Reusable Concurrent Data Types

Vincent Gramoli[1] and Rachid Guerraoui[2]

[1] NICTA and University of Sydney
vincent.gramoli@sydney.edu.au
[2] EPFL
rachid.guerraoui@epfl.ch

Abstract. This paper contributes to address the fundamental challenge of building Concurrent Data Types (CDT) that are reusable and scalable at the same time. We do so by proposing the abstraction of Polymorphic Transactions (PT): a new programming abstraction that offers different compatible transactions that can run concurrently in the same application.

We outline the commonality of the problem in various object-oriented languages and implement PT and a reusable package in Java. With PT, annotating sequential ADTs guarantee novice programmers to obtain an atomic and deadlock-free CDT and let an advanced programmer leverage the application semantics to get higher performance.

We compare our polymorphic synchronization against transaction-based, lock-based and lock-free synchronizations on SPARC and x86-64 architectures and we integrate our methodology to a travel reservation benchmark. Although our reusable CDTs are sometimes less efficient than non-composable handcrafted CDTs from the JDK, they outperform all reusable Java CDTs.

1 Introduction

Abstract data types (ADTs) have shown to be instrumental in making sequential programs reusable [1]. ADTs promote (a) *extensibility* when an ADT is specialized through, for example, inheritance by overriding or adding new methods, and (b) *composability* when two ADTs are combined into another ADT whose methods invoke the original ones. Key to this reusability is that there is no need to know the internals of an ADT to reuse it: its interface suffices. With the latest technology development of multi-core architectures many programs are expected to scale with a large number of cores: ADTs need thus to be shared by many threads.

Unfortunately, most ADTs that export shared methods, often called *Concurrent Data Types (CDTs)*, are not reusable: the programmer can hardly build upon them. For example, programmers cannot reuse the popular concurrent data types of C++, Java and C# libraries. CDTs typically export a set of methods, guaranteeing that, even if invoked concurrently, each of these methods always appears as if it was executed in sequence. This property, known as *atomicity* (or linearizability [2]), lets the programmer reason in terms of sequential accesses. However, atomicity is generally not preserved under extension or composition, hence annihilating reusability.

Basically, CDTs are synchronized using either lock-based (i.e., mutual exclusion) or lock-free primitives (e.g., compare-and-swap). On the one hand, CDTs that rely

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 182–206, 2014.

on locks have limited composability as a user could accidentally write two composite methods that deadlock when calling in different order two existing methods that require distinct locks. The same CDTs might not be extensible either as adding a new method may require to know the lock granularity used by existing methods. On the other hand, lock-free CDTs relying on hardware primitives can generally modify only one or two memory words atomically, requiring the user to precisely identify these words before obtaining a scalable and atomic composite method. Knowing these internals may, however, not even help extending lock-free CDTs as we will describe in Section 2.

Some synchronization schemes do enable reusability, yet their performance does not scale with concurrency. Typically, *Transactional Memory (TM)* systems ensure that within a sequence of shared memory reads/writes, all execute atomically (the transaction *commits*) or none of them execute (the transaction *aborts*) [3,4]. One can exploit TM to write an atomic CDT easily: it suffices to (a) write the bare sequential code of the ADT and then (b) to encapsulate each of the methods of the resulting ADT into a transaction. Transactional methods commit only if their execution is equivalent to a serial one. TMs typically provide composability [5] as a new composite operation encapsulated in a transaction can invoke multiple existing methods from a (transactional) CDT. Also, specific transactions facilitate extensibility by preventing anomalies when inheriting from an existing CDT [6]. Nevertheless, classic transactions are overly conservative and clearly hamper scalability simply because they cannot exploit the application semantics [7,8,9,10,11,12].

In light of this lack of scalability, expert programmers would implement handcrafted libraries whose semantics is difficult to understand to say the least: instead of being simply equivalent to a sequential execution (or atomic), an iteration over a CDT would typically return different results depending on the current status of concurrent updates of the same CDT. This strategy clearly promotes scalability while preventing a programmer, who ignores the underlying implementation details, from reusing the abstraction. Built-in C++ thread building block library, java.util.concurrent package and C# System libraries all adopt this strategy, hence limiting the ability for novices to write concurrent code in main object-oriented languages.

In this paper, we propose the *Polymorphic Transaction* (PT) methodology, which helps write concurrent programs that are both scalable and reusable. Its main novelty is not in providing a novel transaction semantics but in combining multiple of them to

Table 1. The use-cases in which we applied the PT methodology

Use-cases of the PT methodology	Data structure	Type	Annotated methods	Non-protected methods	Total
ReusableLinkedQueue	Linked list	Queue	13	2	15
ReusableVector	Vector	Collection	37	11	48
ReusableLinkedListSortedSet	Linked list	Set	11	4	15
ReusableHashMap	Hash table	Map	11	3	14
ReusableSkipListSet	Skip list	Set	11	4	15
Vacation	Red-black trees	Database	3	88	91
Total			86	112	198

scale to high levels of parallelism as they let advanced programmers exploit the application semantics. The PT methodology achieves better scalability than classic TM systems because it ensures the atomicity of the CDT operations but not of their read/write sequences. It also retains the appealing simplicity of TM systems as novice programmers obtain a safe (but less efficient) concurrent program if they ignore these semantics. In summary, it gives a framework for all programmers to write software pieces that combine with one another. To illustrate the performance potential of the PT methodology, we implemented (a) the *polymorphic* software transactional memory (PSTM), (b) on top of which we built a Java package of reusable CDTs that we use as a new TM benchmark suite on x86-64 and SPARC architectures, (c) we compared this library to the JDK (incuding java.util.concurrent) and (d) we integrated our solution to the STAMP travel reservation application, called vacation [13].

In contrast with lock-based and lock-free libraries, our library is reusable, thereby simplifying the life of concurrent programmers. In fact, we prove that our semantics combine with each other which translates into the composability and extensibility of our library as opposed to mainstream Java, C++ and C# concurrent libraries. To write an atomic (linearizable) CDT, the programmer writes a semantically equivalent bare sequential ADT and annotates each of its methods with one of the existing transaction forms without the need of altering the sequential code. To reuse existing CDTs, the programmer can either (a) compose these CDTs by invoking their methods in a method annotated with one existing transaction form or (b) extend these CDTs by inheriting from them and adding new methods annotated with one of the transaction forms. If the form of the annotation is omitted then the default form guarantees atomicity regardless of the application semantics. The four forms of PSTM, detailed in Section 3, are as follows:

- **Hand-over-hand:** A form of transaction that allows update methods to run concurrently. It builds upon a locking technique where each accessed location remains protected until the next location(s) within the same sequence gets protected. This technique is known as chain-locking, lock-coupling, or hand-over-hand locking [14]. As opposed to hand-over-hand locking, a hand-over-hand transaction may abort and release all its locks rather than blocking, thus being deadlock-free. (Hand-over-hand transactions guarantee elastic-opacity [9].)
- **Snapshot:** A form of transaction that allows read-only methods to run concurrently with updates. This form exploits multiversion concurrency control [15] to provide *snapshot isolation*, a property of production database systems that allows reads to execute at a different time from writes. Snapshot isolated transactions are prone to the write-skew problem when they concurrently read a set of data and later update disjoint subsets of these data, however, our form applies exclusively to read-only methods and guarantees atomicity.
- **Opacity:** the default form of transaction. Similar to strict-serializability targeted by database systems, opacity guarantees that transactions execute as if all their accesses were executed at some indivisible point in time (serializability) between the time they are invoked and the time they return (strictness). In contrast with database transactions, opaque transactions are guaranteed to never observe an inconsistent state of the system (even transiently) be they doomed to abort or still pending [16].

– **Irrevocability:** The form of a transaction that never aborts [17]. This form can be used to enforce that an atomic series of accesses executes exactly once. It is typically useful for executing I/O operations or invoking legacy code that cannot be rolled back, however, this form should be avoided when possible as it prevents transactions from executing concurrently.

A novel aspect of this work is to allow several transactional forms in the same application hence raising a new interesting *compatibility* challenge: guaranteeing that methods synchronized with different semantics do not affect the semantics of each other when accessing the same mutable data concurrently. For example, consider a hand-over-hand transaction, t_h, reading x before a concurrent opaque transaction, t_o, writes x. This write-after-read (WAR) conflict would typically be detected by t_o but ignored by t_h. Upon writing and detecting the conflict, if t_o resolves the conflict by aborting or delaying one of the two transactions, then concurrency would be suboptimal. Conversely, if t_o ignores the conflict, it may violate its semantics by committing: if say a later conflict on y requires that t_o be serialized before t_h. To cope with this, we prevent a WAR conflict from being resolved eagerly by the transaction that conflicts by writing, instead it is always resolved by the transaction that conflicts by reading (regardless of its form). This is described in Section 4 along with the resolution of write-after-write (WAW) and read-after-write (RAW) conflicts.

To integrate our methodology in the Java programming language, we extended the Deuce [18] bytecode instrumentation framework, so that synchronizing a bare sequential method simply consists of annotating it with either a hand-over-hand, a snapshot, an opaque or an irrevocable transaction. As detailed in Section 5, the produced bytecode is automatically instrumented so that shared reads/writes get redirected to the transactional reads/writes of the appropriate form featured by PSTM. We only annotated few methods in our benchmarks (cf. Table 1): all methods they call are automatically instrumented. We compared our reusable package to the JDK packages. First, we devised reusable CDTs using specific but restrictive techniques from the JDK like java.util.Collections.synchronizedSet or java.util.concurrent.copyOnWriteArraySet. Note that we could have also used our own implementation of a universal construction [19] to achieve similar results. Second, we tested mainstream non-reusable CDTs like the lock-based java.util.Vector or the lock-free java.util.concurrent.ConcurrentLinkedQueue [20].

While our implementation could benefit from recent speculative hardware instructions, even in its software form, the PT methodology helps improving significantly the performance of existing reusable techniques from the JDK (2.4× speedup). We also tested as a baseline the performance of non-reusable but well-engineered JDK CDTs and we observed great differences: while our CDTs could, in some executions, speedup the performance of the non-reusable JDK CDTs by 4×, our experiments also outline circumstances where reusability comes at a cost. All these experimental results are reported in Section 6.

Finally, we discuss the related work in Section 7 and conclude in Section 8.

2 Overview

Most concurrent object-oriented libraries trade reusability off for efficiency. We distinguish their two reusability limitations, namely extensibility and composability issues, and describe how the PT methodology addresses them.

2.1 Extensibility

Illustrating the issue. In Java, the ConcurrentLinkedQueue type of the JDK 7 exports an inconsistent size method. The problem comes from the fact that this CDT aims at implementing the lock-free algorithm from Michael and Scott designed to provide efficient offer (i.e., push) and poll (i.e., pop) [20] but aims also at implementing the Collection interface including a size method for a neat integration in the Java API. On the one hand, a size method is useful to count the number of elements comprised in this collection: although size remains optional, various Collection CDTs do provide it. On the other hand, the algorithm of Michael and Scott was optimized to export deadlock-free offer and poll without aiming at supporting a size method or allowing extensibility.

The problem of extending the Michael and Scott's algorithm with a size, which could access concurrently the same data as offer and poll, is far from being trivial, precisely due to the way the algorithm was originally proposed. In short, the algorithm was made deadlock-free by relying exclusively on compare-and-swap for synchronization. Comparing-and-swapping versions of the data structure to compute the size would annihilate effective concurrency while using locks to protect the data structure would not prevent the offer and poll from concurrently updating the structure. This lack of extensibility, which is inherent to the synchronization used, led expert programmers to implement a non-atomic size method.

Specifically, this size consists of traversing the underlying linked list from the head to the tail while elements are pushed at the head and popped at the tail. Assume that some elements are moved from the tail to the head, one after the other, so that the size s changes by ± 1. As the size method does not protect the head and the tail of the queue, it simply ignores any of these moved elements and returns an incorrect value way smaller than $s-1$. Precisely because predicting the outcomes of this size requires to understand the implementation internals, the resulting CDT is not reusable.

We reported this ConcurrentLinkedQueue issue to the JSR166 expert group. Following up our report, this unexpected behavior has been warned in the documentation of the class ConcurrentLinkedQueue on the JSR166 site since revision 1.54 and the issue is still present in the JDK 7. Since then other researchers unaware of this warning observed the same problem [21]. This size problem simply illustrates the more general lack of extensibility. One may think of using ArrayBlockingQueue to obtain a correct size that returns the current value of a counter, however, such a size implementation requires to modify all insertion and removal methods to make them adjust the counter. Apart from the size example, a programmer would have similar problems as soon as she tries to extend these CDTs with, for example, a sum method.

The PT solution. Figure 1 illustrates how to exploit the PT methodology to cope with the ConcurrentLinkedQueue issue. It requires that the methods pop and push accessing

```
class ReusableLinkedQueue {
   ...
   @Transactional(form = SNAPSHOT)
   public int size() {
    int count = 0;
    for (Node<E> p=first(); p!=null;
           p=p.getNext()) {
       if (p.getItem() != null) {
          if (++count == Integer.MAX_VALUE)
             break;
       }
    }
    return count;
   }
```

Fig. 1. PT fixes the ConcurrentLinkedQueue.size() problem and allows extensibility

mutable shared variables use no explicit synchronizations besides annotations. In this particular example, the size is added as a sequential size method annotated with a form called *snapshot* denoted by @Transactional(form = SNAPSHOT).

The resulting implementation is inherently extensible. The snapshot transaction form guarantees that all shared read accesses of the size method, including the one to p.getNext(), return values present at a common point in time between the invocation and the response of size. To this end, the implementation (detailed in Section 3) associates a version to each value written by any transaction, a snapshot transaction records the highest version upon start and identifies the correct value to return upon reading based on the associated version. In particular, all updates to mutable shared variables are tracked using metadata so that size can detect that a field of ReusableLinkedQueue is being or has been overridden by a concurrent method (e.g., offer or poll) and choose to return a preceding version of the field to bypass the conflict or to abort.

Note that one could have safely omitted the form parameter here (@Transactional) hence adopting the default opaque semantics instead, however, it would limit concurrency by often aborting the size or its potential conflicting updates.

Related issues. Similarly, C# concurrent libraries trade reusability for efficiency. Consider the System.Collections.ConcurrentDictionary CDT as another example. This CDT cannot be easily extended with a correct size() or sum() method, in particular one should not use the existing GetEnumerator() to count or sum-up the elements as the resulting method would not be atomic.

Note that a subset of these problems arise upon inheritance and are thus referred to as *inheritance anomalies* [22].

2.2 Composability

Illustrating the issue. In most languages, there is no clear way of ensuring that atomicity gets preserved under composition of methods into another (the new one invoking the existing ones). This difficulty made it hard to identify bugs in basic Java CDTs, like java.util.Vector. Similar bugs have been unveiled thanks to automated frameworks helping researchers detect atomicity violations [23,24,25,26,27]. As noted earlier [24,26], the version 1.4.2 of the JDK suffered from a critical issue related to one

```
public ReusableOldVector(Collection c) {
  init(c);
}

@Transactional(form = OPAQUE)
public void init(Collection c) {
  elementCount = c.size();
  elementData = new Object[(int)Math.min(
     (elementCount*110L)/100,Integer.MAX_VALUE)];
  c.toArray(elementData);
}
```

Fig. 2. PT fixes the Vector constructor problem and allows composability

of the constructors of java.util.Vector, a widely used abstraction that is supposed to be thread-safe. Upon constructing a new Vector based on an existing Collection c of objects, an ArrayOutOfBoundsException could be raised. The reason is that between the time the size of the collection c is computed and the time c gets converted into an array, a concurrent update may modify the size of the collection c.

The PT solution. The java.util.Vector issue can be easily fixed using our PT methodology that instruments all transactional shared accesses (including to the Collection). The obtained ReusableOldVector simply consists of the original constructor placed into the init method that is annotated with a keyword @Transactional(form = OPAQUE) as depicted in Figure 2. We actually copy-pasted the constructor into a transactional init method simply because the instrumentation is automated for methods but not constructors. Note that we use the opaque form in this example as we motivate later in Section 3.1.

We implemented a ReusableVector CDT by converting all the synchronized methods of the java.util.Vector of the JDK 7 (hence the name ReusableOldVector for the fix of the version 1.4.2) into sequential methods annotated using the opaque transactional wrapper. An advantage of our transaction annotations is that each method, be it private (e.g., ensureCapacityHelper) or public (e.g., ensureCapacity) can be annotated as a transaction. In contrast, nesting of locks may be problematic leading to deadlocks when a programmer encapsulates in a synchronized block a call to an external method already using synchronized.

Related issues. In C#, the aforementioned ConcurrentDictionary CDT exposes GetOrAdd(k,v) and AddOrUpdate(k,v') that are not the (atomic) composition of getting, adding and updating actions. Actually, we observed a lost update problem when GetOrAdd(k,v) and AddOrUpdate(k,v') run concurrently. Intuitively, any concurrent execution of these two methods should always end up in a final state where k is present and its associated value is v': either GetOrAdd fails in adding if AddOrUpdate is linearized first, or v is updated to v' if AddOrUpdate is linearized second. The lost update may lead, however, to an inconsistent final state in which k is present with value v. Precisely because its behavior is incorrect, such subtlety is not visible at the level of the interface of this CDT.

Within the last two years, more than 300 bugs due to this lack of composability were identified in real-world applications [28,27].

Table 2. Domain and states of the algorithm

Domain of the algorithm	
X	the set of references
V	the set of values
$T \subseteq \mathbb{N}$	the set of versions
State of transaction t	
$form \in \{$opaque, hand-over-hand, snapshot, irrevocable$\}$	transaction form (initially opaque)
$wset \subset X \times V$	the write set (initially $\emptyset$)
$rset \subset X \times T$	the read set (initially $\emptyset$)
$bkp \subset X \times V \times T$	backup of value-version (init. $\emptyset$)
$lb \in \mathbb{N}$	versions lower bound (initially 0)
$ub \in \mathbb{N}$	versions higher bound (initially 0)

3 Polymorphic Transactional Memory

We present a polymorphic software transactional memory (PSTM) that underlies our PT methodology. The PSTM implementation has four distinct forms of transactions, opaque, hand-over-hand, snapshot, and irrevocable, hence the name. A bytecode instrumentation phase automatically redirects all shared memory accesses of annotated methods, including the accesses within their nested methods, to the proper transaction form. (Details about nesting semantics are given in Section 5.3.) At run-time the method starts by calling the tx-start passing the optional form as a parameter, invokes tx-read/tx-write instead of directly accessing the shared memory and calls tx-commit right before returning. If the corresponding transaction aborts it restarts and the method returns after the transaction successfully commits.

The domain and transaction states of PSTM are depicted in Table 2, the revocable transactions code is depicted in Algorithm 1. Conflicts are detected at the level of accesses to an object field to enable higher concurrency than object-based detection, thus we say that PSTM is *field-based*. Each field reference is associated with a *versioned lock* that stores the version of the associated reference if unlocked, or its owner if locked ($\ell.owner = \perp$ indicates that the lock is not held). Each transaction consults a global counter, *clock* (Line 2), and maintains version lower and upper bounds, resp. *lb* and *ub*, that help checking whether an access is consistent. Like most time-based software transactional memories (STMs) [29], all transactions update the memory lazily by buffering writes into a write-set, *wset*, until it commits, and have *invisible reads*: none of the read accesses from any transaction is visible from other transactions.

Our solution is deadlock-free because a transaction that cannot acquire a lock simply releases all the previous locks it acquired (and aborts). Adapting more elaborate contention managers [30] to obtain stronger progress guarantees, e.g., to avoid starvation, is left to future work. For the sake of efficiency, only writes lock and reads do not lock, however, the values read must be validated each time a read or a write occurs to make sure that they have not been overridden by concurrent transactions.

We omitted the pseudocode of several helper functions. The function vervalver (Lines 6) is a three-read process spinning until the value and versioned lock returned are

guaranteed to be consistent (as if they were both read atomically). The truncate function (Line 20) discards the oldest entries from the read-set *rset* to keep the two most recent ones. Finally, lock acquires a lock on a given reference and returns the previous lock state or raises an exception if the lock is taken while unlock releases the lock on the given reference, store reports changes in memory, set-ver associates a new version with some value, get-ver/get-val return the versioned lock and the value of the reference, respectively, and *bkp*.version/value returns the old (backup) version/value of the given reference.

3.1 Opaque Transactions

The opaque semantics captures the intuitive single-global-lock semantics provided by common monomorphic (i.e., non-polymorphic) STMs. It has the strongest semantics, hence, it can be used to guarantee atomicity of any method. It clearly benefits the novice programmers who ignore other forms, but in general it limits scalability when applied to long methods. In our package, we used opaque transactions for the short methods with few accesses, like head, first, firstEntry, firstKey and most of the ReusableVector methods because their exploitable concurrency is limited.

Our implementation of opaque transactions follows the LSA algorithm [31]: it acquires locations eagerly, upon write at Line 28. Upon reading a location with a lower version than *ub*, the opaque transaction knows that this value has not been concurrently overridden so it can safely read it and record the corresponding read entry for further validation (Line 11). If the read location has a higher version than *ub*, then the opaque transaction tries to increase its *ub* (Line 15): if the validation is successful then it upgrades *ub* to the value the clock had at the time right before the validate was invoked. This upgrade allows an opaque transaction that observes a value committed after it started to be serialized after the conflicting transaction.

Upon writing, the transaction tries to lock the reference and aborts if it read the *ref* before it got overridden (Line 30). Upon commit, the read set is revalidated (Line 46), the value-version pair is copied (Line 48), the *wset* is reported to memory (Line 49) with a higher version (Lines 44), and locks are released (Line 51).

3.2 Hand-over-Hand Transactions

Hand-over-hand transactions relax the opaque semantics to one that resembles hand-over-hand locking [14]. More precisely, they guarantee elastic-opacity but their implementation differ from $\mathscr{E}$-STM elastic transactions [9] to be made compatible with other transactions (e.g., hand-over-hand transactions record backup versions). Hand-over-hand transactions are well-suited for ensuring atomicity of search structures that are traversed in a specific order. These transactions speed up traversals looking for a single location and possibly updating multiple ones. If used in other circumstances, like for computing the size of a structure, the size method may return a semantically incorrect result (like most concurrent libraries do), hence the need for complementary forms. In our package we used it for wrapping the methods contains, get, insert, insertAll, put, remove, replace, removeAll, putIfAbsent and the like.

Algorithm 1. PSTM algorithm for revocable transaction t

```
1:  tx-start(tx-form)_t:                                  ▷ the form parameter
2:    lb ← ub ← clock                                     ▷ versions lower-/upper-bound
3:    if tx-form ≠ ⊥ then form ← tx-form                  ▷ initialize tx form
4:    else form ← opaque                                  ▷ opaque by default

5:  tx-read(ref)_t:                                       ▷ transactional read
6:    ⟨ℓ,v⟩ ← vervalver(ref)                             ▷ get lock and value copies atomically
7:    if ℓ.owner ∉ {t,⊥} then abort()                     ▷ locked by other, conflict
8:    if ℓ.owner = t then                                 ▷ if locked by me
9:      v ← w.val : w ∈ wset ∧ w.ref = ref                ▷ return my written value
10:   if ℓ.owner = ⊥ ∧ ℓ.version ≤ ub then                ▷ if no conflict
11:     rset ← rset ∪ {⟨ref,ℓ.version⟩}                    ▷ record read entry
12:   if ℓ.owner = ⊥ ∧ ℓ.version > ub then                ▷ ref's been written, conflict
13:     if form = opaque then                             ▷ if opaque tx
14:       now ← clock                                     ▷ record clock locally
15:       if validate() then ub ← now else abort()        ▷ upgrade upper bound
16:       rset ← rset ∪ {⟨ref,ℓ.version⟩}                  ▷ record read entry
17:     else if form = hand-over-hand then                ▷ if hand-over-hand tx
18:       if ¬validate() then abort()                     ▷ validate (potentially truncated) read-set
19:       rset ← rset ∪ {⟨ref,ℓ.version⟩}                  ▷ record read entry
20:       if wset = ∅ then truncate(rset,2)               ▷ keep only last two entries
21:     else if form = snapshot then                      ▷ if snapshot tx
22:       if (old = bkp.version(ref)) ≤ ub then           ▷ sufficiently old version
23:         v ← bkp.value(ref)                            ▷ return old version
24:         rset ← rset ∪ {⟨ref,old⟩}                     ▷ record read entry
25:       else abort()                                    ▷ old version is too recent
26:   return v

27: tx-write(ref,value)_t:                                ▷ transactional write
28:   try ℓ = lock(ref) catch-e abort()                   ▷ acquire the lock and copy old lock state
29:   if ℓ.owner = ⊥ ∧ ℓ.version > ub then                ▷ ref's been written, WAW conflict
30:     if ref ∈ rset then abort()                        ▷ cycle in precedence graph
31:     if ¬validate() then abort()                       ▷ validation of some of the conflicts
32:   wset ← wset ∪ {⟨ref,value⟩}                         ▷ buffer write entry
33:   return ok

34: validate()_t:                                         ▷ make sure read set has not changed
35:   for all ⟨r,ver⟩ ∈ rset do                           ▷ for any read entry...
36:     ℓ ← get-ver(r)                                    ▷ reread its versioned lock
37:     if ver ≠ ℓ.version ∨ ℓ.owner ∉ {t,⊥} then         ▷ if has been overriden/locked
38:       return false                                    ▷ validation fails (simplified)
39:   return true

40: abort()_t:                                            ▷ rollback before automatic restart
41:   for all w ∈ wset do unlock(w.ref)                   ▷ release all locks

42: tx-commit()_t:                                        ▷ try to commit
43:   if wset ≠ ∅ then                                    ▷ if something to redo
44:     ts ← clock++                                      ▷ fetch-and-increment global counter
45:     if ts > lb+1 then                                 ▷ if concurrent update...
46:       if ¬validate() then abort()                     ▷ validate read set, check WAR conflicts
47:     for all w ∈ wset do                               ▷ apply writes and release locks
48:       bkp ← bkp ∪ {⟨w.ref,get-val(w.ref),get-ver(w.ref)⟩}   ▷ backup
49:       store(w.val,w.ref)                              ▷ write in memory
50:       set-ver(w.ref,ts)                               ▷ upgrade version
51:       unlock(w.ref)                                   ▷ release lock
```

A hand-over-hand transaction automatically ignores the old values read during its read-only prefix (i.e., as long as $wset = \emptyset$). When a hand-over-hand transaction still in its read-only prefix reads a location, it creates a new read entry in its *rset* and discards

all but the two last entries by truncation (Line 20). By contrast, the $\mathscr{E}$-STM elastic transactions [9] used to keep only one extra entry to ensure the atomicity of the list-based set. This was made possible thanks to a marking trick used before re-allocating the memory in unmanaged language. As we do not control memory reclamation in Java, we could not use the same trick, which explains why a hand-over-hand transaction needs to maintain up to two read entries to guarantee correctness of pointer-based structures. Although keeping two entries is actually sufficient for multiple search structures (e.g., linked lists, skip lists, hash tables), more entries could be thought for other application semantics. If the read location has a higher version than *ub*, the entries of its (potentially truncated) read set get revalidated (Line 18) to make sure its read values are still up-to-date. By exploiting the semantics of search structures, hand-over-hand transactions enable higher concurrency than traditional transactions. In particular, a hand-over-hand transaction that has traversed an ordered structure and that is updating its end would not conflict with a concurrent transaction updating the beginning of the structure.

When a hand-over-hand transaction writes for the first time, it has to revalidate the two entries of its *rset*. When a hand-over-hand transaction has already written (i.e., it is no longer executing its read-only prefix) it behaves like an opaque transaction: it stops truncating the *rset*. A hand-over-hand transaction commits as an opaque transaction except that its validation may occur on a truncated read set (Line 46).

3.3 Snapshot Transactions

In contrast with opaque transactions, snapshot transactions are read-only and tolerate concurrent updates by potentially returning values that can be slightly out-of-date at the time it commits. Note that atomicity is ensured because all its read values are guaranteed to be up-to-date at a common point of the execution between the invocation and the response of the transaction. In our package, snapshot transactions are used for methods iterating over a collection of elements: descendingSet, headMap, headSet, size, subMap, subSet, tailMap, toArray, toString and the like.

To exploit concurrency between updates and snapshots the implementation of a snapshot transaction builds upon multi-version concurrency control. Multi-version concurrency control has proved useful in software transactional memories, like JVSTM [32], to guarantee either opacity or snapshot isolation but not to combine both. Maintaining the minimum of versions per object that maximizes the variety of output histories comes at a cost [33]: the proposed useless-prefix multi-version (UP MV) STM guarantees this property but, as a drawback, does not support invisible reads. To avoid such constraints, we chose to maintain two versions at each location. All update transactions create a backup value-version pair before overriding them (Line 48). The snapshot transaction has simply to detect that the location it aims to access has a higher version than its upper bound *ub* (Line 12) to try getting an older version (Line 22). The transaction has to abort if the old version is too recent at Line 25 as there are no older versions.

3.4 Irrevocable Transactions

We provide irrevocable transactions that never abort. They are used to execute atomically a series of statements in a pessimistic manner without speculation, similar to

critical sections, and are particularly useful for executing external actions like I/O. One can delimit an irrevocable transaction using a dedicated Irrevocable annotation. We omitted the pseudocode of irrevocable transactions, as they simply consist of (implicit) mutual exclusion [34]. All regions annotated as irrevocable are identified by the underlying Java agent, Deuce [18], and automatically wrapped in a critical section. In contrast with other forms, an irrevocable transaction starts by trying to acquire a reader-writer lock in exclusive mode that is held until the commit of the irrevocable transaction is called. This strategy prevents an irrevocable transaction from running concurrently with any other transaction but lets revocable transactions run concurrently. A revocable transaction actually acquires a shared reader-writer lock to guarantee this. Hence, any transaction trying to execute while an irrevocable transaction is running is blocked until the irrevocable transaction commits.

4 Correctness

In this section, we discuss the correctness of PSTM. First, it is crucial that all transaction forms be pairwise *compatible*, meaning that the semantics of each transaction form be preserved despite concurrency. In particular, the semantics of some writing transaction should not impact the semantics of another transaction accessing the written elements. Second, a concurrent library should always be linearizable and *reusable*.

4.1 Invariants

The model is a concurrent environment where a set of threads execute transactional methods on shared data types. The synchronization semantics of each method is given by its transaction form that can be of the type *opaque*, *hand-over-hand* or *snapshot* but we ignore the *irrevocable* form as it cannot run concurrently with others. A *transaction* is the execution sequence of a method read and write accesses to the shared memory. It *completes* either by committing, meaning that the corresponding method returns and all its changes are visible from other transactions, or by aborting, meaning that no changes are visible. Note that the system implicitly starts a new transaction executing the same method if the preceding one aborted. A *well-formed* execution of this model is an execution where each transaction executed by one thread completes before the same thread starts another: the nesting discussion is deferred to Section 5.3.

For the sake of compatibility our three revocable forms defer conflict resolution to the same conflicting transaction and never ignore WAW conflicts.

Invariant 1. *Let t_1 and t_2 be two transactions involved in a WAW conflict. At least one of these two aborts.*

For the sake of high concurrency, we adjust the conflict resolution strategy depending on the transaction form. We differentiate the semantics of our forms in the way they handle RAW and WAR conflicts. Reads are *idempotent*, as they do not affect the system state; hence the decision taken by the reading transaction detecting a RAW conflict, which depends on the semantics of this transaction, never affects the semantics of other transactions. Specifically, reads can interchangeably return committed values or abort, this result is invisible from the standpoint of concurrent transactions.

Invariant 2. *Let two transactions t_r and t_w be involved in a RAW conflict where t_r executes the conflicting read whereas t_w executes the conflicting write. Transaction t_r either ignores the conflict by resuming or resolves it by aborting itself.*

The problem of enhancing concurrency is more subtle upon WAR conflicts. If a transaction tries to solve a WAR conflict upon detecting it by writing, then it could either conservatively limit concurrency (e.g., by aborting while its semantics is hand-over-hand) or it could violate the semantics of other transactions (e.g., committing while the conflicting transaction is opaque and this conflict would induce a cycle in the precedence graph observed by this transaction). This issue is addressed by forcing the reading transaction to solve all WAR conflicts, which requires all reading transactions to (re-)validate either at some later read, write or commit.

Invariant 3. *Let two transactions t_r and t_w be involved in a WAR conflict where t_r executes the conflicting read whereas t_w executes the conflicting write. Transaction t_r either ignores the conflict by resuming or resolves it by aborting itself.*

4.2 Semantics Preservation

Opacity requires committed transactions to be strictly serializable and non-committed ones to observe consistent states [16]. The semantics of opaque transactions is preserved due to Invariant 1 and the fact that all transactions write values at commit time so that the read operations cannot return transient values (Line 15). As opposed to other forms of optimistic transactions [35,36], a snapshot transaction is not necessarily serialized at its commit time as it only returns values that were present at its start time (Lines 10 and 22) to exploit multi-versioning while ensuring strict serializability. Finally, hand-over-hand transactions prevent some read/write from being interleaved with conflicting writes to ensure elastic opacity [9]. As they are not necessarily strictly serializable, they allow to implement efficient linearizable CDTs.

4.3 Linearizability of the Data Type

One can easily deduce a linearization point for each operation of a transaction form, at which the transaction of the corresponding form appears to execute instantaneously. The opaque transaction always keep the locks until commit hence a valid linearization point is the point at which it starts releasing its first lock (Line 51); a read-only opaque transaction linearization point is at Line 6 of its last read. The snapshot transaction may return values that have been overridden, hence its linearization point cannot be taken from its commit phase, however, since it makes sure that all versions it observes falls in its range upper bound, ub, a valid linearization point is the point where it sets its timestamp to the global clock (Line 2). The hand-over-hand transaction is well-suited for some data types but not all, and this is the responsibility of the expert to use it appropriately. For example, one cannot implement a data type exporting a putIfAbsent(x, y) method synchronized with hand-over-hand transaction. The hand-over-hand transaction may ignore conflicting writes, hence acting as if it was linearized after them: a valid linearization point for an appropriate data type is when it grabs the lock of its first

write (Line 28) or at Line 6 of its last read (if read-only). Recall that linearizability is ensured precisely because it is defined for arbitrary objects (or types) without requiring that all low-level reads and writes of a method appear as if they were all executed instantaneously [2].

4.4 Reusability

Extensibility is ensured by the fact that our transaction forms are compatible as discussed previously, hence adding a new method annotated with one of the proposed forms guarantees that the semantics of existing methods will not be affected.

Composability is guaranteed by the fact that whatever forms protect original methods, the programmer always has the possibility to derive a composite annotated method that will execute atomically. By default the semantics of the composite method would be opaque which guarantees the atomicity of any method. In particular, while two traversals may be originally annotated as hand-over-hand ignoring some conflicts for the sake of concurrency, a new composite method annotated as opaque that reuses them switches their semantics to opaque. The simplicity stems from the fact that the source code of the original methods does not need to be available as the switch is transparently done at the bytecode level. The nesting of different forms is discussed in Section 5. Note that in addition to concurrent methods annotated with transactions, bare sequential methods (without annotation) can be composed into a composite concurrent method that is annotated. This allows programmers to reuse existing sequential ADTs (in addition to transactional CDTs) to produce CDTs that are themselves reusable.

5 Language Integration

We integrated the PT methodology to Java to simplify the development and reuse of concurrent objects using annotations. We detail below how the bytecode gets automatically instrumented, how exceptions are handled, how transactions nest within each other and to which extend one can use legacy code.

5.1 Bytecode Instrumentation

Our implementation of the PT methodology extends the Deuce [18] bytecode instrumentation framework to support multiple forms of transactions. Figure 3 depicts the process of the PT methodology: (1) The programmer first compiles the data types whose methods accessing mutable shared variables are annotated with transaction forms—these annotations persist in the bytecode. Then (2) the Java agent automatically produces a transactional version of all objects used to redirect all their shared accesses invoked within a transaction to the tx-read/tx-write of the corresponding transaction form of PSTM. (3) This outputs the bytecode of the corresponding reusable CDT that can be run by any JVM.

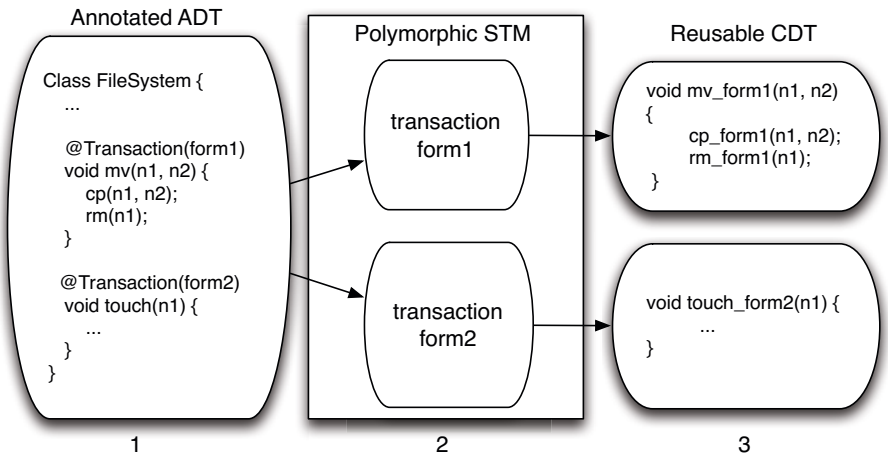


Fig. 3. Our PT methodology relies on annotating manually a sequential (or transactional) type, and producing a reusable CDT by automatically instrumenting methods using the transactional wrappers of the underlying polymorphic transactional memory system (e.g., PSTM)

5.2 Exception Handling

Our framework supports exception handling within transactions. An exception raised within a transaction provokes the transaction to commit and the exception gets propagated outside the scope of the transaction similarly to synchronized blocks and as implemented in Deuce. The advantage of this semantics is to guarantee that the cause of the exception remains visible if the exception itself is visible. An alternative interesting semantics is *failure atomicity* where an exception is considered a failure from which the system recovers by rolling back to the most recent checkpoint. For a failure-atomic exception handler in Java using STMs we refer the reader to the CXH compiler [37].

5.3 Nesting Semantics

For the sake of safety, we adopt a conservative flat nesting approach by imposing the most restrictive (when comparable) form of the inner/outer transaction to always prevail. In our form examples, opaque prevails over snapshot and hand-over-hand. To motivate our choice take the following non-trivial example where Alice would like to reuse Bob's package. For efficiency purpose Bob's package provides a hand-over-hand contains(y) and a hand-over-hand put(x) methods. Alice would like to derive a new data type by nesting these two methods into an opaque putIfAbsent(x,y) that inserts x in a data structure only if y is absent. It is crucial that the contains(y) and put(x) inherit the opaque semantics of its parent putIfAbsent(x,y) transaction to avoid a write-skew problem if a putIfAbsent(y,x) happens to run concurrently. If the opaque semantics is not inherited, then there exists an execution in which both contains(x) and contains(y) executing concurrently return false and then both x and y get successfully inserted, leading to an inconsistent state where both x and y are present. Note that Alice has to be

an expert who understands the semantics of a transaction to use it. This is particularly important for her to be aware that putIfAbsent cannot be executed as a hand-over-hand transactions in her new data type.

5.4 Legacy Code

The PT methodology recommend not to use other forms of synchronization besides transaction forms, however, legacy code can be invoked through irrevocable transactions. In particular, the PT methodology does not guarantee compatibility between the transaction forms and the explicit use of compare-and-swap and mutual exclusion as it there is no clear semantics on conflicting accesses using these different synchronization techniques. A potential risk is that non-transactional accesses would typically observe transient states if they could access transactional CDTs as we do not provide strong atomicity [38]. Note that requiring CDTs to be accessed transactionally can be enforced in Java through the use of pre-existing setters and getters as, for example, when accessing ThreadLocal variables. Finally, the PT methodology can still be used to turn most sequential ADTs into equivalent atomic CDT.

6 Evaluation

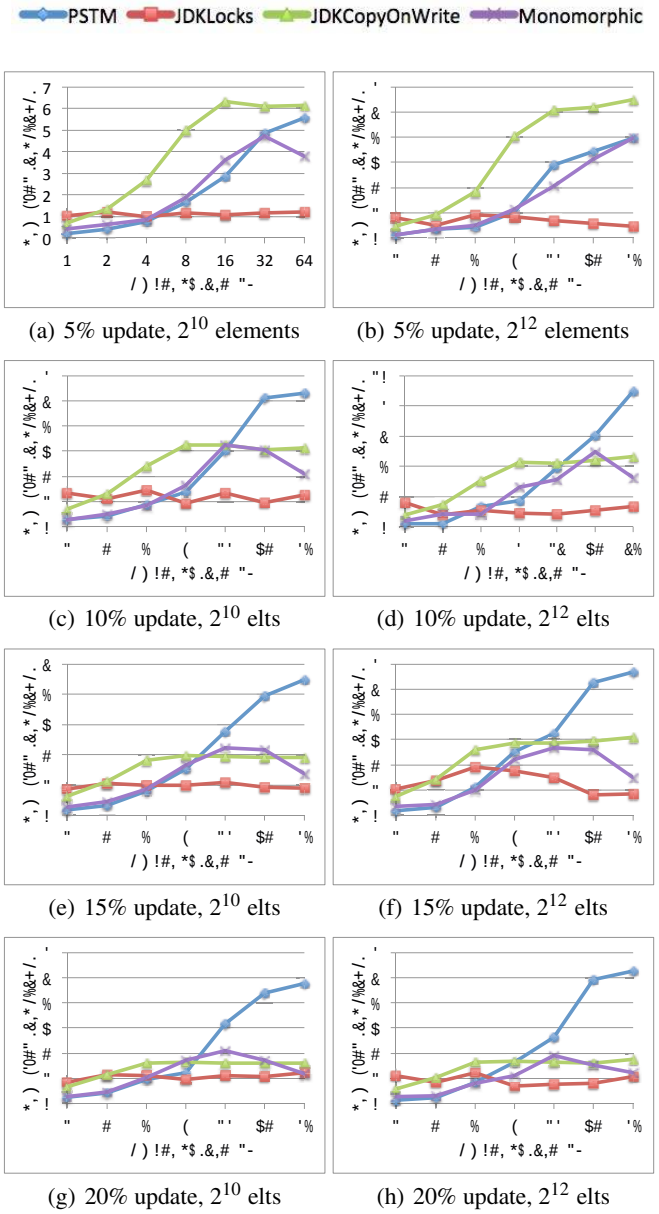
In this section, we evaluate our methodology in Java. We compare our reusable library to lock-based and lock-free libraries from the JDK and STM-based libraries, on SPARC and x86-64 architectures using Synchrobench and the Vacation application.

6.1 Settings

We used two 64-way machines with different architectures: an UltraSPARC T2 (Niagara 2) 1.165GHz with 32GB of memory and a 2U server with 4 AMD Opteron 6378 2.4GHz 16-core processors with 128GB of memory. (All graphs except the last ones report the results from SPARC.) Each data point of the graphs corresponds to the throughput averaged over 3 runs of 13 seconds executed in separate JVM instances and where the 10 first seconds of each run are used to warmup each JVM. (Each point of the graph thus takes nearly 40 seconds to be computed and we carefully checked that the variance was negligible enough for the results to be meaningful.) The JVM runs in server mode with 2G of initial/maximum Java heap size.

6.2 PT Methodology vs JDK

First, we evaluate two techniques from the JDK 6 to construct reusable set CDTs: (1) the copy-on-write wraps a set ADT into a java.util.concurrent.copyOnWriteArraySet to obtain an array whose methods are guaranteed to be atomic and whose read-only methods are wait-free (JDKCopyOnWrite), and (2) a lock-based one consisting of wrapping a set ADT into a synchronizedSet (JDKLocks) to transparently make its methods atomic. Second, we evaluate the PT methodology when based on PSTM and when using four implementations of state-of-the-art (monomorphic) STMs:



(a) 5% update, 2^{10} elements (b) 5% update, 2^{12} elements

(c) 10% update, 2^{10} elts (d) 10% update, 2^{12} elts

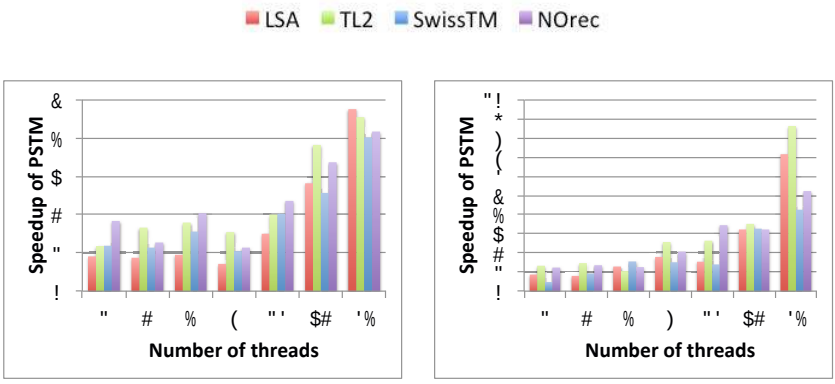
(e) 15% update, 2^{10} elts (f) 15% update, 2^{12} elts

(g) 20% update, 2^{10} elts (h) 20% update, 2^{12} elts

Fig. 4. Throughput (normalized over sequential) obtained when using polymorphic transactions (PSTM), the lock-based synchronizedSet from the JDK, the copyOnWriteArraySet from the JDK and the highest throughput we obtained from our four monomorphic STMs (LSA, TL2, SwissTM, NOrec). Workloads include 10% of size, from 5% to 20% of updates (add or remove with the same probability) and from 70% to 85% of contains.

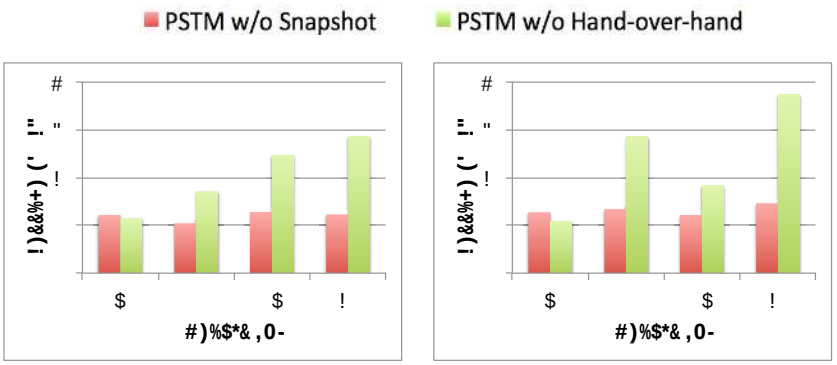
LSA [31], TL2 [39], SwissTM [40] and NOrec [41]. For evaluating them on the same ground, all these implementations are field-based and match the interface of Deuce [18] (in particular, LSA, TL2, and NOrec are the standard versions provided with Deuce). We tested all STMs including PSTM and observed that PSTM was more efficient than other STMs on ReusableLinkedQueue, ReusableLinkedListSortedSet, ReusableHashMap and the ReusableSkipListSet thus we only report the data from the ReusableLinkedListSortedSet. This benchmark comprises add/remove (5–20%), contains (70–85%) and size (10%) methods on a sorted linked list data structure, methods that are all provided by Java CDTs.

Figure 4 depicts the throughput of our PT methology (PSTM), of existing monomorphic STMs, and of existing copy-on-write and pessimistic lock-based solutions, all normalized over the throughput of bare sequential code, on SPARC. About the monomorphic STMs curve, we have chosen, for each single point, the maximum throughput we obtained from LSA, TL2, SwissTM, and NOrec. The detailed speedup of PSTM over each of these STMs is presented in Section 6.5. The overall performance of PSTM is better than the synchronization alternatives. At low levels of contention, when update ratio is 5% or at low number of threads, PSTM executes slower than a copy-on-write and pessimistic lock-based alternatives. The reason is that PSTM suffers from the overhead (due to wrapping each individual access) that is common to STM implementations including monomorphic ones. This overhead is however rapidly compensated as PSTM scales well with contention whereas the copy-on-write solution scales badly and the lock-based solution does not even scale. More precisely, PSTM speeds up the existing copy-on-write solution by 2.4× on average, and the existing pessimistic lock-based solution by 4.7× on average at the highest level of parallelism we have at our disposal (64 hardware threads).



(a) 2^{10} elements (b) 2^{12} elements

Fig. 5. Speedup of PSTM over each monomorphic STM: LSA, TL2, SwissTM and NOrec, from 1 to 64 threads (the throughput is identical when speedup has value 1)



(a) 2^{10} elements (b) 2^{12} elements

Fig. 6. Speedup of PSTM over the variant that does not use snapshot transactions and the one that does not use hand-over-hand transactions

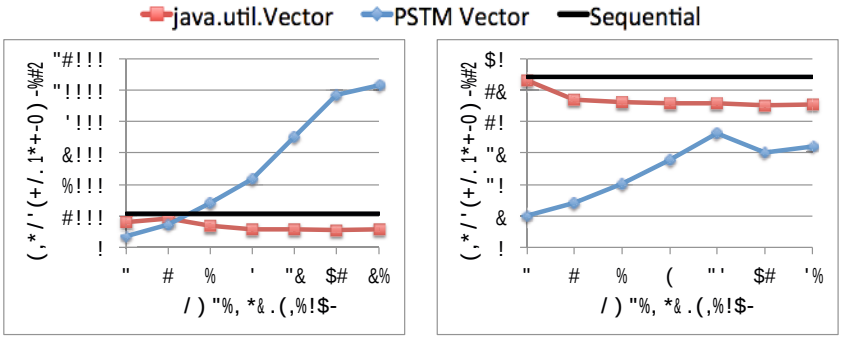
6.3 Polymorphism vs Monomorphism

Figure 5 depicts the speedup of PSTM over monomorphic STMs, LSA, TL2, SwissTM, NOrec, as the throughput of PSTM divided by the throughput of the corresponding monomorphic STM (with 20% update) on SPARC.

These results show that PSTM scales better than other STMs. More precisely, PSTM presents a slight overhead at low levels of parallelism, typically when running a single thread but rapidly compensates this slight overhead in concurrent executions. This overhead is caused by the fact that polymorphism adds some necessary checks at each access to determine the type of the current transaction and that it records one version at each write for multi-version concurrency control. At large levels of parallelism, PSTM is significantly more efficient as its polymorphism exploits adequately concurrency whereas monomorphic STM executes a single form of transaction, which has a fortiori the strongest semantics that also limits concurrency. More precisely, PSTM outperforms the tested monomorphic STMs by up to a factor of 8.6× on 64 threads. This improvement is specific to polymorphism as PSTM outperforms every single monomorphic STM by a mean factor of at least 4 on 64 threads.

6.4 Adding Forms Is Beneficial

We have also evaluated the advantage of combining three revocable transaction semantics instead of only two. Figure 6 illustrates the speedup of using the three revocable semantics (PSTM) over the use of only two of them at high level of concurrency (64 threads) for different update ratios on SPARC. "PSTM without Snapshot" indicates the speedup of PSTM over a variant where all snapshot transactions have been replaced by opaque transactions. (All transactions of this variant are either opaque or hand-over-hand.) "PSTM without Hand-over-hand" indicates the speedup of PSTM over another variant where all hand-over-hand transactions have been replaced by opaque transactions. (All transactions of this variant are either opaque or snapshot.)



(a) Read-only workload (b) Contended workload

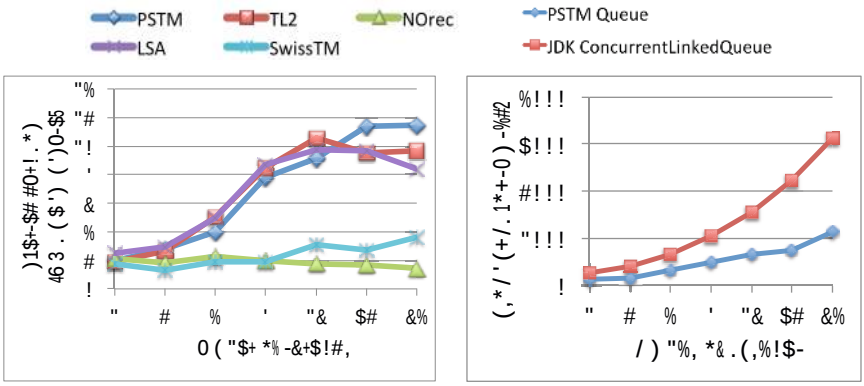
Fig. 7. Comparison of our ReusableVector (PSTM Vector) against the java.util.Vector from the JDK and the bare sequential Vector

The overall result is that exploiting the three revocable forms of PSTM is always beneficial as the speedup is never below 1. In particular the speedup of PSTM over "PSTM without hand-over-hand" speedup tends to grow with the update ratio. This result is not surprising as we expected the combination of the three revocable semantics to be especially suited to limit the number of aborts, thus, it is natural for its gain to increase with the contention. An interesting observation is that the speedup of PSTM over "PSTM without snapshot" is generally low. This is explained in part by the implementation of the latter being particularly lightweight: "PSTM without snapshot" has less overhead because it does not backup values upon write as multiple versions are not needed. By contrast, both PSTM and "PSTM without hand-over-hand" have snapshot transactions and require one backup per write.

6.5 java.util.Vector vs ReusableVector

The vector benchmark comprises add, remove and contains and compare the performance obtained with our ReusableVector against the lock-based java.util.Vector from the JDK and against a bare sequential code version that is taken from the java.util.Vector from which we removed all locks.

Figure 7 depicts the throughput for a read-only workload and a contented workload (with 10% updates) on the java.util.Vector from the JDK 7 and on our ReusableVector (on SPARC). Interestingly, the PT methodology does a better job in outperforming sequential code when concurrency can be exploited. The reason is that our approach differentiates automatically read and write accesses to object fields and enables read sharing. By contrast, the java.util.Vector relies essentially on synchronized methods that act as mutual exclusion independently from their access mode. Consequently, our solution performs better than the java.util.Vector on the read-only workload by up to 4× (Figure 7(a)). However, we can observe the high overhead due to the bookkeeping of TM wrappers at low levels of parallelism. When almost no concurrency can be exploited (Figure 7(b)), our approach executes significantly slower.



(a) STAMP Vacation (b) Reusable vs non-reusable queue

Fig. 8. STAMP Vacation results and comparison of our ReusableLinkedQueue (PSTM Queue) against the j.u.c.ConcurrentLinkedQueue from the JDK

6.6 The Vacation Application

We evaluate our methodology with a Java version of STAMP vacation [13]. This application is a typical transactional application in that it uses a travel reservation database engine that organizes cars, rooms, flights and customers tables into four red-black trees. Tables are accessed through three transactions to (a) check prices and reserve few items, (b) delete customers and (c) add or remove items of a reservation. To evaluate the PT methodology, we made the first transaction read-only by simply returning prices and annotated it as snapshot, we annotated the two others as opaque (all transactions are opaque when running the monomorphic STMs). We set the initial and maximum Java heap size to 4G and use the recommended low contention parameters of vacation.

Figure 8(a) depicts the vacation performance as the inverted duration time on x86-64. The performance of PSTM keeps scaling up to 64 threads at which point it becomes 19% faster than monomorphic alternatives. Although monomorphic STMs stop scaling at 32 threads, they are more efficient than PSTM at lower levels of parallelism confirming our observations on micro-benchmarks. In particular, TL2 achieves good performance at 16 threads, which may be due to TL2's code being optimized through the use of metadata pools to reduce memory reclamation. This feature seems appealing in more realistic benchmarks, like vacation, that tend to use more memory for longer than our micro-benchmarks.

6.7 j.u.c.ConcurrentLinkedQueue vs ReusableQueue

We also evaluated the cost of reusability by comparing the performance of one of our reusable CDT against a similar but non-reusable lock-free CDT on the x86-64 architecture (our SPARC results were similar). We compare the queue CDT of the JDK 7 as described in Section 2.1 to the ReusableLinkedQueue as they both rely on a linked list implementation where elements are added to the head and the remove operation

searches for the given value by traversing the list. Figure 8 shows the performance of our ReusableLinkedQueue against the ConcurrentLinkedQueue running 30% of size, 1% of updates (add/remove), 69% of contains on a 128-element queue. The performance difference is quite substantial as the non-reusable queue speeds up the reusable queue by up to 3×. We see two reasons: (a) some overhead is induced by the extra bookkeeping of our synchronizations that triggers the Java garbage collector more often, (b) the atomicity of the reusable size and updates precludes a lot of non-atomic executions allowed by the non-reusable skip list. Even though we may reduce the overhead using hardware transactional memory opcodes, making sure that someone can reuse a concurrent library comes with a substantial cost. As opposed to transactional memories that tend to scale badly [12], PSTM performance scales.

7 Related Work

There is a large body of work on concurrent object-oriented programming languages. Some approaches rely on monitors, like Guava [42], that may restrict inter-method concurrency. SCOOP allows to specify an object accessed by a different process as *separate* [43]. A client object must acquire an exclusive lock on a separate object before invoking it through a routine. SCOOP was ported to Java [44] but is not inherently deadlock-free [45]. Some recent lock-inference techniques are deadlock-free, yet they require the programmer to provide a semantic description of methods [46].

One of the original motivations for transactional memory (TM) is to alleviate lock-related problems like deadlocks [3]. Without deadlocks the program is guaranteed to execute, and a simple exponential backoff strategy can manage contention so that the program progresses. The first TM to handle concurrency in a dynamic control flow redirects speculative accesses to Java object copies [47]. The back-end interface of this TM implementation was later improved in a Java library supporting interchangeable transactional factory [48]. Lightweight transactions were suggested to avoid copying entire objects by using a mapping of addresses to word-sized ownership records [49] before field-based instrumentation was proposed [18].

Exploiting highly concurrent transactions was extensively explored [7,50,51,52,8,35,53,54]. The aim of Galois [52], JANUS [53] and CSpec [54] was not to simplify concurrent programming but to enhance optimistic concurrency in complex scenarios; Galois requires explicit commutativity specifications, Janus exploits an offline learning phase of commutative relations and CSpec converts already concurrent code with annotations and locks.

Note that the PT methodology could potentially achieve similar concurrency results as open nesting [7] and transactional boosting [8] as they all exploit the application-level semantics. In contrast with our solution, both techniques acquire abstract locks eagerly and need explicit abort handlers to compensate their actions upon roll-back. Existing implementations of open nesting require to order transactions and to guarantee that transactions are nested in this specific order to prevent an abort handler from deadlocking [55]. Transactional boosting suggests to add timeouts to avoid deadlocks when two transactions acquire abstract locks in different order [8]. Note that our current implementation of the PT methodology needs annotated sequential code but does not

use compensating actions. As it keeps all the locks it acquires until commit or abort time, it is inherently deadlock-free.

8 Concluding Remarks

Concurrent programming would greatly be simplified if concurrent libraries were made reusable: a programmer could build upon any CDT without having to understand its synchronization internals. The PT methodology helps reaching this goal by allowing collaborative development of scalable libraries any programmer can compose and extend, hence confirming our recent observation [11]. This new methodology promotes a clear separation of the implementation of synchronization semantics, which requires advanced programming skills, from the raw sequential code that describes the expected behavior of an abstract data type.

The ease of use of this methodology is demonstrated using automatic instrumentation of method bytecode. We confirmed using a novel Java library that reusability of CDTs comes at a cost. However, we also observe that this cost can be rapidly compensated by exploiting the high level of concurrency of existing multicore architectures. Actually, one does not even have to sacrifice scalability for reusability.

Future work includes (a) formalizing a framework to derive incompatibilities of synchronization semantics and (b) optimizing our current implementation through concurrent irrevocable transactions [56] or transactional instruction extensions with Java opcodes to reduce overhead.

Acknowledgments. The Java port of SwissTM is from Mihai Letia. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Meyer, B.: Reusability: The case for object-oriented design. IEEE Software 4(2), 50–64 (1987)
2. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
3. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
4. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
5. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP, pp. 48–60 (2005)
6. Wakita, K., Yonezawa, A.: Linguistic supports for development of distributed organizational information systems in object-oriented concurrent computation frameworks. SIGOIS Bull. 12, 185–198 (1991)
7. Moss, J.E.B.: Open nested transactions: Semantics and support. In: Workshop on Memory Performance Issues (February 2006)
8. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: PPoPP, pp. 207–216 (2008)

9. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 93–107. Springer, Heidelberg (2009)
10. Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., Pingali, K.: Exploiting the commutativity lattice. In: PLDI, pp. 542–555 (2011)
11. Gramoli, V., Guerraoui, R.: Democratizing transactional programming. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 1–19. Springer, Heidelberg (2011)
12. Turon, A.: Reagents: expressing and composing fine-grained concurrency. In: PLDI, pp. 157–168 (2012)
13. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: IISWC (2008)
14. Bayer, R., Schkolnick, M.: Concurrency of operations on b-trees. In: Readings in Database Systems, pp. 129–139 (1988)
15. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
16. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP, pp. 175–184 (2008)
17. Welc, A., Saha, B., Adl-Tabatabai, A.R.: Irrevocable transactions and their applications. In: SPAA, pp. 285–296 (2008)
18. Korland, G., Shavit, N., Felber, P.: Deuce: Noninvasive software transactional memory. Transactions on HiPEAC 5(2) (2010)
19. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC (1996)
21. Burnim, J., Necula, G., Sen, K.: Specifying and checking semantic atomicity for multithreaded programs. In: ASPLOS, pp. 79–90 (2011)
22. Matsuoka, S., Yonezawa, A.: Analysis of inheritance anomaly in object-oriented concurrent programming languages. In: Research Directions in Concurrent Object-Oriented Programming, pp. 107–150. MIT Press, Cambridge (1993)
23. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349 (2003)
24. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Softw. Eng. 32(2), 93–110 (2006)
25. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
26. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: Static checking and inference for Java. ACM Trans. Program. Lang. Syst. 30 (2008)
27. Lin, Y., Dig, D.: Check-then-act misuse of java concurrent collections. In: ICST, pp. 164–173 (2013)
28. Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M., Yahav, E.: Testing atomicity of composed concurrent operations. In: OOPSLA, pp. 51–64 (2011)
29. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-based software transactional memory. IEEE Trans. Parallel and Distributed Systems 21(12), 1793–1807 (2010)
30. Scherer, I.W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
31. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
32. Cachopo, J.A., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
33. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: PODC, pp. 16–25 (2010)

34. Carlstrom, B.D., Chung, J., Chafi, H., McDonald, A., Cao Minh, C., Hammond, L., Kozyrakis, C., Olukotun, K.: Transactional execution of Java programs. In: SCOOL (2005)
35. Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. In: POPL, pp. 19–30 (2010)
36. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. In: PPoPP, pp. 387–388 (2014)
37. Harmanci, D., Gramoli, V., Felber, P.: Atomic boxes: Coordinated exception handling with transactional memory. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 634–657. Springer, Heidelberg (2011)
38. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. 5 (2006)
39. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
40. Dragojević, A., Guerraoui, R., Kapałka, M.: Stretching transactional memory. In: PLDI, pp. 155–165 (2009)
41. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: PPoPP, pp. 67–78 (2010)
42. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: OOPSLA, pp. 382–400 (2000)
43. Meyer, B.: Systematic concurrent object-oriented programming. Commun. ACM 36(9), 56–80 (1993)
44. Torshizi, F.A., Ostroff, J.S., Paige, R.F., Chechik, M.: The SCOOP concurrency model in Java-like languages. In: CPA, pp. 7–24 (2009)
45. West, S., Nanz, S., Meyer, B.: A modular scheme for deadlock prevention in an object-oriented programming model. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 597–612. Springer, Heidelberg (2010)
46. Gueta, G.G., Ramalingam, G., Sagiv, M., Yahav, E.: Concurrent libraries with foresight. In: PLDI, pp. 263–274 (2013)
47. Herlihy, M., Luchangco, V., Moir, M., Scherer, I.W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
48. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: OOPSLA, pp. 253–262 (2006)
49. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA, pp. 388–402 (2003)
50. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., Olukotun, K.: The atomos transactional programming language. In: PLDI, pp. 1–13 (2006)
51. Carlstrom, B.D., McDonald, A., Carbin, M., Kozyrakis, C., Olukotun, K.: Transactional collection classes. In: PPoPP, pp. 56–67 (2007)
52. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI, pp. 211–222 (2007)
53. Tripp, O., Manevich, R., Field, J., Sagiv, M.: JANUS: exploiting parallelism via hindsight. In: PLDI, pp. 145–156 (2012)
54. Xiang, L., Scott, M.L.: Compiler aided manual speculation for high performance concurrent data structures. In: PPoPP, pp. 47–56 (2013)
55. Ni, Y., Menon, V., Abd-Tabatabai, A.R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: PPoPP, pp. 68–78 (2007)
56. Spear, M.F., Silverman, M., Dalessandro, L., Michael, M.M., Scott, M.L.: Implementing and exploiting inevitability in software transactional memory. In: ICPP, pp. 59–66 (2008)