Transactional Collection Classes

Brian D. Carlstrom Austen McDonald Michael Carbin Christos Kozyrakis Kunle Olukotun

Computer Systems Laboratory
Stanford University
{bdc, austenmc, mcarbin, kozyraki, kunle}@stanford.edu

Abstract

While parallel programmers find it easier to reason about large atomic regions, the conventional mutual exclusion-based primitives for synchronization force them to interleave many small operations to achieve performance. Transactional memory promises that programmers can use large atomic regions while achieving similar performance. However, these large transactions can conflict when operating on shared data structures, even for logically independent operations. *Transactional collection classes* address this problem by allowing long-running transactions to operate on shared data while eliminating unnecessary conflicts. Transactional collection classes wrap existing data structures, without the need for custom implementations or knowledge of data structure internals.

Without transactional collection classes, access to shared data from within long-running transactions can suffer from data dependency conflicts that are logically unnecessary, but are artifacts of the data structure implementation such as hash table collisions or tree-balancing rotations. Our transactional collection classes use the concept of *semantic concurrency control* to eliminate these unnecessary data dependencies, replacing them with conflict detection based on the operations of the abstract data type.

The design and behavior of these transactional collection classes is discussed with reference to the related work from the database community such as multi-level transactions and semantic concurrency control, as well as other concurrent data structures such as `java.util.concurrent`. The required transactional semantics needed for implementing transactional collection are enumerated, including open-nested transactions and commit and abort handlers. We also discuss how isolation can be reduced for greater concurrency. Finally, we provide guidelines on the construction of classes that preserve isolation and serializability.

The performance of these classes is evaluated with a number of benchmarks including targeted micro-benchmarks and a version of SPECjbb2000 with increased contention. The results show that easier-to-use long transactions can still allow programs to deliver scalable performance by simply wrapping existing data structures with transactional collection classes.

Categories and Subject Descriptors C.5.0 [*Computer Systems Implementation*]: General; D.1.3 [*Programming Techniques*]: Concurrent Programming – parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – concurrent programming structures

General Terms Performance, Design, Languages

Keywords Transactional Memory, Collection Classes, Java, Multiprocessor Architecture

1. Introduction

Transactional memory has been proposed as a way to ease parallel programming [15, 28, 14, 13], which has recently become more important with the shift towards chip multiprocessors (CMPs) [18, 17]. Although the promise of transactional memory is an easier-to-use programming model, evaluation of proposed systems thus far has focused on applications that have short critical regions tuned for use with locks [4].

For transactional memory to have a real impact, it should not focus on competing with existing hand-tuned applications but emphasize how transactions can make parallel programming easier while maintaining comparable performance [5]. Building efficient parallel programs is difficult because fine-grained locking is required for scaling, which burdens programmers with reasoning about operation interleaving and deadlocks. Large critical regions make it easier on programmers, but degrade performance. However, long-running transactions promise the best of both worlds: a few coarse-grained atomic regions speculatively executing in parallel.

While programming with fewer, longer transactions can make it easier to create correct parallel programs, the downside is that updates to shared state within these transactions can lead to frequent data dependencies between transactions and more lost work when there are conflcits. The dependencies can arise from both the program's own shared data structures as well as underlying library and run-time code. Often the implementation of these structures is opaque to a programmer, so eliminating dependencies is difficult.

Existing solutions utilize a technique called open nesting [2, 25] to expose updates to shared data structures early, before commit, and reduce the length of dependencies. This violates the isolation property of transactions and can lead to incorrect and unpredictable programs. However, structured use of open-nested transactions can give the performance benefits of reduced isolation while preserving the semantic benefits of atomicity and serializability for the programmer.

In this paper, we use *semantic concurrency control* and *multi-level transactions* combined with object-oriented encapsulation to create data structures that maintain the transactional properties of atomicity, isolation, and serializability by changing unneeded memory dependencies into logical dependencies on abstract data types. At times when full serializability is not required for program correctness, isolation between transactions can be relaxed to improve concurrency. Simple examples like global counters and unique identifier (UID) generators illustrate the usefulness of reduced isolation. The UID example is quite similar to the monotonically increasing identifier problem that the database community uses to demonstrate the tradeoffs between isolation and serializability [10].

To illustrate the need for semantic concurrency control when programming with long transactions, we present a parallelization of a high contention variant of the SPECjbb2000 benchmark [29]. This parallelization includes both the use of `Map` and `SortedMap` as well as the simpler examples of global counters and unique identifier (UID) generation. While the abstract data type examples show how transactional properties can be preserved, the counter and UID examples illustrate how selectively reducing isolation and forgoing serializability can be beneficial as well.

Specifically, our contributions are:

- The design and evaluation of transactional collection classes for use with hardware or software transactional memory systems. Transactional collection classes wrap existing Java collection implementations, and then can be used in long-running transactions without causing unnecessary dependencies. Because they offer the same interface as the underlying implementation, they can serve as drop-in replacements in existing programs.
- Transactional collection classes that allow programmers to compose multiple operations on transactional objects atomically — something unattainable with undisciplined use of isolation-reducing mechanisms such as open nesting.
- We show how selectively reducing isolation can yield higher performing data structures by creating a transactional work queue from the `Queue` class.
- We provide an overview of the transactional memory semantics needed to support the construction of collection classes in either hardware or software systems.
- We summarize design principles about managing state with semantic concurrency control for the case when full isolation and serializability is desired. We also discuss alternative implementation strategies that we did not explore that may be more appropriate for other transactional memory implementations.

The rest of the paper is organized as follows. Section 2 discusses semantic concurrency control in databases and its application to transactional memory via transactional collection classes. We also cover the related background on concurrent collection classes. Section 3 covers the design and implementation of transactional wrappers for Java collections. Section 4 discusses the transactional memory semantics required for our proposal, and Section 5 lists our guidelines for serializability. Section 6 evaluates the scalability of our transactional collection classes, and we conclude in Section 7 with some observations and suggestions for future directions.

2. Supporting Long-Running Transactions

The database community has studied the problem of frequent dependency conflicts within long-running transactions. We examined the literature surrounding semantic concurrency control, one solution to the long-running transaction problem. This section describes the evolution of semantic concurrency control, drawing similarities between the problems of databases and the problems of transactional memory. We will then show an example of how these ideas can be applied directly to transactional memory.

2.1 Database Concurrency Control

Isolation, one of the four ACID properties of database transactions, means that changes made by a transaction are not visible to other transactions until that transaction commits. An important derivative property of isolation is serializability, which means that there is a serial ordering of commits that would result in the same final outcome. Serializability is lost if isolation is not preserved, because if a later transaction sees uncommitted results that are then rolled back, the later transaction's outcome depends on data from a transaction that never committed, which means there is no way a serial execution of the two transactions would lead to the same result.

One method for databases to maintain isolation and therefore serializability, is strict two-phase locking. In this form of two-phase locking, the growing phase consists of acquiring locks before data access and the shrinking phase consists of releasing locks at commit time [10].

While simple, this isolation method limits concurrency. Often transactions contain sub-operations, known as nested transactions, which can access the state of their parent transaction without conflict, but which themselves can cause dependencies with other transactions. Moss showed how two-phase locking could be used to build a type of nested transactions where sub-operations could become child transactions, running within the scope of a parent, but able to rollback independently, therefore increasing concurrency (called *closed nesting*) [24]. Gray concurrently introduced a type of nested transaction where the child transaction could commit before the parent, actually reducing isolation and therefore further increasing concurrency because the child could logically commit results based on a parent transactions that could later abort (called *open nesting*) [9].

Open-nested transactions may seem dangerous — exposing writes from a child transaction before the parent commits and discarding any read dependencies created by the child — but they can be very powerful if used correctly. Trager notes how System R used open-nesting "informally" by releasing low-level page locks before transaction commit in violation of strict two-phase locking [30]. System R protected serializability through higher-level locks that are held until commit of the parent transaction, with compensating transactions used to undo the lower-level page operations in case the parent transaction needed to be rolled back.

System R's approach was later formalized as *multi-level transactions*: protecting serializability through locks at different layers [32, 26]. Going a step further and incorporating knowledge about the way specific data structures operate allowed semantically non-conflicting operations to execute concurrently; this was called *semantic currency control* [31, 27]. Finally, *sagas* focused on using compensating transactions to decompose a long-running transaction into a series of smaller, serial transactions [8].

2.2 Concurrent Collection Classes

Beyond parallel databases, another area of research in concurrency is data structures. Easier access to multi-processor systems and programming languages, like Java, that include threads have brought attention to the subject of concurrent collection classes. One major area of effort was `util.concurrent` [20], which became the Java Concurrency Utilities [16]. The original work within `util.concurrent` focused on `ConcurrentHashMap` and `ConcurrentLinkedQueue`, the later based on work by [23]. However, the upcoming JDK 6 release extends this to include a `ConcurrentSkipListMap` that implements the new `NavigableMap` interface that is an extension `SortedMap`.

The idea behind `ConcurrentHashMap` is reducing contention on a single size field and frequent collisions in buckets. The approach is to partition the table into many independent segments, each with their own size and buckets. This approach of reducing contention through alternative data structure implementations has been explored in the transactional memory community as well as we will see below.

2.3 Transactional Memory

Hardware Transactional Memory (HTM) [15] and Software Transactional Memory (STM) [28] have both seen a resurgence of research activity. For HTM, this has been due to the move from pursuing gains through instruction-level parallelism to thread-level parallelism with chip multi-processors. For STM, this has been due to the arrival of new innovative techniques for lower overhead implementations [13, 6], as well as the need to more easily take advantage of chip multi-processor systems.

There has been some work at the intersection of transactional memory and concurrent data structures. Adl-Tabatabai et al. used a `ConcurrencyHashMap`-like data structure to evaluate their STM system [1]. Kulkarni et al. suggested the use of open-nested transactions for queue maintenance for Delaunay mesh generation [19]. While this work addressed issues with specific structures, it did not provide a general framework for building transactional data structures.

Pausing transactions was suggested as an alternative to open-nesting for reducing isolation between transactions by Zilles and Baugh in [33]. Pausing could be used in the place of open nesting to implement semantic concurrency control, but because pausing does not provide any transactional semantics, traditional methods of moderating concurrent access to shared state such lock tables would need to be used.

Recently, Moss has advocated a different approach less focused on specific data structures. Based on his experience with closed-nested transactions [24], multi-level transactions [26], and transactional memory [15], he has been advocating the use of abstract locks built with open-nested transactions for greater concurrency. This paper builds on this concept and develops a set of general guidelines and mechanisms for practical semantic concurrency in object-oriented languages. We also include an evaluation of a full implementation of collection classes for use in SPECjbb2000.

2.4 The Need for Semantic Concurrency Control

To explain how ideas from the database community can be applied to transactional memory, we will consider a hypothetical `HashTable` class:

```
class HashTable {
    Object get  (Object key) {...};
    void   put  (Object key, Object value) {...}; }
```

Semantically speaking, `get` and `put` operations on different keys should not cause data dependencies between two different transactions. Taking advantage of this would be utilizing semantic concurrency control and is based on the fact that such operations are commutative.

The problem is that semantically independent operations may actually be dependent at the memory level due to implementation decisions. For example, hash tables typically maintain a load factor which relies on a count of the current number of entries. If we just use the traditional `java.util.HashMap`-style implementation within a transaction, semantically non-conflicting inserts of new keys will cause a memory-level data dependency as both inserts will try and increment the internal size field. Similarly, a `put` operation can conflict with other `get` and `put` operations accessing the same bucket.

Alternative `Map` implementations built especially for concurrent access such as `ConcurrentHashMap`, internally use multiple hash table segments to reduce contention. As mentioned above, others have used similar techniques in transactional contexts to reduce chances of conflicts on a single size field [1]. Unfortunately, while the segmented hash table approach statistically reduces the chances of conflicts in many cases, its does not eliminate them. In fact, the more updates to the hash table, the more segments likely to be touched. If two long-running transactions perform a number of insert or remove operations on different keys, there is a large probability that at least one key from each transaction will end up in the same segment, leading to memory conflicts on the segment's size field.

The solution is to use multi-level transactions. The low-level transactions are open-nested and used to record local changes and acquire higher-level abstract data type locks. The high-level transaction then uses these locks to implement semantic concurrency control.

In our `HashTable` example, the `get` operation takes a read lock on the key and retrieves the appropriate value, if any, all within an open-nested transaction. The `put` operation can use a thread-local variable to store the intent to add a new key-value pair to the table, deferring the actual operation. If the parent transaction eventually commits, a *commit handler* is run that updates the `HashTable` to make its changes visible to other transactions, as well as aborting other transactions that hold conflicting read locks. If the parent transaction is aborted, an *abort handler* rolls back any state changed by open-nested transactions.

Before applying multi-level transactions, an unnecessary memory-level conflict would abort the parent transaction. Now, memory-level rollbacks are confined to the short-running, open-nested transaction on `get` and the closed-nested transaction that handles committing `put` operations. In the `get` case, the parent does not rollback, and the `get` operation is simply replayed. In the `put` case, only the commit handler can have memory-level conflicts, and it too can be replayed without rolling back the parent transaction. Note that semantic conflicts are now handled through code in the commit handler that explicitly violates other transactions holding locks on modified keys. The responsibility for isolation, and therefore serializability, has moved from the low-level transactional memory system to our higher-level abstract data type.

To summarize, our general approach to building transactional versions of abstract data types is as follows:

1. take semantic locks on read operations
2. check for semantic conflicts while writing during commit
3. clear semantic locks on abort and commit

Having a general approach is more desirable than relying on data structure-specific solutions, like segmented hash tables. For example, the `SortedMap` interface is typically implemented by some variant of a balanced binary tree. Parallelizing a self-balancing tree would involve detailed analysis of the implementation and solving issues like conflicts arising from rotations. Semantic concurrency control avoids these issues by allowing the designer to reuse existing, well-designed and tested implementations.

In the following section, we will discuss more about our approach, as we cover the semantic operations involved with Java collection classes as well as our implementation of semantic locks.

3. Transactional Collection Classes

Simply accessing data structures within a transaction will achieve atomicity, isolation, and serializability, but long-running transactions will be more likely to violate due to the many dependencies created within the data structure. Simply using open-nesting to perform data structure updates would increase concurrency, but prevents users from atomically composing multiple updates, as modifications will be visible to other transactions. Transactional collection classes leverage semantic knowledge about abstract data types to allow concurrent and atomic access, without the fear of long-running transactions frequently violating.

Creating a transactional collection class involves first identifying semantic dependencies, namely which operations much be pro-

Read \ Write	put	remove
containsKey	if put adds a new entry with same key	if remove takes away entry with matching key
get	if put adds a new entry with same key	if remove takes away entry with matching key
size	if put adds a new entry	if remove takes away an entry
entrySet.iterator.hasNext	if hasNext is false and put adds a new entry	remove takes away key in iterated range
entrySet.iterator.next	put adds key in iterated range	remove takes away key in iterated range
Write		
put	if both write to the same key	if both operate on the same key
remove	if both operate on the same key	if both remove the same key

Table 1. Semantic operational analysis of the `Map` interface showing the conditions under which conflicts arise between primitive operations. Both read and write operations are listed along the left side but only write operations are listed across the top. The read operations are omitted along the top since read operations do not conflict with other read operations. If the condition is met, there needs to be an ordering dependency between the two operations. For example, the upper left condition says that if a `put` operation adds an entry with a new key in one transaction and another transaction calls `containsKey` on that same key returning false, there is a conflict between the transactions because they are not serializable if the `put` operations commits before the `containsKey` operation, which would be required to return true in a serializable schedule.

Methods	**Read Lock**	**Write Conflict**
Read		
containsKey	key lock on argument	
get	key lock on argument	
size	size lock	
entrySet.iterator.hasNext	size lock on false return value	
entrySet.iterator.next	key lock on return value	
Write		
put	key lock on argument	key conflict based on argument, size conflict if size increases
remove	key lock on argument	key conflict based on argument, size conflict if size decreases

Table 2. Semantic locks for `Map` describe read locks that are taken when executing operations as well as lock based conflict detection that is done by writes at commit time. For example, the `containsKey`, `get`, `put`, and `remove` operations take a lock for the key that was passed as an argument to these methods. When a transaction containing `put` or `remove` operations commits, it aborts other transactions that hold locks on the keys it is adding or removing from the `Map` as well as on other transactions that have read the size of the `Map` if it is growing or shrinking.

Category	**Field**	**Description**
Commited State		*commited state visible to all transactions*
	`Map map`	the underlying `Map` instance
Shared Transaction State		*state managed by open nesting, encapsulated within* `TransactionalMap`
	`Map key2lockers`	map from keys to set of lockers
	`Set sizeLockers`	set of size lockers
Local Transaction State		*state visible by the local thread*
	`Set keyLocks`	set of key locks held by the thread
	`Map storeBuffer`	map of keys to new values, special value for removed keys
	`int delta`	change in size due to changes in `storeBuffer`

Table 3. Summary of `TransactionalMap` state.

tected from seeing each other's effects. The second step is then to enforce these dependencies with semantically meaningful locks. In this section, we discuss the creation of the `TransactionalMap`, `TransactionalSortedMap`, and `TransactionalQueue` transactional collection classes.

3.1 TransactionalMap

Our `TransactionalMap` class allows concurrent access to a `Map` from multiple threads while allowing multiple operations from within a single thread to be treated as a single atomic transaction. `TransactionalMap` acts as a wrapper around existing `Map` implementations allowing the use of special purpose implementations to be used.

Determining Semantic Conflicts

We build classes such as `TransactionalMap` by determining which operations cannot be reordered without violating serializability. Our first step is to analyze the `Map` abstract data type to understand which operations commute under which conditions. We then use semantic locks to preserve serializability of non-commutative operations based on these conditions. To understand which `Map` operations can be reordered to build `TransactionalMap`, we performed a multi-step categorization of the operations as described below.

The first categorization of operations is between *primitive* or *derivative* methods. Primitive methods provide the fundamental operations of the data structure while the derivative methods are conveniences built on top of primitive methods. For example, operations such as `isEmpty` and `putAll` can be implemented using `size` and `put`, respectively, and need not be considered further in our analysis. In the case of `Map`, this categorization helps us reduce the dozens of available methods to those shown in the left column of Table 1.

Write / *Read*	put	remove
entrySet.iterator.hasNext	hasNext is false and put adds new lastKey	hasNext returns true about lastKey and remove takes away lastKey
entrySet.iterator.next	put adds key in iterated range	remove takes away key in iterated range
comparator		
subMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
headMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
tailMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
tailMap.iterator.hasNext	hasNext is false and put adds new lastKey	hasNext returns true about lastKey and remove takes away lastKey
lastKey	put adds a new lastKey	remove takes away the lastKey

Table 4. Semantic operational analysis of the `SortedMap` interface. This focuses on new and changed primitive operations relative to the `Map` interface in Table 1.

Methods	Read Lock	Write Conflict
Read Only		
entrySet.iterator.hasNext	last lock on false return value	
entrySet.iterator.next	range lock over iterated values, first lock	
comparator		
subMap.iterator.next	range lock over iterated values	
headMap.iterator.next	range lock over iterated values, first lock	
tailMap.iterator.next	range lock over iterated values	
tailMap.iterator.hasNext	last lock on false return value	
firstKey	first lock	
lastKey	last lock	
Write		
put	key lock on argument	key&range conflicts on argument first&last lock on endpoint change size conflict on increases
remove	key lock on argument	key&range conflicts on argument first&last lock on endpoint change size conflict on decreases

Table 5. Semantic locks for `SortedMap`. This focuses on new and changed primitive operations relative to the `Map` interface in Table 2.

Category	Field	Description
Commited Transactional State		*commited state visible to all transactions*
	`SortedMap sortedMap`	the underlying `SortedMap` instance
	`Comparator comparator`	read-only `Comparator` instance
Shared Transactional State		*state managed by open nesting, encapsulated within `TransactionalMap`*
	`Set firstLockers`	set of first key lockers
	`Set lastLockers`	set of last key lockers
	`Set rangeLockers`	set of last key lockers
Local Transactional State		*state visible by the local thread*
	`Set rangeLocks`	set of range locks held by the thread
	`SortedMap sortedStoreBuffer`	sorted map of keys to new values, special value for removed keys

Table 6. Summary of `TransactionalSortedMap` state. This focuses on the additions of state of the the `TransactionalMap` superclass in Table 3.

The second categorization is between *read-only* methods and those that `write` logical state of the `Map`. Since read-only operations always commute, the writing methods affect the serializability of each read method, so we focus our conflict detection efforts there. In Table 1, we list the read and write operations in the left column, showing when they conflict with the write operations in the `put` and `remove` columns.

The `put` and `remove` operations can conflict with methods that read keys, such as `containsKey`, `get`, and `entrySet.iterator.next`. Note that even the non-existence of a key, as determined by `containsKey`, conflicts with the addition of that key via `put`. Similarly, in cases where `put` and `remove` update the semantic size of the `Map`, these methods conflict with operations that reveal the semantic size, namely the `size` and `entrySet.iterator.hasNext`. `entrySet.iterator.hasNext` reveals the size indirectly since it allows someone to count the number of semantic entries in the `Map`. Typically this is used by transactions that enumerate the entire `Map`, which conflict with a transaction that adds or removes keys.

Implementing Semantic Locks

Up to this point, we have focused on analyzing the behavior of the `Map` abstract data type. Such analysis is largely general and can be used with a variety of implementation strategies. Now we shift gears to discuss how we used this analysis in our specific `TransactionalMap` class implementation. We discuss alternative implementation strategies in Section 5.1.

Our discussion in Section 2 concluded that we must release dependencies on data structure internals (using open nesting) to avoid unnecessary memory conflicts. To maintain correctness, we then use semantic locks to implement multi-level transactions, preserving the logical dependencies of the abstract data type.

In our analysis of `Map`, we found that reordering methods depended on two semantic properties: size and the key being operated on. While these choices are Map specific, other classes should have similar elements of abstract state.

Table 2 shows the conditions under which locks are taken during different operations. Read operations lock abstract state throughout the transaction. Write operations detect conflicts at commit time by examining the locks held by other transactions. If other transactions have read abstract state being written by the committing transaction, there is a conflict and the readers are aborted to maintain isolation. For example, a transaction that calls the `size` method acquires the size lock and would conflict with any committing transaction that changes the size (e.g., `put` or `get`).

Table 3 summarizes the internal state used to implement `TransactionalMap`. The `map` field is simply a reference to the wrapped `Map` instance containing the committed state of the map. Any read operations on the `map` field are protected by the appropriate key and size locks. These locks are implemented by the `key2lockers` and `sizeLockers` fields. These fields are shared so that transactions can detect conflicts with each other but encapsulated to prevent unstructured access to this potentially isolation-reducing data.

To maintain isolation, the effects of write operations are buffered locally in the current transaction. The `storeBuffer` field records the results of these write operations. Almost all read operations need to consult the `storeBuffer` to ensure they return the correct results with respect to the transaction's previous writes. The one exception is `size`, which instead consults the `delta` field providing the difference in size represented by the `storeBuffer` operations.

Commit and abort handlers are critical to the correct maintenance of transactional classes. When a transaction is aborted, a compensating transaction must be run to undo changes made by earlier open-nested transactions, in this case releasing semantic locks and clearing any locally buffered state. The `keyLocks` field locally stores locks to avoid explicitly enumerating `key2lockers` when performing this compensation. Commit handlers are used to perform semantic conflict detection as described above, to release the committing transaction's locks after it has completed, and to merge the locally buffered changes into the underlying data structure.

The owner of lock is the top-level transaction at the time of the read operation, not the open-nested transaction that actually performs the read. This is because the open-nested transaction will end soon, but we need to record that the outermost parent transaction needs to be aborted if a conflict is detected. Indeed, it is the handlers of the top-level transaction, whether successful or unsuccessful, that are responsible for releasing any locks taken on its behalf by its children.

One of the most complicated parts of `TransactionalMap` was the implementation of `Iterator` instances for the `entrySet`, `keySet`, and `values`. The iterators need to both enumerate the underlying map with modifications for new or deleted values from the `storeBuffer` and enumerate the `storeBuffer` for newly added keys. The iterator takes key locks as necessary as they are returned by the `next` methods as well as the size lock if `hasNext` indicates that the entire `Set` was enumerated.

3.2 TransactionalSortedMap

Our `TransactionalSortedMap` class extends `TransactionalMap` to provide concurrent atomic access by multiple non-conflicting readers and writers to implementations of the Java `SortedMap` interface. The `SortedMap` abstract data type extends `Map` by adding support for ordered iteration, minimum and maximum keys, and range-based sub-map views.

Determining Semantic Conflicts

The `SortedMap` interface extends the `Map` interface by both adding new methods for dealing with sorted keys, but also by defining the

Read \ *Write*	put	take	poll
peek	if peek returned null		
Write			
put			
take			
poll	if poll returned null		

Table 7. Semantic operational analysis of the `Channel` interface showing the conditions under which conflicts arise with write operations list on top.

Methods	**Read Lock**	**Write Conflict**
Read		
peek	if empty	
Write		
put		if now non-empty
take		
poll	if empty	

Table 8. Semantic locks for `Channel` describe empty locks that are taken when executing operation as well as lock based conflict detection that is done by writes at commit time.

semantics of existing methods such as `entrySet` to provide ordering. Mutable `SortedMap` views returned by `subMap`, `headMap`, and `tailMap` also has to be considered in our analysis. In Table 4, we perform a similar categorization of abstract data type operations as in the last section, focusing on the new primitive operations and on the operations with changed behavior such as `entrySet`. New operations that derivative, such as `firstKey`, are omitted.

The categorization shows that all of the new operations are read only. In addition to the key and size properties of `Map`, methods now also read ranges of keys as well as noting the first and last key of the `SortedMap`. The existing write operations `put` and `remove` are now updated to show their effects on ranges of keys as well as the endpoint keys. Specifically, a `put` or `remove` operation conflicts with any operation that reads a range of keys that includes the key argument of the `put` or `remove`. It's important to note that ranges are more that just a series of keys. For example, inserting a new key in one transaction that is within a range of keys iterated by another transaction would violate serializability if we did not detect the conflict. In addition to the range keys, `put` and `remove` can affect the first and last key by respectively adding or removing new minimum or maximum elements, thus conflicting with operations that either explicitly request these values with `firstKey` or `lastKey` or implicitly read these values through an iterator, including iterators of views.

Implementing Semantic Locks

Table 6 summarizes the extensions to the internal state of `TransactionalMap` used to implement `TransactionalSortedMap`. The `sortedMap` field is a `SortedMap`-typed version of the `map` field from `TransactionalMap`. The `comparator` field is used to compare keys either using the `Comparator` of the underlying `SortedMap` if one is provided or using `Comparable` if the `SortedMap` did not specify a `Comparator`. Note that the `comparator` is established during construction and thereafter is read only so no locks are required to protect its access.

The key-based locking of `TransactionalMap` is extended in `TransactionalSortedMap` by key range locking, provided by the `rangeLockers` field. As with key locks, writers must determine which subset of outstanding transactions conflict with their updates. We chose a simple `Set` to store the range locks, meaning updates to a key must enumerate the set to find matching ranges

Category	Field	Description
Commited State		*commited state visible to all transactions*
	`Queue queue`	the underlying `Queue` instance
Shared Transaction State		*state managed by open nesting, encapsulated within `TransactionalQueue`*
	`Set emptyLockers`	set of empty lockers
Local Transaction State		*state visible by the local thread*
	`List addBuffer`	list of locally added elements
	`List removeBuffer`	list of locally removed elements

Table 9. Summary of `TransactionalQueue` state.

for conflicts. An alternative would have been to use an interval tree to store the range locks, but the extra complexity and potential overhead seemed unnecessary for the common case. The endpoint-based locking of `TransactionalSortedMap` is provided by the `firstLockers` and `lastLockers` fields. Like size locking, and unlike key range locking, endpoint locking does not not require any search for conflicting transactions, since endpoint lockers are conflicting whenever the corresponding endpoint changes.

The local transaction state for `TransactionalSortedMap` consists primarily of the `rangeLocks` field which allows efficient enumeration of range locks for cleanup on commit or abort without enumerating the potentially larger global `rangeLockers` field. In addition, the `sortedStoreBuffer` provides a `SortedMap` reference to the `storeBuffer` field from `TransactionalMap` in order to provide ordered enumeration of local changes.

As with `TransactionalMap`, one of the more difficult parts of implementing `TransactionalSortedMap` was providing iteration. In order to provide proper ordering, iterators must simultaneously iterate through both the sortedStoreBuffer and the underlying sortedMap, while respecting ranges specified by views such as `subMap`, and taking endpoint locks as necessary.

3.3 TransactionalQueue

In the database world, SQL programs can request reduced isolation levels in order to gain more performance. Similarly, sometimes in transactional memory it is useful to selectively reduce isolation. One example is in creating a `TransactionalQueue`. The idea is inspired by a Delaunay mesh refinement application that takes work from a queue and may add new items during processing. Open-nested transactions could be used to avoid conflicts by immediately removing and adding work to the queue [19]. However, if transactions abort, the new work added to the queue is invalid, but may be impossible to recover since another transaction may have dequeued it. Our `TransactionalQueue` provides the necessary functionality by wrapping a `Queue` implementation with a `Channel` interface from the `util.concurrent` package [20].

Providing the simpler `Channel` interface lowers the design complexity by eliminating unnecessary `Queue` operations that do not make sense for a concurrent work queue, such as random access operations, instead only providing operations to enqueue and dequeue elements. To improve concurrency, we do not maintain strict ordering on the queue, so we have few semantic conflicts between transactions. As Table 7 shows, if transactions confine themselves to the common `put` and `take` operations, no semantic conflicts can ever occur. The only semantic conflict we check is if a transaction detects an empty `Queue` via a `null` result from `peek` or `poll`, then we will detect a conflict if another `put` or `offer` adds a new element.

Table 9 summarizes the internal state used to implement `TransactionalQueue`. The `queue` field holds the current committed state of in an underlying `Queue` instance. The `emptyLockers` field tracks which transactions have noticed when the queue is empty. The `addBuffer` field tracks new items that need to be added to the queue when the parent transaction commits while the `removeBuffer` tracks removed items that should be returned to the queue of the parent transaction aborts. While simple in construction compared to the fully serializable `TransactionalMap` and `TransactionalSortedMap` classes, the Delaunay example shows the benefits of having a transactional aware queue that allows multiple operations within a single transaction.

4. Semantics for Transactional Collection Classes

In this section we discuss the functionality necessary to implement transactional collection classes. We refer to these mechanisms and their use as *transactional semantics*. While all transactional memory systems offer an interface for denoting transactional regions, some of them already provide elements of the transactional semantics we identify.

Nested transactions: open and closed

Some systems implement *flat* closed-nested transactions: the child simply runs as part of the parent without support for partial rollback. However to reduce lost work due to unnecessary conflicts, our implementation needs partial rollback of the commit handlers run as closed-nested transactions. That way, any conflicts during update of the underlying data structure will only roll back the commit handler and not the entire parent.

Open nesting is probably the most significant requirement for semantic concurrency control. It is the enabling feature that allows transactions to create semantic locks without retaining memory dependencies that will lead to unnecessary conflicts. However, while open-nested transactions are a necessary feature for supporting semantic concurrency control, they are not sufficient without some way of cleaning up semantic locks when the overall transaction finishes — this is the purpose of commit and abort handlers.

Commit and abort handlers

Commit and abort handlers allow code to run on the event of successful or unsuccessful transaction outcome, respectively. Transactional collection classes use these handlers to perform the required semantic operations for commit and abort, typically writing the new state on commit, performing compensation on abort, and releasing semantic locks in both cases.

Commit handlers typically run in a closed-nested transaction so that any memory conflicts detected during their updates to global state do not cause the re-execution of the parent. Handlers execute at the end of the parent transaction so it has visibility into the parent's state. This is useful for cleaning up any thread-local values.

Abort handlers typically run in an open-nested transaction. As with commit handlers, they are nested within the parent so they can access the parent's state before it is rolled back. Open nesting allows the abort handler to undo any changes performed by the parent's open-nested transactions; otherwise, any work done by the abort handler would simply be rolled back along with the parent.

When a commit or abort handler is registered, it is associated with the current level of nesting. If the nested transaction is aborted, the handlers are simply discarded without executing — rollback should clear the state associated with the handlers. If the nested transaction commits, the handlers are associated with the parent so necessary updates/compensation will happen when the parent completes/aborts.

Discarding newly registered handlers prevents a handler from running in unexpected situations. Since a conflict could be detected at any time, an abort handler could be invoked at any point in the execution of a transaction. Conceptually, this is very similar to the problem of reentrancy of Unix signal handlers: it is difficult to insure that data structure invariants hold. With signal handlers, the approach is usually to do very little within handlers except to note that the signal happened, letting the main flow of control in the program address the issue. Fortunately, nested transactions and encapsulation can provide more guarantees about the state of objects. If the only updates to the encapsulated state, such as the local tables and store buffers, are made with open-nested transactions, then we can be sure that when an abort handler runs, these encapsulated data structures are in a known state.

Discarding newly registered handlers on abort interacts with using abort handlers for compensation of non-transactional operations. However, these operations should not have been performed during the body of the transaction but rather during commit handlers. While logically this makes sense for output operations deferred to the end of the transaction, it also works for input operations as they can be performed in the commit handler of an open-nested transaction that registers an abort handler to push back input as needed. While handlers are not a general solution for handling all cases of non-transactional operations, these semantics cover two frequently cited examples of using handlers to mix I/O and transactions.

Some systems use two-phase commit as part of executing commit handlers. Two-phase commit breaks a transaction commit into two parts: *validation* and *commit*. After validation is completed, the transaction is assured that it will be able to commit. Typically commit handlers are run in the commit phase after validation. This guarantees that any non-transactional action such as I/O do no need to worry that the parent will get violated after an irreversible action is performed. Note that for our uses of semantic concurrency control, we are not performing any non-transactional operations, only updates to data structures in memory, so that two phase commit is not strictly required, although its presence is not a problem.

Program-directed transaction abort

Transactional memory systems can automatically abort transactions with serializability conflicts. Some systems provide an interface for transactions to abort themselves, perhaps if they detect a problem with the consistency property of ACID transactions. Semantic concurrency control requires the additional ability for one transaction to abort another when semantic-level transactional conflicts are detected. Specifically for our proposal, an open-nested transaction needs a way to request a reference to its top-level transaction than can be stored as the owner of a lock. Later if another transaction detects a conflict with that lock, the transaction reference can be used to abort the conflicting transaction.

5. Serializability Guidelines

The most difficult part of semantic concurrency control is analyzing the abstract data type to determine the rules for commutative operations and determining a set of semantic locks to properly preserve commutativity. However, once this is done, we found the actual implementation of semantic concurrency control via multi-level transactions was fairly straightforward using a simple set of rules:

- The underlying state of the data structure should only be read within an open-nested transaction that also takes the appropriate semantic locks. This ensures that the parent transaction contains no read state on the underlying state that could cause memory-level conflicts.
- The underlying state of the data structure should only be written by a closed-nested transaction in a commit handler. This preserves isolation since semantic changes are only made globally visible when the parent transaction commits.
- Because write operations should not modify the underlying data structure, write operations need to store their state in a transaction-local buffer. If semantic locks are necessary because the write operation logically includes a read operation as well, the locks should be taken in an open-nested transaction, as above.
- The abort handler should clear any changes made with open-nested transactions including releasing semantic locks and clearing any thread-local buffers. Only one abort handler is necessary and it should be registered by the first open-nested transaction to commit.
- The commit handler should apply the buffered changes to the underlying data structure. As it applies the changes, it should check for conflicting semantic locks for the operations it is performing. After it has applied the changes, it follows the behavior of the abort handler, ensuring that the buffer is cleared and that semantic locks are released. As with the abort handler, only a single commit handler is needed, registered on the first write operation.

Note that if we want reduced isolation, we typically violated the second rule by allowing writes to the underlying state from within open-nested transactions. For example, in `TransactionalQueue`, `take` operations removed objects from the underlying queue without acquiring a lock.

5.1 Discussion

Alternatives to optimistic concurrency control

Detecting conflicting changes at commit time is known as *optimistic concurrency control*. Another approach is to detect conflicts as soon as possible (*pessimistic concurrency control*). In our system, write operations could detect conflicting semantic locks when the operation is first performed, instead of waiting until commit. A contention management policy can then be used to decide how to proceed. One approach is to have the conflicting write operation wait for the other transaction to complete. However, this leads to the usual problems with locks, such as deadlock. The downside to optimistic concurrency control is that it can suffer from livelock since long-running transactions may be continuously rolled back by shorter ones. Here again, contention management policies can be applied to give repeatedly violated transactions priority. A discussion of contention management in transactional memory systems can be found in [11]. The choice of optimistic concurrency control for semantic-level transactions is independent of the underlying concurrency control in the transactional memory system.

Redo versus undo logging

Our approach to buffering changes and replaying them at commit time is a form of *redo logging*, so called because we redo the work of the operations in the local buffer on the global state. The alternative is *undo logging*, where we update the global state in place. If there are no conflicts, the undo log is simply dropped at commit time. If there is a conflict and the transaction needs to abort, the undo log can be used to perform the compensating actions to roll back changes made to the global state by the aborting transaction. We choose redo logging because it is a better fit to optimistic concurrency control, since undo logging requires early conflict detection since only one writer can be allowed to update a piece of semantic state in place at a time. Note that the choice of redo versus undo logging for semantic-level transactions is independent of the underlying logging used by the transactional memory system.

Single versus multiple handlers

Our approach uses one commit and one abort handler per parent transaction. These handlers know how to walk the underlying lock and buffer structures to perform the necessary work on behalf of all previous operations to the data structure. An alternative is for each operation to provide its own independent handlers.

Moss extends this alternative by proposing that each abort handler should run in the memory context that was present at the end of the child transaction in which it was registered, interleaved with the undoing of the intervening memory transactions [25]. For example, suppose we have a series of operations that make up a parent transaction $AXBYC$, with A, B, and C being memory operations and X and Y being open-nested transactions. Moss suggests that to abort at the end of C, we should logically perform the inverse actions $C^{-1}Y^{-1}B^{-1}X^{-1}A^{-1}$ to abort. Logically, this consists of rolling back the memory operations of C, followed by running the abort handler for Y, followed by rolling back B, then running the abort handler for X, and finally rolling back the memory operations of A.

We found this extra complexity to be unnecessary for our implementation of semantic concurrency control. Moss's semantics aim to guarantee that the handler will always run in a well-defined context known to the handler at its point of registration. However, our guidelines give handlers similar guarantees since object-oriented encapsulation ensures that only the transaction that registered the handler can make updates to the state that the handler will later access on abort.

Alternative semantic locks

In our initial categorization of `Map` methods into primitive and derivative operations, we considered several methods as primitive that semantically speaking are strictly derivative if one focuses on correctness and not performance. After all, it is possible to build a writable `Map` by subclassing `AbstractMap` and implementing only the `entrySet` and `put` methods, although such an association list style implementation would need to iterate the `entrySet` on `get` operations, taking locks on many keys unnecessarily, compromising concurrency while maintaining correctness.

While we might seem to be contradicting our own methodology, in fact in one considers the expected running time of operations to be part of the semantics of an abstract data type, we are consistent. Certainly if one is happy to have a linear cost for `Map` retrievals, then it is fine to treat `get` as a derived operations. However, if you expect to have close to constant cost for a `Map` and logarithmic for a `SortedMap`, you need to follow the more typical approach of `HashMap` and `TreeMap` and consider other methods such as `containsKey`, `get`, `size`, and `remove` as primitive operations, which avoids the need to iterate the `entrySet` for these operations.

While the semantic locks derived from the primitive methods in Table 1 and Table 4 preserve isolation and serializability, they still do not allow the optimal concurrency possible. One limitation is making `isEmpty` a derivative method based on `size`, resulting in `isEmpty` taking a size lock. To see why this is a problem, consider two transactions running this code:

```
if (!map.isEmpty()) map.put(key, value);
```

These transactions should commute as long as they add different keys, but the current implementation will cause one to abort because of the size lock. However, taking the size lock is necessary for the similar case of two transactions running:

```
if (map.isEmpty()) map.put(key, value);
```

These `put` operations should not commute because a serial ordering would require that only one would find an empty map. The solution is to make `isEmpty` a primitive operation with its own separate semantic lock that is violated only when the size changes to or from zero.

Note that we have focused on the third categorization for the primitive methods in the interest of space, but that logically this categorization is important for the derivative methods as well. This can expose unexpected concurrency limitations. For example, a straightforward implementation of `entrySet.remove` might use `size` to determine if the `remove` operation actually found a matching key, which is used to calculate the boolean return value. This would add an unnecessary dependency on the size, which could cause unnecessary conflicts if others concurrently added or removed other keys.

Extensions to `java.util.Map`

C++ programmers often use idioms like `if (map.size()) ...` instead of `if (!map.empty()) ...`, arguably because it's easy for a reader to miss the negation when reviewing code. Similarly, Java programmers frequently use `if (map.size() == 0) ...` instead of `isEmpty`. However, as noted in the previous discussion, using `size` instead of `isEmpty` unnecessarily restricts concurrency.

Similar problems exist for `Map` methods that reveal more information than strictly necessary. For example, since the write operations `put` and `remove` return the old value for a key, they effectively read the key as well. If this return value is unused, this is an unnecessary limitation of semantic concurrency. To be specific, there is no reason that two transactions that write to the same key need to be ordered in any way. For example, it's perfectly acceptable for two transactions to do this:

```
map.put("LastModified", new Date());
```

These transactions can commit in any order so long as they do not read the "LastModified" key.

The solution is to offer alternative variants to methods such as `put` and `remove` that do not reveal unnecessary information, allowing the caller to decide which is appropriate.

TransactionalSet and TransactionalSortedSet

We did not discuss `TransactionalSet` and `TransactionalSortedSet` classes here because they can be built as simple wrappers around the `TransactionalMap` and `TransactionalSortedMap`, respectively, as has been done similarly for `ConcurrentHashSet` implementations built on top of `ConcurrentHashMap` and even `HashSet` implementations around `HashMap` as found in [7].

Leaking uncommitted data

While our guidelines prevent leaking of uncommitted data between transactions using the same transactional collection class, values used within the class's semantic locks, such as keys or range endpoints, can be visible to the other open-nested transactions operating on the instance. For example, if a newly allocated string is used as a key name in a `TransactionalMap`, the `key2lockers` table would have an entry pointing to an object that is only initialized within the adding transaction. However, if another transaction adds another key that hashes to the same bucket, the table will call `Object.equals` to compare the new key to the existing key, which is uninitialized from this second transaction's point of view.

In [25], Moss proposes making object allocation and type initialization an open-nested transaction, so at least access to this uncommitted object will not violate Java type safety. However, the constructor part of object allocation cannot be safely made part of an open-nested transaction because it could perform arbitrary oper-

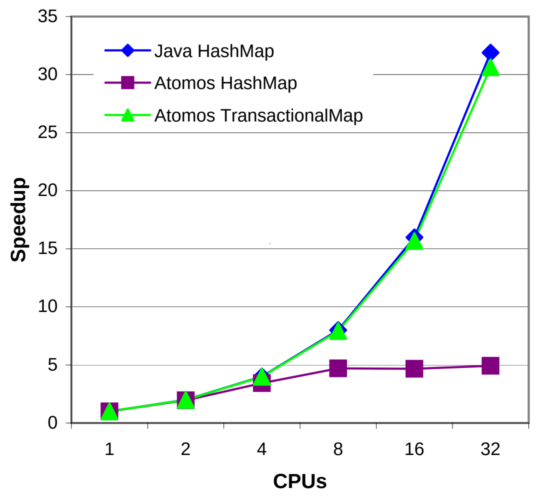

Figure 1. TestMap results show that Atomos can achieve the scalability of Java when the concurrently accessed `HashMap` is wrapped in a `TransactionalMap`.

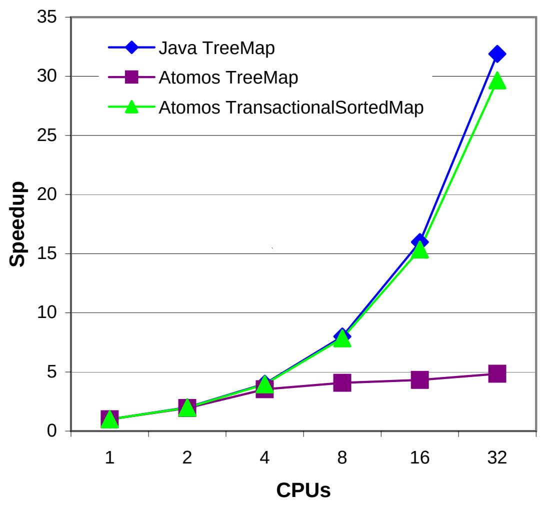

Figure 2. TestSortedMap results parallel TestMap showing that `TransactionalSortedMap` provides similar benefits to a concurrently accessed `TreeMap`.

ations that might require compensation. Moss notes that for some common key classes such as String, it is safe and even desirable to run the constructor as part of open-nested allocation, but this is not a general solution.

Our proposal is to not directly insert such potentially uncommitted objects into semantic locking tables but instead insert copies. One approach would be to use existing mechanisms such as `Object.clone` or `Serializable` to make a copy, similar to what is proposed by Harris in [12], which uses `Serializable` to copy selected state out of transactions that are about to be aborted. Alternatively, a new interface could be used to request a committed key from an object, allowing it to make a selective copy of a subset of identifying state, rather than the whole object like `clone` or `Serializable`, perhaps simply returning an already committed key.

6. Evaluation

To evaluate our transactional collection classes we use variants of a common transactional memory micro-benchmark as well as a custom version of SPECjbb2000 designed to have higher contention. We evaluate both traditional Java and transactional Atomos versions. The results focus on benchmark execution time, skipping virtual machine startup. The single-processor Java version is used as the baseline for calculating speedup.

6.1 Environment

Atomos is a transactional programming language described in [2]. It provides the necessary support for the transactional semantics described in the previous section, allowing it to be used to evaluate our transactional collection classes. The Java programs and Atomos environments are both based on the Jikes Research Virtual Machine (JikesRVM), version 2.3.4.

JikesRVM was run with an execution-driven simulator of a PowerPC CMP system that implements the TCC continuous transaction architecture for evaluating Atomos as well as MESI snoopy cache coherence for evaluating Java locking [22]. The simulator was extended with support for closed- and open-nested transactions as well as commit and abort handlers as described in [21]. All instructions, except loads and stores, have a CPI of 1.0. The memory system models the timing of the L1 caches, the shared L2 cache, and buses. All contention and queuing for accesses to caches and buses is accurately modeled.

While we perform our experiments with a specific language, virtual machine, and HTM implementation, we believe that the observations and conclusions apply to other hardware transactional memory systems as well.

6.2 Map and SortedMap Benchmarks

TestMap is a micro-benchmark based on a description in [1] that performs multi-threaded access to a single `Map` instance. Threads perform a mixture of operations with a breakdown of 80% lookups, 10% insertions, and 10% removals. To emulate access to the `Map` from within long-running transactions, each operation is surrounded by computation. There should be little semantic contention in this benchmark but frequent memory contention within the `Map` implementation such as the internal size field.

Figure 1 summarizes our results for TestMap. As expected, Java with `HashMap` shows near linear scalability because the lock is only held for a small time relative to the surrounding computation. The Atomos `HashMap` result shows what happens when multiple threads try to simultaneously access the `Map` instance, with scalability limited as the number of processors increases because of contention on the `HashMap` size field. Atomos results with a `TransactionalMap` wrapped around the `HashMap` show how scalability can be regained when unnecessary memory conflicts on the size field are eliminated.

TestSortedMap is a variant of TestMap that replaces lookup operations using `Map.get` with a range lookup using `SortedMap.subMap`, taking the median key from the returned range. As with TestMap, there is little semantic contention as the ranges are relatively small and serve just to ensure there are not excessive overheads from the range locking implementation.

Figure 2 shows that Java with a `SortedMap` scales linearly as expected. Atomos with a plain `TreeMap` fails to scale because of non-semantic conflicts due to internal operations such as red-black tree balancing. Finally, Atomos with a `TransactionalSortedMap` wrapped around a `TreeMap` instance regains the scalability of the Java version.

TestCompound is a variant of TestMap that composes two operations separated by some computation. The results are shown in Figure 3. In the Java version, a coarse grained lock is used to ensure that two operations act as a single compound operation. For Atomos, the entire loop body, including other computation before and

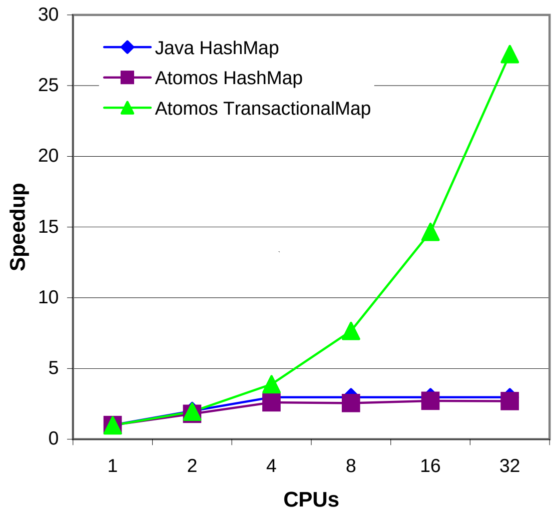

Figure 3. TestCompound results shows that Java scalability is limited by use a coarse grained lock to protect a compound operation which scales as a single Atomos transaction.

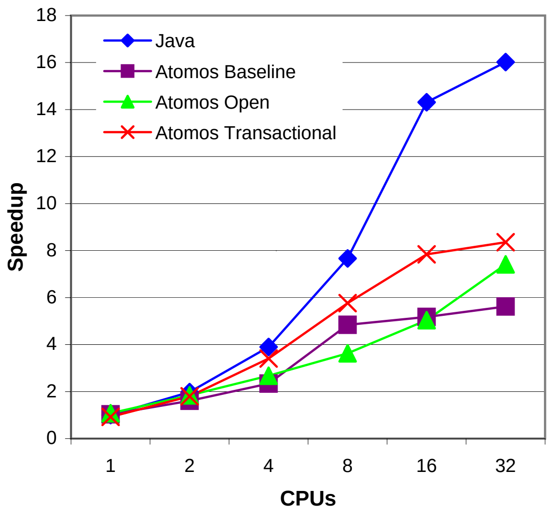

Figure 4. SPECjbb2000 results in a high-contention configuration caused by sharing a single warehouse.

after the compound operation, is performed as a single transaction. In this case, the Java version scales poorly since a single lock is held during the computation between the two operations, with little difference to the Atomos `HashMap` result. However, the `TransactionalMap` result shows that transactional collection classes can provide both composable operations and concurrency.

6.3 SPECjbb2000

SPECjbb2000 [29] is an embarrassingly parallel Java program with little inherent contention. As a benchmark, it focuses more on ensuring that the underlying virtual machine and supporting system software are scalable. There are a few shared fields and data structures in SPECjbb2000, each protected by `synchronized` critical regions in the Java version. Each application thread is assigned an independent warehouse to service TPC-C style requests, with only a 1% chance of contention from inter-warehouse requests. Previous results have shown that a transactional version of SPECjbb2000 can scale as well as Java [2].

In order to make scaling performance of SPECjbb2000 more challenging, we created a version that uses a single warehouse for all operations. In addition, for our Atomos version of SPECjbb2000, we do not use the Java critical regions to create transactions. Instead we turn each of five TPC-C operations into single atomic transactions. This is to emulate a first step baseline parallelization by a novice parallel programmer. The correctness of this parallelization is easy to reason about because all the parallel code excluding the thread startup and loop setup is now executed within transactions. We changed both Java and Atomos versions to use `java.util` collection classes in place of the original binary tree implementation, following the pattern of SPECjbb2005.

Figure 4 shows the results for our modified version of SPECjbb2000. First, we see that our modifications to use a single warehouse significantly impact the scalability of the Java version, which usually would achieve nearly linear speedup on 32 processors. The *Atomos Baseline* version with large transactions suffers even further from a variety of conflicting memory operations.

Using techniques described in [3], we were able to identify several global counters such as the `District.nextOrder` ID generator as the main sources of lost work due to conflicts. By wrapping reads and writes to the these counters in open-nested transactions, we were able to preserve the counter semantics while reducing lost work as shown by the *Atomos Open* result.

Additional conflict analysis identified three shared `Map` instances that were frequent sources of conflicts: `Warehouse.historyTable`, `District.orderTable`, and `District.newOrderTable`. When these were wrapped with `TransactionalMap` and `TransactionalSortedMap` as appropriate, we achieved the *Atomos Transactional* result shown in the figure.

The use of simple open-nested counters and transactional collection classes yielded a reasonable speedup for little effort. A final analysis revealed more opportunities for improvement, such as splitting transactions and relaxing strict isolation.

7. Conclusions

We believe the true promise of transactional memory is making parallel programming easier. This means we should evaluate transactional memory systems for their ability to run long transactions scalably, not by their ability to hold their own against code with very short transactions based on fine-grained locking parallelizations. Even if a system scales well with embarrassingly parallel applications or fine-grained transactions, it is also important to show scalability for applications with long-running transactions accessing shared data, since the ultimate goal is to make parallel programming easier by giving the programmer the perfomance of fine-grained locking while only using coarse-grained transactions.

We have shown that semantic currency control allows us to concurrently access data structures while preserving the isolation, and therefore serializability, properties of transactions. We described how we built `TransactionalMap` and `TransactionalSortedMap` collection classes for this purpose using the concept of multi-level transactions built upon open nesting. We also showed, with our `TransactionalQueue`, how these ideas can be used to break the isolation property in structured ways when it is desired to trade serializability for performance. We expect that such reusable collection classes would be part of the standard library of a transactional programming language such as Atomos.

While standard library classes are convenient for many programmers, we have shown a straightforward operational analysis and implementation guidelines that allow programmers to safely design their own concurrent classes, in cases where they need to create new or augment existing data structures.

Finally, we hope that our evaluation will convince the implementers of both hardware and software transactional memory systems of the benefits and need for rich transactional semantics. As the database community has shown, there is a lot more to transactional systems than simple atomicity.

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. Additional support was also available through NSF grant 0444470 and through the MARCO Focus Center for Circuit & System Solutions (C2S2), under contract 2003-CT-888. Brian D. Carlstrom is supported by an Intel Foundation PhD Fellowship.

References

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[2] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, New York, NY, USA, June 2006. ACM Press.

[3] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A Transactional Application Profiling Environment. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 199–208. June 2005.

[4] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, February 2006.

[5] D. Dice and N. Shavit. What really makes transactions faster? In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[6] R. Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051,, Intel Research Cambridge, 2005.

[7] Free Software Foundation, *GNU Classpath 0.18*. `http://www.gnu.org/software/classpath/`, 2005.

[8] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM Press.

[9] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154. IEEE Computer Society, 1981.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[12] T. Harris. Exceptions and side-effects in atomic blocks. In *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[14] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[16] *Java Specification Request (JSR) 166: Concurrency Utilities*, September 2004.

[17] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *Conference Record of Hot Chips 15 Symposium*, Stanford, CA, August 2003.

[18] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE MICRO Magazine*, 25(2):21–29, March–April 2005.

[19] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, June 2006.

[20] D. Lea. *package util.concurrent*. `http://gee.cs.oswego.edu/dl`, May 2004.

[21] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, June 2006. IEEE Computer Society.

[22] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

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

[24] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, April 1981.

[25] J. E. B. Moss. Open Nested Transactions: Semantics and Support. In *Poster at the 4th Workshop on Memory Performance Issues (WMPI-2006)*. February 2006.

[26] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 72–83, New York, NY, USA, 1986. ACM Press.

[27] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, August 1995.

[29] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.

[30] I. L. Trager. Trends in systems aspects of database management. In *Proceedings of the 2nd International Conference on Databases*. Wiley & Sons, 1983.

[31] W. Weihl and B. Liskov. Specification and implementation of resilient, atomic data types. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 53–64, New York, NY, USA, 1983. ACM Press.

[32] G. Weikum and H.-J. Schek. Architectural issues of transaction management in multi-layered systems. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 454–465, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[33] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.