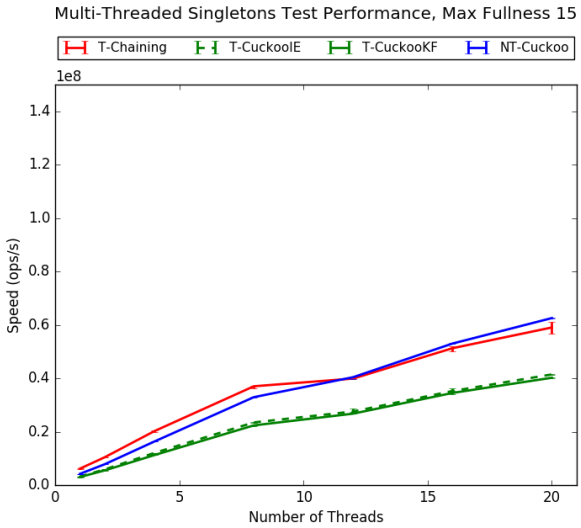
Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.001	0.003	0.004	0.004	0.006
T-CuckooIE	0.000	0.001	0.001	0.002	0.002	0.003
T-CuckooKF	0.000	0.001	0.001	0.002	0.002	0.003
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.3.8: Hashmap Performance (90F/5I/5E): 125K Buckets, Maximum Fullness 15



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.001	0.001	0.001
T-CuckooIE	0.000	0.000	0.000	0.000	0.001	0.001
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.001
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000

Figure B.3.9: Hashmap Performance (90F/5I/5E): 1M Buckets, Maximum Fullness 15



Hashmap	Number of Threads Abort Rate (%)					
	2	4	8	12	16	20
T-Chaining	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooIE	0.000	0.000	0.000	0.000	0.000	0.000
T-CuckooKF	0.000	0.000	0.000	0.000	0.000	0.000
NT-Cuckoo	0.000	0.000	0.000	0.000	0.000	0.000