Transactions on Mergeable Objects

Deepthi Devaki Akkoorath, and Annette Bieniusa

University of Kaiserslautern, Kaiserslautern, Germany
{akkoorath,bieniusa}@cs.uni-kl.de

Abstract. Destructible updates on shared objects require careful handling of concurrent accesses in multi-threaded programs. Paradigms such as Transactional Memory support the programmer in correctly synchronizing access to mutable shared data by serializing the transactional reads and writes. But under high contention, serializable transactions incur frequent aborts and limit parallelism. This can lead to a severe performance degradation.
In this paper, we propose *mergeable transactions* which provide a consistency semantics that allows for more scalability even under contention. Instead of aborting and re-executing, object versions from conflicting updates on shared objects are merged using data-type specific semantics. The evaluation of our prototype implementation in Haskell shows that mergeable transactions outperform serializable transactions even under low contention while providing a structured and type-safe interface.

Keywords: Concurrent programming, Transactional memory, Mergeable objects, Relaxed Consistency

1 Introduction

In imperative programming languages, data structures are in general mutable, and updates are executed in-place. Therefore, the effect of an update is immediately reflected on the data structure. If the data structure is shared between multiple threads, the programmer must synchronize potential concurrent accesses to shared state to prevent memory corruption and often ensure progress by rendering updates visible to all threads.

Data structures that implement an abstract data type are often called objects (akin to objects in object-oriented programming). The correctness condition that is traditionally applied to shared concurrent objects is linearizability [12]. An object is linearizable if the result of concurrent operations is equivalent to some legal sequential execution of these operations. For example, concurrent increments of a linearizable counter have to be executed in a sequential order to prevent the loss of updates. This limits the inherent parallelism of an application and imposes high cost due to synchronization.

The final publication is availale at Springer via http://dx.doi.org/10.1007/978-3-319-26529-2_23

In contrast, (purely) functional programming languages, such as Haskell's core language, employ referential transparency. Pure functions do not update destructively. Thus, data structures are immutable by default. Though immutability implies thread-safety, it limits how concurrently running threads can exchange information. To overcome this restriction, Haskell offers different monadic interfaces supporting in-place updates and shared memory synchronization, the most prominent being: IO references (`IORef`), mutable references (`MVars`), and transactional variables (`TVars`).

For example, shared references to an (immutable) data item of type `a` can be encapsulated as IO references `IORef a`. IO references are intialized when created, and can be operated on using the following functions:

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

In addition, the function `atomicModifyIORef` allows to atomically apply a function on the referenced object in a thread-safe way:

```
atomicModifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

All calls to `atomicModifyIORef` need to be serialized to achieve atomicity for reading and modifiying the value. Haskell's `MVars` impose even more synchronization by blocking access to objects between calls to `takeMVar` and `putMVar`.

Transactional variables (`TVars`) provide a very similar interface, though their access is restricted to memory transactions. Transactions in this context are sequences of reads and writes that are transparently synchronized by the Software Transactional Memory (STM) system [9]. All operations on `TVars` within a transaction are executed atomically, and isolated from concurrent threads, thus providing a consistent view of the state. Yet again, when transactions concurrently operate on the same `TVar`, with at least one thread updating the variable, the operations conflict. Transactions thus fail their serializability certification check and have to re-execute [21].

Semantically, serializability is unnecessarily strict for a multitude of applications. For example, Fig. 1 shows the code snippet for kmeans clustering from the STAMP benchmark suite[16]. The K-means algorithm partitions n data points into k clusters such that the total distance for the data point to their respective cluster centre is minimized. Classical TMs serialize all transactions that access the same cluster center. However, the only requirement for correctness of the algorithm is that after all points are processed `newcenter` and `numElems` must contain the sum of all points and the number of points that belong to that cluster, respectively. Even with relaxed transactions [18, 19, 5], conflicts and hence aborts can still arise when updates cannot be serialized.

To optimize the synchronization on shared mutable data structures, we introduce *mergeable objects*. Instead of blocking or aborting updates to objects for serializability, updates are first applied to thread-local object versions. Instead of a get/set interface, mergeable objects therefore implement abstract data types with type-specific read and update operations. When committing the locally performed changes to shared memory, the different versions of an object are merged

```
assign clusters (x,y) = do
  let cluster = nearestPoint clusters (x,y)
  let center = newcentre cluster
  atomically $ do
    x' <- readTVar (xcord center)
    writeTVar (xcord center) (x' + x)
    y' <- readTVar (ycord center)
    writeTVar (ycord center) (y' + y)
    n <- readTVar (numElems cluster)
    writeTVar (numElems cluster) (n+1)

thread clusters points = mapM_ (assign clusters) points
```

Fig. 1. Kmeans: Computations by a thread using serializable transactions.

based on the object's semantics. Updates become visible to other threads only after the `merge` operation is called.

We propose Mergeable Transactional Memory (MTM) based on *mergeable objects* with relaxed consistency semantics (Section 3). Similar to snapshot isolation, MTM transactions read from a consistent snapshot and operate concurrently on shared objects. Instead of aborting and re-executing in case of conflicts, transactions commit their changes by *merging* states of concurrently updated objects. All updates from a transaction become visible together. An efficient merge operation enables MTM to execute multiple updates in parallel to other threads and execute the merge inside the critical section.

If `newcenter` and `numElems` in Fig.1 are represented using *mergeable counters* (Section.2) instead of raw integers, transaction can commit by merging the values, thus eliminating aborts. Moreover, we can rewrite the algorithm from Fig.1 to Fig.2 where a transaction (represented by `eventually`) process all points. MTM then executes it without synchronisation with other threads, thus allowing more parallelism. With serializable transactions, this would result in more conflicts and sequential execution of transactions.

```
thread clusters points = eventually $ mapM_ (assign clusters) points

assign clusters (x,y) = do
  let cluster = nearestPoint clusters (x,y)
  let center = newcentre cluster
  x' <- readCVar (xcord center)
  writeCVar (xcord center) (incrBy x' x))
  Y' <- readCVar (ycord center)
  writeCVar (ycord center) (incrBy y' y)
  c' <- readCVar (numElems cluster)
  writeCVar (numElems cluster) (incrBy c' 1)
```

Fig. 2. Kmeans: Larger transactions using MTM

The paper makes the following contributions:

```
class Counter {
 int x = 0;
 synchronized void inc() {
  x = x+1;
 }
}
```

```
type Counter = IORef Int
inc :: Counter -> IO ()
inc c = atomicModifyIORef (\x -> x+1,
      ()) c
```

Fig. 3. Linearizable Counter in a) Java and b) Haskell.

- We introduce the notion of *mergeable objects* and propose a classification of mergeability (Section 2).
- We introduce a programming model, MTM based on mergeable objects and transactions, and describe an algorithm for implementing the model (Section 3).
- We present a prototype implementation of MTM in Haskell (Section 4) and evaluate several use cases (Section 5).

2 Mergeable Objects

Instead of a get/set interface, mergeable objects implement abstract data types with type-specific operations. The update operations on mergeable object thus differ from that of linearizable objects in imperative programming; the latter provides in-place updates while operations on mergeable objects conceptually modify a local copy of the object. The result of updates on mergeable objects is visible to other threads only after the `merge` operation is called. Depending on the actual data types, mergeability of objects can be achieved in two ways, *semantic mergeability* and *structural mergeability*.

Semantic Mergeability: Exploiting object semantics to define the merge function has been successfully applied in Conflict-free Replicated Data Types (CRDTs)[23] in the context of distributed database systems. State-based CRDTs rely on lattice-based monotonic data values where the merge computes the least upper bound. Operation-based CRDTs re-execute updates that were issued on the local object instance against the global object, therefore requiring commutativity of concurrent updates to achieve consistency despite different orders of update application at the different replicas.

As an example, consider a shared counter that can be incremented concurrently by different threads. Figure 3 shows implementations with explicit synchronization in Java and in Haskell. For the mergeable counter in Fig. 4, the increment operations are collected and combined locally into a variable `v`, while a separate merge operation integrates the results of the local operations into the global state.

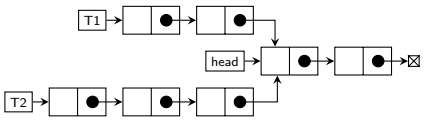
The merge operation for the counter in this case is trivial as all update operations commute. In general, CRDTs employ a number of mechanisms to achieve deterministic results for objects with non-commutative operations, e.g. maintaining tombstones for sets where elements can be added and removed. While

```
data Counter = Counter (IORef Int) Int
inc :: Counter -> Counter
inc (Counter g v) = Counter g (v+1)
read :: Counter -> Int
read (Counter g v) = do
  x <- readIORef g
  return (x+v)
merge :: Counter -> IO ()
merge (Counter g v) = do
  atomicModifyIORef (\x.x+v, ()) g
  return Counter g 0
```

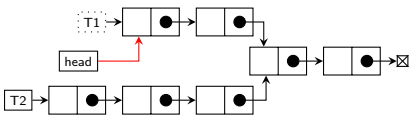
Fig. 4. Mergeable counter in Haskell.

CRDTs have been successful in avoiding costly synchronization in replicated data stores, employing the known specifications of CRDTs in multi-/many-core programs seems prohibitively expensive. In our work, we therefore focus on variants of CRDTs that are optimized for multi-/many-core programs.

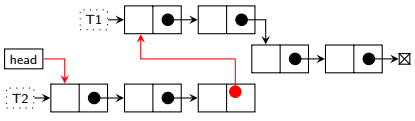
Structural Mergeability: While the merge operation for the counter can be implemented in a simple and efficient way, we have to employ different strategies for larger, composed data structures such as lists and sets. We adopt techniques that have been developed in the context of persistent data structure [7]. A persistent data structure is a mutable data structure that offers accessibility to multiple versions. This technique is widely used to implement purely functional data structures efficiently, in particular linked data structures such as lists, trees etc. When multiple threads modify the data structure, each thread executes updates on a thread-local version of the object, without the need for copying the entire data structure into thread-local storage. The *merge* operation is then reduced to adjust pointers in the local and global version to incorporate the updates in the global version; hence the name *structural* mergeability. The merge operation must preserve the semantics of the abstract data type by resolving potential semantic conflicts due to concurrent updates.



a. While threads T_1 and T_2 are accessing.



b. After thread T_1 merges.



c. After thread T_2 merges.

Fig. 5. Structural mergeability of bags.

As an example for structural mergeability, consider an add-only bag implemented as a persistent linked list. A bag is a *set* data structure allowing duplicate elements to be added. Here, threads can concurrently add elements without violating its semantical correctness. An implementation of a mergeable version of the bag is illustrated in Figure 5. The `head` points to the first node of the global version accessible to all threads. Adding an element to the bag adds a new node at the head of the linked list local to the thread. This results in a multi-headed list. Figure 5a shows the bag after threads T_1 and T_2 have added two and three elements, respectively, and before merging. The list pointed to by `T1` represents the view of the bag to thread T_1, similarly for `T2`. Both versions share the nodes of the elements that have been added before the threads started. When merging `T1`, it updates the global head to point to `T1` (Fig. 5b). When merging `T2`, it has to update both the global head and the local tail of `T2` to include changes of `T1` in the merge (Fig. 5c). The `merge` of an add-only bag is efficient because it requires manipulation of only two pointers.

3 Mergeable Transactions

To leave the triggering of the merge to the programmer poses a number of issues. For example, the programmer might forget to call the function at all. Merging updates to different objects is not atomic, thus possibly violating invariants. We therefore enhance the programming model for mergeable objects with a weak form of transactions.

Mergeable Transactional Memory (MTM) allows to compose operations on shared objects. Akin to STM, MTM guarantees atomicity, isolation and (weak) consistency for dynamic transactions. In contrast to STM, conflicting updates from concurrently executed transactions do not lead to aborts, but are merged during commit.

MTM does not provide serializability. Instead it provides a weak consistency described by the following properties:

- *commits* are totally ordered.
- *reads* and *updates* satisfies the program order.
- All *reads* from a transaction are guaranteed to observe a *consistent prefix* of the committed updates, and preceding updates from the same transaction.
- The *consistent prefix* includes previously committed updates from the current thread, thus obeying the program order.

3.1 Operational Semantics of MTM

To specify the consistency semantics of MTM transactions, we introduce a call-by-need core calculus, Λ_{MTM}, with an operation semantics based on transition rules. Figure 6 shows the syntax of Λ_{MTM}. It relies on disjoint sets of variables (Var) and references (Ref). A value is either a reference r, a mergeable value m, a function, a monadic return, an integer i or unit ().

$$
\begin{array}{lll}
x \in \text{Var}, r \in \text{Ref} & & \\
v \in \text{Val} & ::= & \boxed{\texttt{r}} \mid m \mid \lambda x.e \mid \texttt{return}\ e \mid i \mid () \\
e \in \text{Exp} & ::= & v \mid x \mid e\ e \mid e \gg\!= e \mid \texttt{forkIO}\ e \mid \texttt{eventually}\ e \mid \boxed{\texttt{commit}\ \Theta\ e} \mid \texttt{new}\ e \mid \\
 & & \texttt{read}\ e \mid \texttt{write}\ e \mid e + e \mid e * e \mid \ldots
\end{array}
$$

Fig. 6. Syntax of Λ_{MTM}.

Expressions are given as values, variables, function application, monadic bind, thread fork, MTM transactions, and arithmetic expressions. The expressions marked in gray do not appear in source programs, but represent dynamically generated locations and intermediate system states arising during commits.

$$
\begin{array}{lll}
t & \in \text{ThreadId} & \\
\Theta & \in \text{Heap} & = \text{Ref} \rightharpoonup \text{Exp} \\
P & \in \text{ThreadPool} & = \text{ThreadId} \rightharpoonup \text{Exp}
\end{array}
$$

Fig. 7. State-related definitions.

A program state $P; \Theta$ is a pair consisting of a thread pool P (partial mapping of thread identifiers to expressions) and a heap Θ. A reference l corresponds to an object allocated on the heap Θ. Dereferencing $\Theta(l)$ yields the associated object, while a heap update $\Theta[l \mapsto e]$ returns a heap that is identical to Θ, but maps l to e. Similarly, we denote updates in the thread pool P by $P\{t \mapsto e\}$.

The evaluation of a program starts in an initial state $\{t_0 \mapsto e\}; \emptyset$ with an empty heap and a main thread t_0. The evaluation stops when the program reaches a final state of the form $\{t_0 \mapsto v_0, \ldots, t_n \mapsto v_n\}; \Theta$. The reduction rules in Fig. 8 define the semantics of the language constructs. Each global reduction step $\rightarrowtail$ nondeterministically selects a thread from P, thus modeling an arbitrary thread scheduling.

The IO Monad is the top-level evaluation context. Rule IO-Monad enables the execution of reductions within the current context. Spawning a thread (rule Spawn) adds a new entry with a fresh thread identifier to the thread pool and returns unit to the parent thread. A transactional expression is evaluated against a copy of the current heap (rule Txn), possibly using multiple transactional transitions denoted by $\Rightarrow$.

Within a transaction, reading an object returns the value referenced in the heap (rule Read). Similarly, after applying the updates the resulting value is written back to the heap (rule Write), replacing the previous value. When allocating a new object, rule New ensures that the heap is extended using a fresh reference (i.e. one that has not been used in the heap or in concurrently running threads). The initial value of the object is then added to the transaction-local heap instance under the new reference.

Finally, an evaluated transaction is represented as a commit record consisting of the local heap copy, containing possible modifications, and the expression to

Evaluation contexts:

$$\mathbb{E} ::= []\ e \mid []\ggg= e \mid [] + e \mid v + [] \mid [] * e \mid ...$$

Expression evaluation $\rightarrow$:

$$(\lambda x.e)\ e' \rightarrow e[e'/x] \qquad \frac{e \rightarrow e'}{\mathbb{E}[e] \rightarrow \mathbb{E}[e']} \qquad i + j \rightarrow i \oplus j \qquad i * j \rightarrow i \otimes j$$

$$\texttt{return}\ e' \ggg= e \rightarrow e\ e'$$

Thread evaluation $\rightarrowtail$:

$$\frac{t' \text{ fresh}}{P\{t \mapsto \mathbb{E}[\texttt{forkIO}\ m]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{return}\ ()], t' \mapsto m\}; \Theta} \quad \textsc{Spawn}$$

$$\frac{e; \Theta \Rightarrow \texttt{return}\ e'; \Theta'}{P\{t \mapsto \mathbb{E}[\texttt{eventually}\ e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{commit}\ \Theta'\ e']\}; \Theta} \quad \textsc{Txn}$$

$$P\{t \mapsto \mathbb{E}[\texttt{commit}\ \Theta'\ e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[\texttt{return}\ e]\}; \Theta \uplus \Theta' \quad \textsc{Commit}$$

$$\frac{e \rightarrow e'}{P\{t \mapsto \mathbb{E}[e]\}; \Theta \rightarrowtail P\{t \mapsto \mathbb{E}[e']\}; \Theta} \quad \text{IO-}\textsc{Monad}$$

Evaluation steps in transaction $\Rightarrow$:

$$\frac{\Theta(r) = m}{\mathbb{E}[\texttt{read}\ r]; \Theta \Rightarrow \mathbb{E}[\texttt{return}\ m]; \Theta} \quad \textsc{Read}$$

$$\frac{\Theta(r) = m}{\mathbb{E}[\texttt{write}\ r]; \Theta \Rightarrow \mathbb{E}[\texttt{return}\ ()]; \Theta[r \mapsto m]} \quad \textsc{Write}$$

$$\frac{r \text{ fresh}}{\mathbb{E}[\texttt{new}\ m]; \Theta \Rightarrow \mathbb{E}[\texttt{return}\ r]; \Theta[r \mapsto m]} \quad \textsc{New}$$

$$\frac{e \rightarrow e'}{\mathbb{E}[e]; \Theta \Rightarrow \mathbb{E}[e']; \Theta} \quad \text{MTM-}\textsc{Monad}$$

Fig. 8. Operational Semantics for Λ_{MTM}.

be returned. Rule Commit then applies atomically the heap modifications to the globally shared heap and returns. The changes from the local heap copy Θ' are propagated to the current globally shared heap Θ by merging the individual entries with the thread-local ones. The function $\Cup\!\!\Cup :: \text{Heap} \times \text{Heap} \to \text{Heap}$ defines the heap merge:

$$(\Theta \Cup\!\!\Cup \Theta')(r) = \begin{cases} \text{merge } m\ n & \text{if } \Theta(r) = m, \Theta'(r) = n \\ m & \text{if } r \notin dom(\Theta), \Theta'(r) = m \\ n & \text{if } r \notin dom(\Theta'), \Theta(r) = n \end{cases}$$

3.2 Properties of MTM

Based on the operational semantics for Λ_{MTM}, we can now further characterize MTM transactions.

MTM allows non-serializable transactions. By rule Txn, the heap-modifying side-effects of a transaction `eventually` e are not immediately applied to the shared global state, but deferred to another reduction step under rule Commit. Depending on the scheduling, other transactions may also execute without committing their changes yet. If there are read-write dependencies between the transactions, it is not possible to construct a reduction sequence yielding the same final state.

All updates are eventually applied to the shared state. The type specific merge during the commit ensures that concurrent updates are merged deterministically into a consistent state of the object.

All updates performed by a transaction are made visible atomically. By rule Commit, all updates from a transaction are merged to the globally shared heap in one step, which guarantees atomicity.

All reads performed by a transaction appear to be executed at a single point of time. In addition to publishing the updates atomically, transactions are executed on a consistent snapshot; i.e. a snapshot in which either all updates from some transaction that committed before the snapshot time are visible or none. All read operations within a transaction are guaranteed to see the state of objects from a consistent snapshot taken at the time when the transaction started. Rule Txn shows that all operations inside a transaction are executed against the same state Θ. Although there could be concurrently executing transactions, their updates are not globally visible.

3.3 Algorithm

An algorithm for implementing the semantics of MTM transactions is given in Figure 9. To guarantee that a transaction never tries to read an object that has been modified by another transaction while executing (leading to a read-write conflict), we apply a multi-versioning scheme for mergeable objects. As previous studies have shown [18, 19, 5], multi-versioning of objects can be efficiently employed to achieve permissive transactions.

A shared mutable reference to a mergeable object which can be accessed in a MTM transaction is represented by *var*. A *var* references a list of versions. Each version contains a value of the object and its version identifier.

A transaction *txn* maintains a snapshot id *sid* in addition to a read and write sets which are represented as maps. When the transaction starts, its *sid* is assigned to be the current value of a *globalclock*. The operations of the transaction are executed on the snapshot identified by this *sid* which includes updates from all transactions committed before this time.

```
1: txn: {sid, Map writeset, Map readset}
2: var: {versions, lock}
3: versions : [{val, versionid}]
4: function BEGINTRANSACTION(txn)
5:     txn.sid ← globalclock
6:     txn.writeset ← ∅
7:     txn.readset ← ∅
8: end function
9:
10: function READ(var,txn)
11:     if txn.writeset.contains(var) then
12:         val ← txn.writeset.lookup(var)                ▷ read your own writes
13:     else if txn.readset.contains(var) then
14:         val ← txn.readset.lookup(var)
15:     else
16:         val ← READVERSION(var, txn.sid)
17:         txn.readset.add(var,val)
18:     end if
19:     return val
20: end function
21:
22: function WRITE(var, val, txn)
23:     txn.writeset.insert(var,val)
24: end function
25:
26: function COMMIT(txn)
27:     lockAll(txn.writeset)
28:     versionid ← globalclock++
29:     for all (var,val) ∈ txn.writeset do
30:         v' ← READLATESTVERSION(var)
31:         newval ← merge(v',val)
32:         WRITENEWVERSION(var, newval, versionid)
33:     end for
34:     unlockAll(txn.writeset)
35: end function
```

Fig. 9. MTM Algorithm

A *var* is accessed using the READ and WRITE methods. When reading, if the write set or read set contains a local copy of *var*, it is returned. Otherwise, the version corresponding to the transaction's *sid* is obtained and inserted in the read set. A new value of the object is written back to *var* using method WRITE, which inserts the value in the write set. Reading an object does not necessarily pass over the entire object. Depending on the actual representation of the object, a read might only be reading a reference.

When committing, the transaction acquires a lock on all objects in its write set. This ensures atomicity when two transactions tries to commit to same object. To prevent deadlocks, locks are obtained in a predefined order. Next, a new version id is generated from the current global clock value. The objects updated in the transaction are then merged with the latest version available, using the objects' merge method, hereby creating new versions.

Fig.10 shows the versioned read and write functions. The function WRITENEWVERSION adds the new value with its *vid* to the head of list of versions. Since *globalclock* is incremented during commit, the *sid* of a transaction always denotes the version id of a committed transaction or a concurrently committing transaction. When reading from the list of versions of a *var*, if the required version is not available, a concurrent transaction might be committing that version. Hence, it waits for the lock to be released before retrieving a version with an id equal or smaller than its *sid*. If the lock is released, it means that there is no other transaction which could potentially commit a version required by this transaction. This guarantees that a transaction always reads from a consistent snapshot identified by its *sid*.

4 MTM in Haskell

We implemented a prototype of MTM in Haskell. Harris et al. [9] have highlighted the benefits of Haskell's monadic type system for composing STM actions and restricting access to transactional variables to the STM monad. MTM is implemented analogously to the STM monad, though with different semantics.

For the MTM programming model, we provide an `MTM` monad (Fig.11). The shared mergeable objects used in MTM transactions are of type `CVar`; `CVar`[1] stands for convergent variables indicating that concurrent versions converge into a consistent state. Every operation executed on a `CVar` must be an MTM action. These actions can be sequentially combined using monadic bind. The function

```
eventually :: MTM a -> IO a
```

takes an MTM action, executes it, and returns the result. Using function `modifyCVar` to update a `CVar` guarantees that the mergeable values does not escape a transaction's scope.

The type specification ensures that mergeable objects are accessed only inside a MTM transaction. These objects must be of `class Mergeable` and define a `merge` function. Fig.12 shows the implementation of two mergeable objects. The

[1] The name `MVar` for mergeable variables is already used in Haskell.

```
 1: function readVersion(var, vid)
 2:     v ← var.versions
 3:     if v.head.versionid ≥ vid then
 4:         vr ← v.head
 5:     else
 6:         waituntil (not locked(var))      ▷ Wait for a concurrent committer to write
    required version
 7:         vr ← var.versions.head
 8:     end if
 9:     while vr.versionid > vid do
10:         vr ← vr.next
11:     end while
12:     return vr.val
13: end function
14:
15: function readLatestVersion(var)
16:     return var.versions.head.val
17: end function
18:
19: function writeNewVersion(var, val, vid)
20:     v ← newVersion(val,vid)
21:     var.versions.addHead(v)
22: end function
```

Fig. 10. Versioned read and write operations in MTM.

`Counter` contains two integers, one representing the global value and the other the thread-local increments. The `merge` adds the local increments to the global value `g` and resets the local increments to 0. The `LWWRegister` implements a last-writer-wins register, where the last merge overwrites the previous value.

Example The following example shows how to program with `CVar`s and the MTM monad in Haskell.

```
addToBag :: Int -> CVar (Bag Int) -> CVar (Counter) -> MTM [Int]
addToBag e bag size = do {
  ; b <- modifyCVar bag  (add e)
  ; s <- modifyCVar size (incrBy 1)
  ; return (toList b)
}
```

The function `addToBag` inserts an element to some bag and increments a counter representing the size of the bag. It returns then the elements from the bag in a list, including the added element `e`, but excluding elements that have been concurrently added. When calling the function using `eventually addToBag x b s` with some bag `b` and size counter `s`, the library guarantees that both shared objects are atomically updated and have consistent values.

```
data MTM a  = ...
data CVar a = ...

-- MTM Functions
eventually :: MTM a -> IO a
newCVar    :: Mergeable a => a -> MTM (CVar a)
readCVar   :: Mergeable a => CVar a -> MTM a
modifyCVar :: Mergeable a => CVar a -> (a -> a) -> MTM a

-- Mergeable Objects
class Mergeable a where
   merge :: a -> a -> a
```

Fig. 11. Interface for MTM in Haskell.

```
-- Mergeable Counter
data Counter = Counter Int Int
instance Mergeable Counter where
 merge (Counter g _) (Counter _ i) =
      Counter (g+i) 0

newCounter::Counter
newCounter = Counter 0 0
value :: Counter -> Int
value Counter g l = g+l
incrBy :: Int -> Counter -> Counter
incrBy i (Counter g l) =
 Counter g (l+i)
```

```
-- LWWRegister
type LWWReg = Int
instance Mergeable LWWReg where
    merge g l = l

-- Mergeable Bag
data Bag a = Bag [[a]] [a]
instance CRDT (Bag a) where
 merge (Bag g _) (Bag _ i) =
  Bag (i:g) []
newIntBag :: Bag Int
newIntBag = Bag [[]] []
add :: a -> Bag a -> Bag a
add e (Bag g l) = Bag g (e:l)
```

Fig. 12. Mergeable objects in Haskell.

5 Evaluation

To evaluate the applicability of MTM we ran microbenchmarks, comparing our MTM implementation as Haskell library with a library implementation of a STM algorithm based on 2-phase-commit (2PC) (similar to TL2 [6]) and GHC's STM implementation. GHC's STM is tightly integrated with the runtime system and employs a number of optimization techniques with respect to GC interaction and scheduling. To approximate the runtime overhead incurred by implementing MTM as a library, we use the 2PC implemenation as another point of comparison. All experiments were run on a Quad-core 2.4GHz Intel Xeon processor with two-way hyperthreading, under Linux 2.6.32-64-server Ubuntu x86_64 and GHC version 7.8.3. The results given are the averages taken over 10 runs for each benchmark.

Microbenchmarks: Counter and Bag In a first experiment, we compared the performance of a shared counter and bag under high contention. The STM variants implement the counter as a `TVar Int` and `TVar [Int]`, while MTM relies on a mergeable counter and bag, as introduced in Sec. 3. For the experiment, each thread repeatedly increments the same shared counter. In total, there were 2×10^6 increments distributed over the available number of threads.

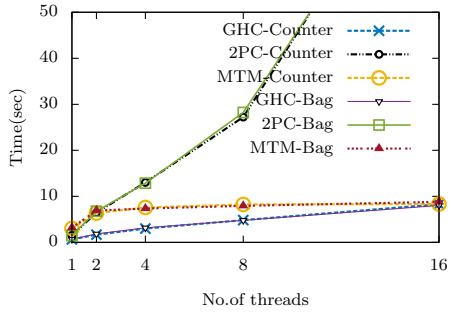


Fig. 13. Every thread updates once the same shared object in a transaction.

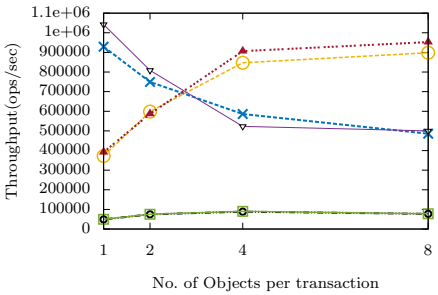


Fig. 14. Every thread updates M objects in a transaction

As Fig. 13 shows, the performance of the library version of STM degrades quickly while both MTM and GHC's STM handle the contention more gracefully.

To evaluate the throughput, we chose a workload where each transaction updates m randomly selected objects from a pool of n objects: the larger the pool (n), the lower the probability of contention; the larger the transaction size (m), the higher the probability of conflicts as it is more likely that transaction executions overlap. For $n = 8$ and various transaction size, MTM yields better performance than the STM implementations, even under low contention (Fig. 14).

Application: K-means: To see how actual applications benefit from the MTM programming model, we reimplemented the K-means benchmark from the STAMP benchmark suite [16] in Haskell described in Section.1.

For the version running GHC's STM and MTM a cluster centre is updated inside a transaction after processing every data point (here: 10^6 points). We also derived an alternative implementation to exploit the semantics of MTM, MTM-Opt, where all points assigned to some thread are processed together, and cluster centers are updated atomically. This version runs longer transactions, but has less frequent updates to cluster centers.

Both under high contention (Fig. 15) and low contention (Fig. 16), MTM-Opt outperforms GHC and MTM. In particular, MTM-Opt is scalable even under high contention in contrast to the other versions. The reason is that GHC's STM and MTM are blocking during commit, which prohibits scalability when the number of concurrent transactions is high. In the optimized version, commits are less frequent and transactions can run in parallel without the need for serializing the updates to shared memory.

6 Related Work

Software Transactional Memory: Relaxing strong guarantees such as serializability has been considered by different STMs. Multi-versioned STMs [5] and Snap-

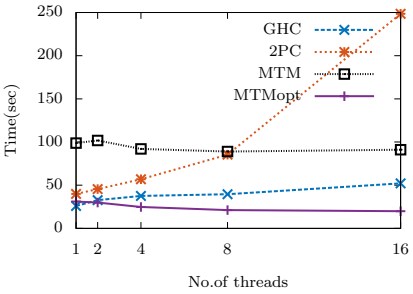


Fig. 15. K-means: High contention.

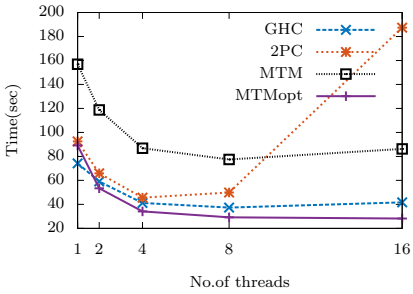


Fig. 16. K-means:Low Contention.

shot Isolation in STMs [19] allow read-only transactions to proceed without any conflicts. However, there may be aborts in case of write-write conflicts. Different apporaches have been proposed to avoid abort or restarting of whole transactions by delaying some computations [20] to commit time and re-executing parts of transaction [5]. Twilight STM [1] allows transaction-specific conflict handling when inconsistencies are detected in commit phase. MTM focuses on introducing the conflict handling mechanisms at the object level.

Composable Memory Transactions [9] provide primitives for making serializable transactions composable in Haskell. The authors describe the benefits of Haskell's type system and monads to achieve safety and composability of transactions. We have adopted these techniques to implement the MTM monad. However, as MTM transactions do never abort, we restrain from providing additional operations that support composability such as `retry` and `orElse`.

Transactional Boosting [11] is a method which allows operations on highly concurrent linearizable objects to execute using concurrent transactions, without the need for acquiring an exclusive lock on the object. A method's abstract lock issues a conflict only if two concurrent method invocations are non-commutative; therefore, concurrent commutative operation on an object can execute without aborting the transaction. Transactional boosting is a pessimistic approach by eagerly acquiring locks on the objects. Optimistic Transactional Boosting [10] is yet another methodology for transforming concurrent data structures to transactional objects. Both approaches take commutativity of operations as the base for detecting conflicts and thus achieving serializability. In contrast, MTM relies on object specific conflict resolution which may allow non-commutative operations to occur in parallel.

Burckhardt et al. [2, 15] propose a programming model for concurrent programs using revisions and isolation types. Each revision is considered a unit of concurrency. It executes operations on its local copy of the shared data concurrently to other threads. The modified data is visible to the main thread only after the revision is explicitly joined. The conflicts occurring due to concurrent updates are resolved using custom merge operation for cumulative types and joinee-wins strategy for versioned types. Though MTM and the revisions model share similar

semantics in executing operations on consistent snapshots and merging conflicting updates, they target different settings. The revisions programming model is a fork-join model and is suitable for short-running threads that operate mostly in isolation. MTM targets long running threads which need to periodically share data with other threads using transactional semantics.

The Phase Reconciliation mechanism [17] detects high contention on data items in in-memory databases. It then switches to a split phase where the transactions update a local per-core copy of the contended data in parallel. After the split phase, the per-core copies are merged and the transactions proceed to execute using classical concurrency control techniques. Whether transactions can be executed in the split phase, is decided based on the commutativity of operations, thus preserving sequential consistency.

Monotonic and Mergeable data structures: Conflict Free Replicated Data Types (CRDT) [22, 23] are replicated data types with mergeable semantics used in distributed database systems with eventual consistency. A state-based CRDT takes its values from a semi-lattice. Two states of the same objects are merged by taking the least upper bound in the semi-lattice. Op-based CRDTs, on the other hand, exploit commutativity of updates to deterministically converge the states of two replicas.

LVars [13, 14] are lattice-based data structures used for deterministic parallel programming in Haskell. The put operation changes an LVar's state in such a way that it monotonically increases in the lattice structure. Updates from concurrent threads on an LVar result in the same state, irrespective in which order they occur, thus guaranteeing determinism. The merge function computes always the least upper bound according to the lattice. LVars focus on deterministic and efficient execution for parallel programming models to support producer/consumer like application.

We believe that lattice-based data structures such as LVars and CRDTs are beneficial for deterministic merging and verifying the correctness of applications. However, it is not trivial how to construct efficient merge operation in order to be useful in an optimistic transactional model to improve performance. In this paper, we have discussed mergeable data structures which are not lattice structures.

Confluent persistent data structures [7, 8] allow operations on multiple versions of a data structure. These operations (e.g. concatenation, union) are constructed in a way such that previous versions are still accessible. Confluent persistent data structures are designed to perform these operations efficiently, in space and time. The applicability of these techniques in mergeable objects is an interesting topic for future work.

Distributed Systems: Weak consistency models such as eventual consistency and causal consistency are being widely researched and used in distributed systems. SwiftCloud [24] is a system that supports client-side replication and uses CRDTs to deterministically merge conflicting updates, while supporting Transactional causal+ consistency. Burckhardt et al. [3] present the idea of eventually consistent transactions and an implementation technique which provides these

semantics. Global Sequence Protocol [4] provides a programming model for replicated data stores and a weak consistency model relying on a global total order of updates. Though many recent works have studied eventual consistency in distributed database systems, few have addressed its applicability in multi-core programs. In this paper, we have discussed a way to achieve weak consistency in software transactions.

7 Conclusion

We have presented mergeable transactions as an alternative to the often too strict semantics of serializable transactions. Using abstract data type specifications, mergeable objects provide type-specific merge functions. We discussed semantic and structural mergeability as design alternatives for efficient merge functions and showed how to apply them to counters and bags. Our evaluation results on a prototype implementation in Haskell underline that for many workloads, especially on long running transactions, MTM outperforms standard STM, by eliminating the necessity for rollback.

In future work, we plan to extend MTM with a broader variety of mergeable objects and efficient implementation techniques. We also want to investigate the applicability of the concept in other programming paradigms, where more optimizations regarding the space and time complexity of mergeable objects are possible than in Haskell. It will be further interesting to study the possibility of co-existence of mergeable objects with non-mergeable objects in transaction, where aborts should be only induced when non-mergeable objects conflict.

Acknowledgments This research is supported in part by the European FP7 project 609 551 SyncFree.

References

1. Bieniusa, A., Middelkoop, A., Thiemann, P.: Brief announcement: Actions in the Twilight - concurrent irrevocable transactions and inconsistency repair. In: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010. pp. 71–72 (2010)
2. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 691–707. OOPSLA '10 (2010)
3. Burckhardt, S., Leijen, D., Fhndrich, M., Sagiv, M.: Eventually consistent transactions. In: Seidl, H. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 7211, pp. 67–86. Springer Berlin Heidelberg (2012)
4. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming (ECOOP 2015). vol. 37, pp. 568–590. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)

5. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172 – 185 (2006), special issue on synchronization and concurrency in object-oriented languages
6. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4167, pp. 194–208. Springer (2006)
7. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (Feb 1989)
8. Fiat, A., Kaplan, H.: Making data structures confluently persistent. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 537–546. SODA '01, Society for Industrial and Applied Mathematics (2001)
9. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 48–60. PPoPP '05 (2005)
10. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. In: Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 387–388. ACM (2014)
11. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 207–216. PPoPP '08 (2008)
12. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
13. Kuper, L., Newton, R.R.: Lvars: Lattice-based data structures for deterministic parallelism. In: Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 71–84. FHPC '13 (2013)
14. Kuper, L., Turon, A., Krishnaswami, N.R., Newton, R.R.: Freeze after writing: Quasi-deterministic parallel programming with lvars. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 257–270. POPL '14 (2014)
15. Leijen, D., Fahndrich, M., Burckhardt, S.: Prettier concurrency: Purely functional concurrent revisions. In: Proceedings of the 4th ACM Symposium on Haskell. pp. 83–94. Haskell '11, ACM, New York, NY, USA (2011)
16. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. pp. 35–46 (Sept 2008)
17. Narula, N., Cutler, C., Kohler, E., Morris, R.: Phase reconciliation for contended in-memory transactions. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 511–524. OSDI'14 (2014)
18. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in stm. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. pp. 16–25. PODC '10 (2010)
19. Riegel, T.: Snapshot isolation for software transactional memory. In: In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT06 (2006)
20. Ruan, W., Liu, Y., Spear, M.: Transactional read-modify-write without aborts. ACM Trans. Archit. Code Optim. 11(4), 63:1–63:24 (Jan 2015)
21. Scott, M.L.: Sequential specification of transactional memory semantics. In: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (Jun 2006)

22. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA (Jan 2011)
23. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. SSS'11 (2011)
24. Zawirski, M., Bieniusa, A., Balegas, V., Duarte, S., Baquero, C., Shapiro, M., Preguiça, N.: SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Research Report RR-8347 (Oct 2013)