Transactional Boosting for Haskell

André Rauber Du Bois, Maurício Lima Pilla, and Rodrigo Duarte

Computação - CDTec, UFPel, Pelotas-RS, Brazil
{dubois, pilla, rmduarte}@inf.ufpel.edu.br

Abstract. Transactional boosting is a methodology used to transform highly concurrent linearizable objects into highly concurrent *transactional* objects. In this paper we describe a STM Haskell extension that allows programmers to write *boosted* versions of highly concurrent abstract data types. Although the technique can only be applied to abstract types that have certain properties, when used correctly, we obtain transactional versions of existing types that are much faster than if they were implemented with pure STM Haskell.

1 Introduction

Transactional memory is a higher level alternative to lock based synchronization in concurrent programming. In this model, all accesses to shared memory are grouped as transactions that can execute concurrently and, if no conflicting accesses to shared memory are detected, may commit, or abort otherwise. Unlike lock based synchronization, transactions are composable and deadlock free.

Transactional memory implementations record the reads and writes that transactions perform and use this information to detect conflicts. A conflict occurs if two or more different transactions access the same memory location and at least one of these accesses is a write. Sometimes the conflict detecting mechanisms used by transactional memory systems are too conservative, and end up detecting false conflicts, or conflicts that might not violate the abstraction of a program. One classical example is two different transactions modifying different parts of a linked list [18, 12, 17, 15]. Although their actions do not conflict, there is a read/write conflict as one transaction is modifying a memory location that was read by another transaction. This kind of read/write conflict detection scheme can have a huge impact in performance when using certain kinds of linked data structures or generally when accessing memory locations that are subject to high contention. On the other hand, using lock based synchronization, or even *lock-free* algorithms, expert programmers can achieve a high level of concurrency at the cost of code complexity. One alternative to combine these two worlds is *transactional boosting* [12]. Transactional boosting is a methodology used to transform highly concurrent linearizable objects into highly concurrent *transactional* objects. It treats base objects as black boxes: we can make a boosted version of a linearizable concurrent library with no knowledge on how it is implemented. Transactional boosting is not applicable to every problem, basically, the methodology requires that method calls have *inverses*, and that only commutative method calls can occur concurrently.

STM Haskell [11] is a Haskell extension that provides the abstraction of *composable memory transactions*. The programmer defines *transactional actions* that are composable i.e., they can be combined to generate new transactions, and are first-class values. Haskell's type system forces threads to access shared variables only inside transactions. As transactions can not be executed outside a call to `atomically`, properties like *atomicity* (the effects of a transaction must be visible to all threads all at once) and *isolation* (during the execution of a transaction, it can not be affected by other transactions) are always maintained.

This text describes our experiments with transactional boosting in the context of the functional language Haskell. The contributions of this paper are as follows:

– We propose a new construct for STM Haskell that lets programmers implement boosted versions of existing fast concurrent Haskell libraries. We also describe how the new construct proposed here can interact with the high-level transactional primitives `retry` and `orElse` available in STM Haskell.
– We present the implementation of three classic transactional boosting examples using available concurrent Haskell libraries and the new proposed primitive. These examples demonstrate that our extension lets programmers transform existing fast implementations of linearizable structures into composable STM Haskell actions.
– Preliminary performance measurements are presented and indicate that, although transactional boosting can only be used in certain cases, when applied correctly, it is possible to achieve much faster STM actions than using the original STM Haskell. We believe that the primitive proposed here can be used by expert programmers do develop fast concurrent libraries for STM Haskell.

This paper is organized as follows: Section 2 reviews the main concepts of transactional memory and STM Haskell. Section 3 presents our extension to STM Haskell. Next, in Section 4, three examples of boosted versions of abstract data types are described and implemented using the new STM Haskell extension: a unique ID generator, a pipeline buffer, and a set data structure. Section 5 explains how the new primitive was implemented and shows the results of preliminary experiments. Finally, Section 6 discusses related work, and Section 7 concludes.

2 Transactional Memory and STM Haskell

2.1 Transactional Memory

Transactional memory was first described as a Hardware feature [13]. This paper focuses on *Software Transactional Memory* (STM), in which transactions are mainly implemented in software, with little hardware support, i.e., a compare and swap operation.

In an STM system, memory transactions can execute concurrently and, if finished without conflicts, a transaction may commit. Conflict detection may be *eager*, if a conflict is detected the first time a transaction accesses a value, or *lazy* when it occurs only at commit time. With eager conflict detection, to access a value, a transaction must acquire ownership of the value, hence preventing other transactions to access it, which is also called *pessimistic* concurrency control. With *optimistic* concurrency control, ownership acquisition and validation occurs only when committing. These design options can be combined for different kinds of accesses to data, e.g., eager conflict detection for write operations and lazy for reads. STM systems also differ in the granularity of conflict detection, being the most common word based and object based. In STM Haskell conflicts are detected at a TVar level, e.g, two different transactions writing to the same TVar (see next Section).

STM systems need a mechanism for version management. With *eager* version management, values are updated directly in memory and a transaction must maintain an *undo log* where it keeps the original values. If a transaction aborts, it uses the undo log to copy the old values back to memory. With *lazy* version management, all writes are buffered in a *redo log*, and reads must consult this log to see earlier writes. If a transaction commits, it copies these values to memory, and if it aborts the redo log can be discarded.

An STM implementation can be *lock* based, or *obstruction free*. An *obstruction free* STM does not use blocking mechanisms for synchronization and guarantees that a transaction will progress even if all other transactions are suspended. Lock based implementations, although offering weaker progress guarantees, are believed to be faster and easier to implement [8].

2.2 STM Haskell

STM Haskell extends Haskell with a set of primitives for writing memory transactions[11]. The main abstractions are *transactional variables* or `TVar`s, which are special variables that can only be accessed inside transactions. Figure 1 shows the main STM Haskell primitives. The `readTVar` function takes a TVar and returns a *transactional action* `STM a`. This action, when executed, will return a value of type `a`, i.e., the TVar's content. Similarly, `writeTVar` takes a value of type `a`, a TVar of the same type and returns a STM action that when executed writes into the TVar.

These transactional actions can be composed together to generate new actions using the basic monadic composition constructs (bind (`>>=`), then (`>>`) and `return`), or by using the syntactic sugar provided by the `do` notation.

The `retry` primitive is used to abort and re-execute a transaction once at least one of the memory positions it has read is modified. `orElse` is a composition operator, it takes two transactions as arguments, if the first one retries then the second one is executed. If both fail the whole transaction is executed again.

An STM action can only be executed with a call to `atomically`:

```
writeTVar :: TVar a -> a -> STM ()
readTVar :: TVar a -> STM a
retry :: STM ()
orElse :: STM a -> STM a -> STM a
atomically :: STM a -> IO a
```

Fig. 1. STM Haskell interface

```
atomically (transferMoney tvar1 tvar2 100.00)
```

It takes as an argument a transactional action (`STM a`) and executes it atomically with respect to other concurrently executing transactions.

3 Transactional Boosting for STM Haskell

Transactional boosting is a methodology used to transform existing highly concurrent linearizable objects into highly concurrent transactional objects. It treats existing objects as black boxes, and performs both conflict detection and logging at the granularity of entire method calls. It can not be applied to every object but to objects where commutative method calls can be identified, and which reasonably efficient inverses exist or can be composed from existing methods [12].

To write a transactional boosted version of an abstract type, the operations associated with this type must have inverses. Hence the STM system must provide ways of registering user defined handlers to be called when the transaction aborts or commits. If a transaction aborts, it must undo the changes it has done to the boosted object and if it commits it must make these changes visible to the rest of the system.

We propose a simple new STM Haskell primitive to build transactional versions of abstract data types:

```
boost :: IO (Maybe a) -> ((Maybe a)-> IO ()) -> IO () -> STM a
```

The `boost` primitive can be used to create a new transactional version of existing Haskell libraries. It wraps a function in a Haskell STM action, providing ways to call this function inside a transaction and also for undoing its effects in the case of an abort. It takes as arguments:

- an *action* (of type `IO (Maybe a)`) that is used by the underlying transactional system to invoke the original function. When the original function is called it might return a result of type `a`, or maybe for some reason the original function could not be called, e.g., an internal lock could not be acquired, in which case the action should return `Nothing`. Hence the return type is `Maybe a`

– an *undo* action (of type `Maybe a -> IO ()`) used to undo the function call in case of an abort. If we want to abort a `STM` action, we must know what was the outcome of executing it, e.g., if we want to undo deleting value x from a set, we must insert x again into the set. As the outcome of executing a transactional action is only known at execution time, the `undo` action takes as argument the value returned by the first argument of `boost`.
– a *commit* action (of type `IO ()`) that is used to commit the action done by the boosted version of the original function, i.e., make it visible to the rest of the system

`boost` returns a new `STM` action that is used inside transactions to access the wrapped function.

4 Examples

This section describes the implementation of three classic transactional boosting examples using existing concurrent Haskell libraries plus the new primitive proposed for STM Haskell. The examples are a *Unique ID Generator* (Section 4.1), a *Pipeline Buffer* (Section 4.2), and a *Set* (Section 4.3).

4.1 Unique ID Generator

We start with a simple example, a unique ID generator. Generating unique IDs in STM systems is problematic. The `generateID` function would typically be implemented using a shared counter that is incremented at each call. As different transactions are trying to increment and read a shared location (the counter), transactional memory implementations detect read/write conflicts and abort at least one of the transactions. The problem is that those may not be real conflicts: as long as different calls to `generateID` return different numbers, we do not care, for example, if the values generated follow the exact order in which the counter was incremented.

A simple and fast thread safe unique ID generator could be implemented using a Fetch-and-Add or Compare-and-Swap (CAS) operation, available in most standard multi-core processors. Haskell, provides an abstraction called `IORef` that represents a mutable memory locations [16]. An `IORef a` is a mutable memory location that may contains a value of type `a`. The library [3] lets programmers perform machine-level compare and swap operation on `IORef`s. Thus, a unique ID generator can be implemented as follows:

```
type IDGer = IORef Int

newID :: IO IDGer
newID = newIORef 0

generateID ::  IDGer -> IO Int
generateID idger = do
```

```
    v <- readIORef idger
    ok <- atomCAS idger v (v+1)
    if ok then return (v+1) else generatetID idger
```

Under transactional boosting, the ID generator must follow the specification in Figure 2.

Function Call	**Inverse**
`generateID`	noop

Commutativity

`x <-generateID` $\Leftrightarrow$ `y <-generateID`	$x \neq y$
`x <-generateID` $\not\Leftrightarrow$ `y <-generateID`	$x = y$

Fig. 2. Unique ID Generator specification

When a transaction that called `generateID` aborts, ideally the ID returned by the call should be returned to a pool of unused IDs. On the other hand, since `generateID` always returns an unique value, the IDs generated are disposable.

Using transactional boosting, the `generateID` function could be implemented as follows:

```
generateIDTB ::  IDGer -> STM Int
generateIDTB idger =  boost ac undo commit
     where
      ac = do
         id <- UniqueIDCAS.generateID idger
         return (Just id)
      undo _ = return ()
      commit = return ()
```

Now `generateIDTB` is an `STM` action that, when executed calls the `ac` action which simply uses the CAS version of the generator to increment the `IORef` counter. If the transaction aborts, or the transaction commits, nothing has to be done, hence the `undo` and `commit` actions are empty.

4.2 Pipeline Buffer

Pipeline is an abstraction where there is a chain of data processing threads that communicate by bounded queues, or buffers. Each thread is in charge of a stage in the pipeline and consumes data from a buffer, processes it, and writes the new data to a different buffer.

A buffer must provide two functions: `offer` used to add a value to the buffer and `take`, that consumes a data item. To implement a buffer using transactional boosting, we need a double-ended queue as it provides inverses for `take` and `offer`. Here we use a thread safe double-ended queue implemented by Ryan Newton [4], and follow the specification in Figure 3:

Function Call	**Inverse**
`offer buf x`	tryPopL buf
`x <-take buf`	pushR buf x

Commutativity

`offer buf x` $\Leftrightarrow$ `y <-take buf`, buffer non-empty

`offer buf x` $\not\Leftrightarrow$ `y <-take buf`, otherwise

Fig. 3. Pipeline Buffer specification

```
data TBBuffer a = Q (SimpleDeque a) (IORef Int)

newTBBuffer :: TBBuffer a
newTBBuffer = do
      q<-newQ
      ioref <- newIORef 0
      return (Q q ioref)

offer :: TBBuffer a -> a -> STM ()
offer (Q c ioref) v = boost ac undo commit
      where
        ac = do
           pushL c v
           return (Just ())
        undo _ = do
             mv <- tryPopL c
             case mv of
                 Just v -> return ()
        commit = do
             v <- readIORef ioref
             ok<- atomCAS ioref v (v+1)
             if ok then return () else commit
```

A `TBBuffer` is represented by a double-ended queue and an `IORef` that contains the size of the buffer. The `offer` function must use `pushL` to add a new value to the queue. If a transaction aborts, the data that was inserted in the queue must be eliminated using `tryPopL`. If the transaction commits, the only thing to be done is to increment the buffer size thus making the new item visible to the transaction that is consuming values. The `take` function uses `tryPopR` to consume data from a buffer:

```
take :: TBBuffer a -> STM a
take (Q c ioref) = boost ac undo commit
   where
     ac = do
        size<-readIORef ioref
        if size ==0 then return Nothing
                    else do
```

```
                  decIORef ioref
                  tryPopR c
undo v = case v of
           Nothing -> return ()
           Just x -> do
                  incIORef ioref
                  pushR c x
commit = return ()
```

If there are not enough items then the transaction aborts by returning `Nothing`. Otherwise it decrements the buffer counter using CAS (with the `decIORef` function) and consumes an item (`tryPopR`). The `undo` action must increment the counter using CAS (`incIORef`) and return the value taken back to the buffer.

4.3 Set

A set is a collection of distinct objects. An implementation of a set usually provides three functions, `add`, `remove` and `contains`.

Function Call	**Inverse**
`add set x` / `False`	noop
`add set x` / `True`	`remove set x` / `True`
`remove set x` / `False`	noop
`remove set x` / `True`	`add set x` / `True`
`contains set x` / _	noop

Commutativity

`add set x` / _ $\Leftrightarrow$ `add set y` / _, $x \neq y$

`remove set x` / _ $\Leftrightarrow$ `remove set y` / _, $x \neq y$

`add set x` / _ $\Leftrightarrow$ `remove set y` / _, $x \neq y$

`add set x` / `False` $\Leftrightarrow$ `remove set x` / `False` $\Leftrightarrow$ `contains set x` / _

Fig. 4. Set specification

As there is no linearizable implementation of a set data structure available for Haskell, the boosted set described here uses a thread safe linked list, described in [18] (see Figure 5). When implementing a boosted version of a set, we must guarantee that if one transactions is working on a key, no other transaction can be using the same key (see Figure 4). We can achieve this by using key-based locking [15]. Key based locking can be implemented using a hash table to associate a lock with each key. The problem is that currently there are no thread safe Hash tables available for Haskell. To implement key locking we use an STM Hash table from the Haskell STM benchmark suite [1]:

```
data KLock = KLock (THash Int Lock)
data Lock = Lock (IORef Integer) (IORef Integer)

newKLock :: IO KLock
newKLock = do
      hasht <- atomically (new hashInt)
      return (KLock hasht)
```

A `KLock` is a hash table that maps `Int`s (keys) to locks. Locks are represented by two `IORef`s. The first is a versioned lock [9]: if it contains `0` the locks is free, if it contains a transaction ID the lock is locked. The second `IORef` counts how many times the lock holder locked the same key. The `newKLock` function creates a new `KLock` by simply creating an empty Hash table. A key can be locked with the `lock` function:

```
lock :: KLock -> Int -> IO Bool
lock (KLock ht) key = do
   mior <- atomically (Data.THash.lookup ht key)
   myId <- getTransID
   case mior of
       Just (Lock ior counter) -> do
                 v <- readIORef ior
                 if (v == 0)
                     then do locked<- atomCAS ior 0 myId
                             case locked of
                                   True -> do
                                      plusOne counter
                                      return True
                                   False -> return False
(...)
```

It takes a `Klock` and a `key` as arguments. If there is already a lock associated with the key, and the lock is free (i.e., contains `0`), it tries to acquire the lock using `atomCAS`. If successful, it increments the counter. If the current transaction already holds the lock, it just needs to increment the counter one more time. If there is no lock associated with the key, a new one must be created and inserted into the hash table:

```
       Nothing -> do
                 ior <- newIORef myId
                 counter <- newIORef 1
                 ok <- atomically (insert ht key (Lock ior counter))
                 if ok then return True else (lock alock key)
```

The `unlock` function

```
unlock :: KLock -> Int -> IO Bool
```

finds the lock associated with the key, and if the current transaction holds the lock it simply decrements the lock's counter. If the counter gets to zero, then the lock is freed using CAS.

Now, a boosted version of a `Set` data structure can be represented by a `KLock` and a linked list:

```
newList :: IO (ListHandle a)
addToTail :: Eq a => ListHandle a -> a -> IO ()
find :: Eq a => ListHandle a -> a -> IO Bool
delete :: Eq a => ListHandle a -> a -> IO Bool
```

Fig. 5. Interface for the concurrent linked list described in [18]

```
data IntSet = Set KLock (ListHandle Int)
```

To `add` a `key` to a set, we must acquire the lock associated with it and then insert the key in the linked list. As the linked list may contain duplicates, we must also make sure that the `key` is not already in the list:

```
add:: IntSet -> Int -> STM Bool
add (Set klock list) key =  boost ac undo commit
   where
     ac =  do
        ok<-lock klock key
        case ok of
           True -> do found <- CASList.find list key
                      if found then return (Just False)
                               else do
                                   CASList.addToTail list key
                                   return (Just True)
           False -> return Nothing
```

If a key was inserted and the transaction aborts, the same key must be deleted from the list and the lock freed. If the transaction commits, the only thing to do is to free the lock:

```
     undo v = do
        case v of
           Just True -> do
                CASList.delete list key
                unlock klock key
                return()
           Just False ->  do
                unlock klock key
                return()
           Nothing -> return ()

     commit = do
        unlock klock key
        return ()
```

To `remove` a key, we must acquire the right lock and then delete the key from the linked list:

```
remove :: IntSet -> Int -> STM Bool
remove (Set klock list) key = boost ac undo commit
    where
     ac =  do
        ok<-lock klock key
        case ok of
           True -> do v<-CASList.delete list key
                      return (Just v)
           False -> return Nothing
     undo ok = do
        case ok of
           Just True -> do
                CASList.addToTail list key
                unlock klock key
                return ()
           Just False-> do unlock klock key
                           return ()
           Nothing -> return ()
     commit = do unlock alock key
                 return ()
```

To `undo` a successful `remove`, the `key` must be inserted back again in the list and the lock freed. As before, to `commit` the operation we just need to liberate the lock.

The `contains` operation

```
contains :: IntSet -> Int -> STM Bool
```

is simpler and the code is omitted. As `contains` does not changes the internal list, if the transaction aborts or commits, nothing has to be done except liberating the lock associated with the searched key.

5 Implementation and Preliminary Experiments

5.1 Prototype Implementation

To test the examples presented in this paper, we extended TL2 STM Haskell [7, 2], a high-level implementation of STM Haskell that uses the TL2 [6] algorithm for transactions. The TL2 implementation uses lazy conflict detection with optimistic concurrency control as happens in the original C implementation of STM Haskell that comes with the GHC compiler[10].

As the TL2 library is implemented completely in Haskell, it is easier to extend and modify. It also provides reasonable performance for a prototype: programs run 1 to 16 times slower than the C implementation, with factors of 2 and 3 being the most common. Experiments using the Haskell STM Benchmark suite also demonstrate that the library provides scalability similar to the original C run-time system [7].

In TL2 STM Haskell, as in other implementations of STM Haskell [14, 5], the STM data type is represented as a state passing monad. Thus, an `STM a` action in the monad is in fact a function that takes the state of the current transaction (e.g. its read and write logs) as an argument, executes a computation in the transaction (e.g. reads a `TVar`), and returns a new transaction state and a result of type `a`:

```
data STM a = STM (TState -> IO (TResult a))
```

The `TResult` type describes the possible outcomes of executing a transaction:

```
data TResult a = Valid TState a | Retry TState | Invalid TState
```

Our implementation of the `boost` extends the type that represents the transaction state (`TState`) with two `IO ()` actions:

```
data TState = Tr {
(...)
   tbUndo :: IO (),
   tbCommit :: IO ()
}
```

The `tbUndo` and `tbCommit` actions are constructed during the execution of a transaction, and the first is executed only if a transaction aborts, i.e., finishes with an `Invalid` state, and the later is executed only if a transaction commits. Hence, the implementation of `boost` becomes simply:

```
boost :: IO (Maybe a) -> (Maybe a-> IO ()) -> IO () -> STM a
boost mac undo commit = STM $ \tState -> do
  r <- mac
  case r of
    Just v -> return (Valid tState{tbUndo=undo (Just v)>>(tbUndo tState),
                                  tbCommit = commit>>(tbCommit tState)} v)
    Nothing -> return (Invalid tState{tbUndo=undo Nothing>>(tbUndo tState)})
```

The *then* monadic combinator (`>>`) is used to add the new actions to the current `tbUndo` and `tbCommit` IO actions.

The design and implementation of the `retry` construct is tied closely to the concept of `TVar`: when a transaction retries, it will not execute again until at least one of the tvars it read is updated. We would like to extend the concept further, for example, in the `Set` example, if we can not acquire the lock for a key, we could retry or abort the transaction and only start it again once the lock is freed. It is difficult to predict all possible cases in which we want a transaction to be restarted and we also do not want to include in the design primitives that are too low level. Hence we decided for a simpler approach: when a transaction that executed TB actions calls `retry`, the transaction is stopped, `tbUndo` is executed, and the transaction waits on the `TVar`s it has read.

We also extend the behavior of the `orElse` construct to support transactional boosting. A call to `orElse t1 t2` will first save the `tbUndo` and `tbCommit` actions of the current transaction's state and execute `t1` with new and empty `tbUndo` and `tbCommit`. If `t1` retries, its newly created `tbUndo` is called and the transaction continues by executing `t2` with the saved actions. If `t1` finishes without retrying, then both new and saved TB actions are combined and `orElse` returns the result of executing `t1`.

5.2 Experiments

The experiments were executed on an Intel Core i7 processor at 2.1 GHz and 8 GiB of RAM and the times presented are the mean of 10 executions. The operating system was Ubuntu 12.04, with Haskell GHC 7.4. The Core i7 processor has 4 real cores plus 4 more with hyper threading.

Figure 6 compares three implementations of the unique ID generator: `ID STM` that uses the C implementation of STM Haskell that comes with GHC, `ID CAS` that uses CAS to increment the ID counter, and `ID TB` that is the boosted implementation described in Section 4.1. The experiment executes 10 million calls to `generateID` in total, dividing these operations by the threads available.

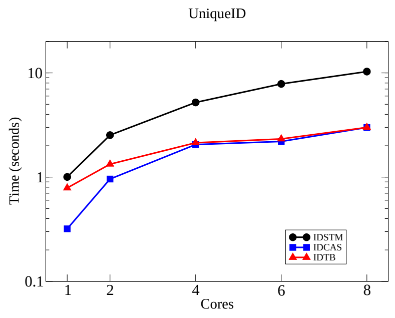

Fig. 6. UniqueID execution times

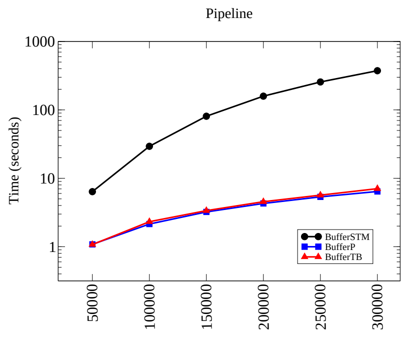

Fig. 7. Pipeline Buffer execution times

The second experiment (Figure 7) creates two threads, a producer and a consumer, that communicate through a pipeline buffer each executing the number of operations described on the X axis. Three different implementations of the pipeline buffer are compared: `Buffer STM` that uses the `TChan` STM library that comes with GHC, `BufferP` is the thread safe deque implemented by Ryan Newton, and `BufferTB` is the boosted version described in Section 4.2.

To benchmark the set data structure, we randomly generated test data consisting of an initial list of 2000 elements and 8 lists of 2000 operations. Each list of operations is given to a different thread and we vary the number of cores used from 1 to 8, as can be seen in Figure 8. Note that the Y axis is a logarithmic scale. Two implementations of Set are compared, one that uses an ordered linked list of TVars and is compiled using the original C STM Haskell (`GHC-STM`), and the boosted version described in Section 4.3 (`TB`).

As can be seen by these preliminary experiments, even though our prototype implementation uses the slower TL2 STM Haskell implementation, all TB examples are still much faster than the original STM Haskell implemented in C.

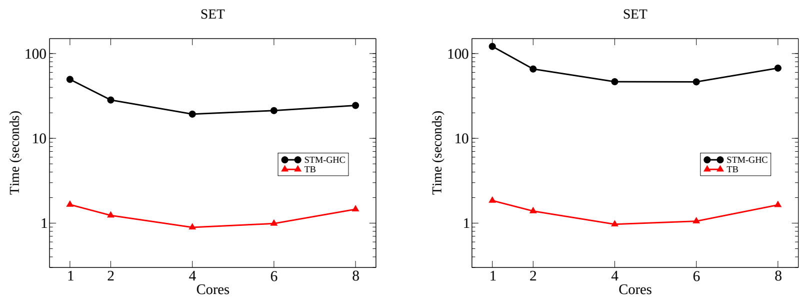

Fig. 8. 8 threads each executing 2000 operations in a Set (On the left: 40% `adds` and `removes` + 60% `contains`, On the right: 65% `adds` and `removes` + 25% `contains`)

This happens because the overhead introduced by transactional layer added to the fast parallel implementations is not enough to harm the performance.

6 Related Work

Other works have extended the original STM Haskell design with *escape* primitives that allow the programmer to change the state of the current transaction hence avoiding false conflicts. The `unreadTVar` [17] construct is an extension to the original STM Haskell interface that improves execution time and memory usage when traversing transactional linked structures. It is used to eliminate from the read set values that are far behind from the current position. For the same purpose in [18] the authors propose `readTVarIO` that reads a TVar without adding a log entry for this read into the current transaction's read set. If the lock of the TVar is being held by a transaction trying to commit, `readTVarIO` blocks until the lock is freed. *Twilight* STM in Haskell [5] extends STM Haskell with safe embedding of I/O operations and a repair facility to resolve conflicts. Twilight splits the code of a transaction into two phases, a functional atomic phase (that behaves just like the original STM Haskell), and an imperative phase where Twilight code can be used. A prototype implementation of Twilight Haskell in Haskell is provided but no performance figures are given.

All these new primitives are used to implement new data types in a way that false conflicts are avoided. The method presented here is used as a way of accessing existing fast highly concurrent structures inside transactions.

7 Concluding Remarks

We have described an STM Haskell extension to write transactional boosted versions of abstract data types. We extended STM Haskell with a single new primitive and have described three examples of its use. Preliminary experiments show that the new primitive can help programmers to write faster transactional libraries if used in the right context. Transactional boosting is low level concurrent programming and can lead to all the problems associated with concurrency, such as deadlocks [12]. We believe that the primitive presented here can be used by expert programmers to write fast concurrent libraries for Haskell.

We can also use `boost` to implemented transactional versions of sequential structures that are not available in STM libraries, however there will be a performance impact depending on the approach used to protect these structures (e.g., a single lock) plus the overhead of the transactional boosting system, as can be seen in the experiments in Section 5.

References

1. The Haskell STM Benchmark. WWW page, http://www.bscmsrc.eu/software/haskell-stm-benchmark, October 2010.
2. TL2 STM Haskell. http://sites.google.com/site/tl2stmhaskell/STM.hs, Feb 2011.
3. Data.CAS. http://hackage.haskell.org/package/IORefCAS-0.1.0.1/docs/src/Data-CAS.html, October 2013.
4. Thread Safe Deque. https://github.com/rrnewton/haskell-lockfree, October 2013.
5. A. Bieniusa et al. Twilight in Haskell: Software Transactional Memory with Safe I/O and Typed Conflict Management. In *IFL 2010*, sep 2010.
6. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC 2006*, pages 194–208, 2006.
7. A. R. Du Bois. An implementation of composable memory transactions in Haskell. In *Software Composition 2011*, LNCS 6708, Swtzerland, 2011. Springer-Verlag.
8. R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.
9. T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.
10. T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell Workshop 2005*, pages 49–61. ACM Press, September 2005.
11. T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*. ACM Press, 2005.
12. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216. ACM, 2008.
13. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. May 1993.
14. F. Huch and F. Kupke. A high-level implementation of composable memory transactions in concurrent Haskell. In *IFL*, pages 124–141, 2005.
15. Y. Ni et al. Open nesting in software transactional memory. In *PPoPP*, pages 68–78. ACM, 1 2007.
16. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.
17. N. Sonmez et al. Unreadtvar: Extending Haskell software transactional memory for performance. In *Trends in Functional Programming*. Intellect Books, 2008.
18. M. Sulzmann, E. S. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in Haskell. *SIGPLAN Not.*, 44(5):11–20, 2009.