# Comparing the performance of concurrent linked-list implementations in Haskell

Martin Sulzmann
IT University of Copenhagen, Denmark
martin.sulzmann@gmail.com

Edmund S. L. Lam
National University of Singapore, Singapore
lamsoonl@comp.nus.edu.sg

Simon Marlow
Microsoft Research Cambridge, UK
simonmar@microsoft.com

## Abstract

Haskell has a rich set of synchronization primitives for implementing shared-state concurrency abstractions, ranging from the very high level (Software Transactional Memory) to the very low level (mutable variables with atomic read-modify-write).

In this paper we perform a systematic comparison of these different concurrent programming models by using them to implement the same abstraction: a concurrent linked-list. Our results are somewhat surprising: there is a full two orders of magnitude difference in performance between the slowest and the fastest implementation. Our analysis of the performance results gives new insights into the relative performance of the programming models and their implementation.

Finally, we suggest the addition of a single primitive which in our experiments improves the performance of one of the STM-based implementations by more than a factor of 7.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

***General Terms*** Languages, Performance

## 1. Introduction

As programmers we are faced with a choice between a variety of synchronization models when implementing shared-state concurrency abstractions. Each model typically offers a unique trade-off between composability, scalability, and absolute performance. Knowing which programming model to use is a skill in itself.

In this paper we study the differences between the synchronization models offered by Haskell [11], in particular those implemented in the Glasgow Haskell Compiler (GHC) [4]. The three main synchronization models supported by GHC are

- STM: Software Transactional Memory [5].
- `MVar`: an elementary lock-based synchronization primitive [12].
- `IORef` and `atomicModifyIORef`: low-level synchronization using mutable variables and an atomic read-modify-write operation.

We are interested in the following questions:

- How much overhead is caused by each synchronization model in GHC?
- How well does each synchronization model scale with the number of processor cores?
- How many processor cores are needed to out-perform a sequential implementation?
- Which other aspects, if any, may influence the choice of synchronization model?

There is very little previous work that addresses these issues, and what there is tends to focus purely on STM performance and scalability (see Section 5).

In summary, we make the following contributions:

- We apply established methods to implement a highly concurrent linked-list using each of GHC's synchronization models (Section 3). We have two STM-based algorithms, and since STM is also able to model the two other forms of synchronization we have a total of six implementations to compare.
- We perform detailed experiments to evaluate the relative trade-offs in each implementation and also draw a comparison with a sequential implementation (Section 4).
- We propose the addition of a single primitive which improves the performance of one of the STM-based algorithms by more than a factor of 7 (Section 4.1).

We discuss related work in Section 5. The upcoming section gives an introduction to the synchronization primitives available in GHC.

## 2. Synchronization Primitives in GHC

In the following section we describe the three basic models of concurrent synchronization available in GHC, starting from the highest-level (STM), and ending with the lowest-level (mutable variables with atomic read-modify-write).

### 2.1 Software Transactional Memory (STM)

STM was added to GHC in 2005 [5] as a way to program concurrent synchronization in a way that is *composable*, in the sense that operations that modify shared state can safely be composed with other operations that also modify shared state.

The basic data object is the `TVar`, or transactional variable. A transaction is a computation performed over a set of `TVars`, yielding a result; each transaction is performed *atomically*. The implementation technique that makes transactions viable is *optimistic concurrency*; that is, all transactions run to completion under the assumption that no conflicts have occurred, and only at the end of the transaction do we perform a consistency check, retrying the transaction from the start if a conflict has occurred. This is "optimistic"

*DAMP'09,* January 20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-417-1/09/01...$5.00.

in the sense that it performs well if conflicts are rare, but poorly if they are common. If conflicts are common (many transactions modifying the same state), then optimistic concurrency can have worse performance that just sequentializing all the transactions using a single global lock.

It is important that a transaction can modify only TVars, because if the transaction is found to be in conflict then its effects must be discarded, and this is only possible if the effects are restricted to a known class of effects that can be undone. In Haskell we have the luxury of being able to restrict effects using the type system, and so for STM there is a new monad, STM, whose only stateful effects are those that affect TVars:

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

Here is a simple STM example to model an ”atomic” bank transfer.

```
transfer :: TVar Int -> TVar Int -> Int -> IO ()
transfer fromAcc toAcc amount =
 atomically $ do
   f <- readTVar fromAcc
   if f <= amount
     then retry
     else do
       writeTVar fromAcc (f - amount)
       t <- readTVar toAcc
       writeTVar toAcc (t + amount)
```

We transfer amount currency units from fromAcc to toAcc. If the balance of fromAcc is insufficient we simply retry. That is, we abort the transaction and try again. There is no point in re-running the transaction if fromAcc has not changed. Hence, the transaction simply blocks until fromAcc has been updated.

To summarize, programming with STM has two advantages. It is straightforward to *compose* ”little” STM computations such as

```
      writeTVar fromAcc (f-amount)
```

with other STM computations to form a ”bigger” STM computation (the actual bank transfer). Furthermore, the programmer does not need to worry in which order to acquire (i.e. lock) the two accounts. STM computations are executed *optimistically*. For our example, this means that if there are two concurrent transfers involving the same set of accounts, (1) neither transaction will block the other, instead (2) the STM run-time optimistically executes both transactions but only one of them can commit and the other is retried. Thus, STM avoids common pitfalls when programming with ”locks” where the programmer carefully has to acquire resources in a specific order to avoid deadlocks.

Next, we take a look at MVars, an elementary lock-based synchronization primitive in Haskell.

## 2.2 MVars

The intermediate-level synchronization method supported by GHC is the MVar, with the following operations:

```
newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

An MVar is like a one-place channel; it can be either full or empty. The takeMVar operation returns the value if the MVar is full or blocks if the MVar is empty, and putMVar fills the MVar if it is empty or blocks otherwise.

It is possible to implement the semantics of MVar using STM, although without some of the useful operational properties of the native implementation. Here is the STM implementation of MVar and takeMVar:

```
newtype MVar a = MVar (TVar (Maybe a))

takeMVar :: MVar a -> IO a
takeMVar (MVar tv) =
  atomically $ do
    m <- readTVar tv
    case m of
      Nothing -> retry
      Just a  -> do
        writeTVar tv Nothing
        return a
```

This is a reasonable implementation of takeMVar in that it has the same blocking behavior as the native implementation: the same set of programs will deadlock with this implementation as would with the native implementation. However, the STM implementation is less useful in two ways:

- *fairness*. The native implementation of MVar holds blocked threads in a FIFO queue, so a thread is never blocked indefinitely as long as the MVar is being repeatedly filled (resp. emptied).
- *single-wakeup*. When there are multiple threads blocked in takeMVar on a single MVar, and the MVar is filled by another thread, then the STM implementation will wake up all the blocked threads. In this case we know that only one of these threads will succeed in its blocked operation and the others will all block again, but the STM implementation in the runtime isn't aware of this property and has to assume that any subset of the blocked transactions can now succeed. The native MVar implementation on the other hand will wake up only one thread, and hence will scale more effectively when there is high contention for an MVar.

In this paper we use MVar primarily to implement the combination of a mutex and a mutable variable. Taking the MVar is equivalent to acquiring the lock and reading the variable, filling the MVar is equivalent to writing the variable and releasing the lock. As with traditional mutexes, when taking multiple MVars we must be careful to take them in a consistent order, otherwise multiple threads trying to take an overlapping set of MVars may deadlock.

## 2.3 IORefs + atomicModifyIORef

This is the lowest level synchronization method available in GHC, and corresponds closely to compare-and-swap-style operations in other languages.

An IORef is a mutable variable, with the following basic operations:

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

corresponding to creation, reading and writing respectively.

One might ask what the *memory model* is for IORefs: for example, are writes made by one thread always observed in order by another thread? Attempts to nail down these issues have lead to significant complexities in other language definitions, for example C++ [2]

and Java [8]. In Haskell, we currently use a model of complete sequential consistency: all writes are observed in the order they are performed. It turns out that this doesn't imply any significant burden on the compiler beyond what the compiler already has to do to ensure *safety*; namely that when a thread calls `readIORef` and then inspects the object returned, the object it reads is actually present. If writes were not ordered in the underlying machine, we might see the write to the `IORef` but not the writes that create the object that it points to, for example. Implementing this safety guarantee implies the use of memory barriers on some types of processor, and typically such memory barriers will also provide sequential consistency for `readIORef` and `writeIORef`.

Still, `readIORef` and `writeIORef` alone are not enough to program most concurrent algorithms, normally we require at least an atomic read-modify-write operation. In Haskell, the operation we provide is[1]

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

The behavior of `atomicModifyIORef` can be understood as being equivalent to the following:

```
atomicModifyIORef r f = do
   a <- readIORef r
   let p = f a
   writeIORef r (fst p)
   return (snd p)
```

with the important exception that the whole operation is performed *atomically* with respect to other threads. The reason it can be performed atomically is Haskell's lazy evaluation: the function `f` is not actually called during `atomicModifyIORef`. Both the contents of the `IORef` and the value returned are *thunks* (also known as suspensions) that will demand the value of `r`, and hence call `f`, when either of their values are demanded.

The implementation of `atomicModifyIORef` in the runtime system looks much like the definition above, except that the final `writeIORef` is replaced by a compare-and-swap: if the current value of the `IORef` is still `a`, then replace it with `fst r`, otherwise loop back to the `readIORef` again. In some ways this is like a mini-transaction over a single variable, and indeed in one of the examples we consider later we do use STM to model `atomicModifyIORef` in order to make a direct comparison between the two.

In contrast to `MVars`, `atomicModifyIORef` can be used to implement *non-blocking algorithms*, in the sense that a thread can always make progress even if many other threads in the system are stalled. STM also has non-blocking properties, but it suffers from other pathologies, such as discrimination against long-running transactions (of course, steps can be taken to mitigate these effects). In the next section, we make use of GHC's synchronization primitives to implement several variants of concurrent singly-linked lists.

## 3. Case Study: Concurrent Singly-Linked List

We consider a standard singly-linked list structure where we use (special) sentinel nodes to represent the head and the tail of the list.

```
data List a
     = Node { val  :: a,
              next :: PTR (List a) }
     | Null

data ListHandle a
     = ListHandle { headList :: PTR (PTR (List a)),
                    tailList :: PTR (PTR (List a)) }
```

[1] the origins of this operation are not clear, but it emerged during a design discussion on the Haskell FFI mailing list in October 2002.

Initial list:

$$\begin{tikzcd}\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4}\end{tikzcd}$$

Concurrent operations: `delete 2 || delete 3`

One possible execution that gives rise to an inconsistent result:

Step 1: `delete 2`

$$\begin{tikzcd}\boxed{1} \arrow[rr, bend left] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4}\end{tikzcd}$$

Step 2: `delete 3`

$$\begin{tikzcd}\boxed{1} \arrow[rr, bend left] & \boxed{2} \arrow[rr, bend right] & \boxed{3} \arrow[r] & \boxed{4}\end{tikzcd}$$

**Figure 1.** Critical sections

The list supports the following functionality:

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

`newList` creates an empty new list. We consider unsorted linked lists. Hence, `addToTail` inserts an element by adding it to the tail, returning a reference to the newly added node. The `find` operation traverses the list starting from the head node and searches for a specified element, until the element is found or the tail of the list is reached. Finally, `delete` removes an element from the list.

Our lists are not ordered, and neither do we provide an operation to insert an element anywhere except at the end. It would be possible to change our algorithms to support ordered lists with insertion, although that would certainly add complexity and would not, we believe, impact our results in any significant way.

In the following, we consider several concurrent implementations where we replace `PTR` by `TVar`, `MVar` and `IORef`.

The challenge is to avoid inconsistencies by protecting critical sections. Figure 1 shows that without any protection (synchronization), concurrent deletion of elements 2 and 3 can lead to a list where element 3 is still present. In turn, we consider several approaches to preventing this from happening.

### 3.1 Lock-free linked list using STM

#### 3.1.1 Straightforward Version

The linked list is of the following form.

```
data List a = Node { val :: a,
                     next :: TVar (List a) }
            | Null
            | Head { next :: TVar (List a) }
```

We use shared pointers, `TVars`, to synchronize the access to shared nodes. To ensure consistency we simply place each operation in an STM transaction. The complete definition of all four functions is given in Table 1. There are no surprises, we can effectively copy the code for a sequential linked list and only need to execute the entire operation inside an `atomically` statement.

We straightforwardly obtain a correct implementation of a concurrent linked list. However, this implementation has severe performance problems as we discuss in detail in Section 4. The main problem is that because each transaction encompasses the entire traversal of the list, any transaction which modifies the list will conflict with any other transaction on the list. This implementation does not have the non-interference properties that we desire, namely that multiple threads should be able to update different parts of the list concurrently.

### 3.1.2 Dissecting Transactions

There are several approaches for dissecting a larger transaction into smaller pieces without compromising consistency (i.e. the linked list operations give us the correct result). One approach is to extend the STM programming model with a way to release parts of the transaction that are no longer required [13]. The solution we choose here is to split up the deletion of nodes into two steps [6].

In case of a delete, we perform a "logical" delete where we only mark the node as deleted. Any operation traversing through a logically deleted node can then "physically" remove this node by swinging the parent's next pointer to the next point of the child. Of course, this operation must be performed atomically. That is, we must "abort" a physical delete if the parent node has changed (for example, has been deleted in the mean time).

For example, given the initial list:

$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4} \end{tikzcd}$$

Concurrent execution of `delete 2 || delete 3` leads to the intermediate result (assuming that `delete 2` takes place first)

Step 1: `delete 2`

$$\begin{tikzcd} \boxed{1} \arrow[rr, bend left] & \boxed{2d} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4} \end{tikzcd}$$

We write lower-case 'd' to indicate that a node has been logically deleted. Execution of the remaining operation `delete 3` yields

$$\begin{tikzcd} \boxed{1} \arrow[rr, bend left] & \boxed{2d} \arrow[r] & \boxed{3d} \arrow[r] & \boxed{4} \end{tikzcd}$$

We manage to logically delete the node but physical deletion fails because the parent has been deleted in the mean time. Physical deletion of node 3 will take place in case of a subsequent traversal of the list.

To implement this idea, we make use of the following refined list structure where we explicitly can mark nodes as (logically) deleted.

```
data List a = Node { val :: a
                   , next :: TVar (List a) }
            | DelNode { next :: TVar (List a) }
            | Null
            | Head    { next :: TVar (List a) }
```

The deletion algorithm is shown in Figure 2. We assume that `x` is the value to be deleted. The go function traverses the list by holding a pointer to the parent (previous) node. Recall that there is always a static head node. Hence, all nodes that will ever be subjected to a delete always have a parent node.

```
go :: TVar (List a) -> IO Bool
go prevPtr = loopSTM $ do
  prevNode <- readTVar prevPtr
  let curPtr = next prevNode
  curNode <- readTVar curPtr

  case curNode of
    Node {val = y, next = nextNode }
      | (x == y) ->
          -- perform logical delete
          do writeTVar curPtr
                (DelNode {next = next curNode})
             return (return True)
      | otherwise ->
          -- continue
          return (go curPtr)

    Null -> return (return False)

    DelNode {next = nextNode } ->
      -- perform physical delete
      case prevNode of
        Node {} -> do
           writeTVar prevPtr
                (Node {val = val prevNode,
                       next = nextNode})
           return (go prevPtr)         -- (***)
        Head {} -> do
           writeTVar prevPtr
                    (Head {next = nextNode})
           return (go prevPtr)
        DelNode {} ->
           return (go curPtr)          -- (+++)

loopSTM :: STM (IO a) -> IO a
loopSTM stm = do
  action <- atomically stm
  action
```

**Figure 2.** STM lazy deletion algorithm

The returned `action` can be viewed as a continuation. For example, in case the current node is (logically) deleted and the parent node is still present (`***`), we swing the parent's next pointer to the current node's next pointer. The STM guarantees that this operation is executed atomically. If successful we "go" to the parent, that is, we stay where we are because the next node we visit is the child of the just physically deleted node. In case the parent itself is deleted (+++), we do nothing and simply move ahead in the list.

A complete implementation is given in Table 2.

While this algorithm increases performance and concurrency significantly with respect to the straightforward STM algorithm, our experimental results (Section 4) indicate that the overhead of STM is still fairly high and therefore we need a number of processor cores to outrun a sequential linked list implementation.

## 3.2 Hand-over-hand locking using `MVars`

Instead of protecting the parent and child node via STM, we can also use fine-grained locking via `MVars`. The standard method is to use hand-over-hand locking (also known as lock coupling) [1] where we acquire the lock for the child before releasing the parent's lock. There is no need to introduce logically deleted nodes because we retain exclusive access to a node by using locks. Table 3 contains the details.

## 3.3 Lock-free linked list using atomic compare-and-swap

To actually challenge the sequential list implementation, we make use of a CAS (atomic compare-and-swap) operations. A CAS only protects a single node which is in fact sufficient to implement the 'logical delete' algorithm discussed earlier.

Here are some code snippets of the CAS implementation.

```
...
 DelNode { next = nextNode } ->
   case prevNode of
     Node {} -> do b <- atomCAS
                          prevPtr   -- address
                          prevNode  -- old value
                          (Node {val = val prevNode,
                                 next = nextNode})
                           -- new value
                   if b -- check if update
                        -- took place
                   then go prevPtr
                   else go curPtr
     Head {} -> do b <- atomCAS
                          prevPtr
                          prevNode
                          (Head {next = nextNode})
                   if b then go prevPtr
                   else go curPtr
     DelNode {} -> go curPtr
...
```

Like in the dissected STM-based implementation, we only perform a physical delete if the parent is still present. Then, we atomically swing the parent's next pointer to the next pointer of the to-be-deleted node. We use a CAS to perform this atomic update operation. In the event that this operation fails (indicated by `b` being `False`), we simply move ahead in the list.

In GHC, there are at least two ways to implement a CAS (atomic compare-and-swap) operation. We can either use the `atomicModifyIORef` primitive

```
atomCAS :: Eq a => IORef a -> a -> a -> IO Bool
atomCAS ptr old new =
   atomicModifyIORef ptr (\ cur -> if cur == old
                                    then (new, True)
                                    else (cur, False))
```

or we can use STM

```
atomCAS :: Eq a => TVar a -> a -> a -> IO Bool
atomCAS ptr old new =
   atomically $ do
       cur <- readTVar ptr
       if cur == old
        then do writeTVar ptr new
                return True
        else return False
```

The complete details of a CAS-based linked list implementation using the first alternative are given in Table 4. In terms of performance, the `atomicModifyIORef` version of CAS is vastly superior to the STM encoding of CAS. Section 4 provides some concrete measurements.

## 3.4 Composability

We should consider which of our algorithms are composable with other concurrent abstractions, since it is likely that clients of the concurrent linked list will be using it in a context in which there are other shared data objects.

The pure STM implementation (Section 3.1.1) is the only algorithm that is completely composable, in the sense that we can use its operations as part of a larger concurrency abstraction with composite invariants.

None of the other algorithms have this property. Furthermore, only the lazy-delete and CAS algorithms (Sections 3.1.2 and 3.3 respectively) are lock-free, in that any thread performing one of these operations cannot prevent another thread from making progress.

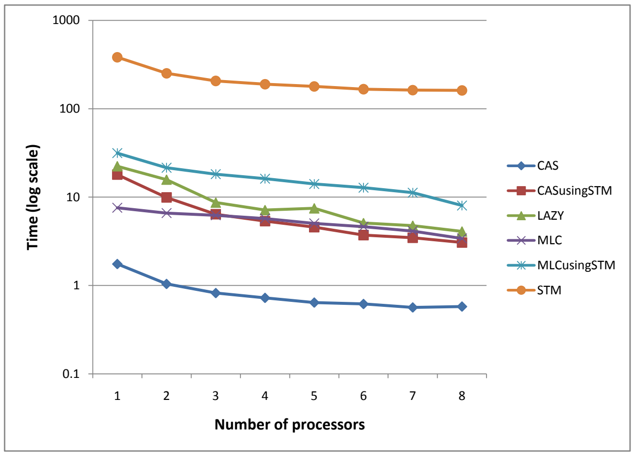

**Figure 3.** Benchmark timings

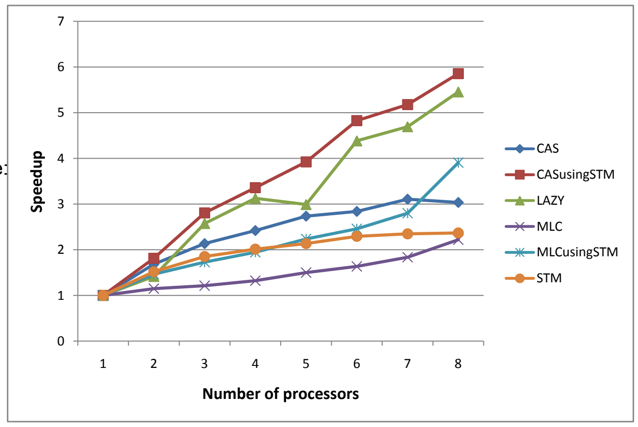

**Figure 4.** Benchmark scaling

## 3.5 Safety

Which of these algorithms are *safe*, in the sense that if a thread is interrupted in the middle of an operation by an asynchronous exception [9], the invariants of the data structure are preserved? In fact the only algorithm that is not safe in this respect is hand-over-hand, since it uses real locks. In order to make it safe we would have to use appropriate exception handlers to restore the locks. This is not difficult—the `MVar` library includes suitable safe abstractions—but it does add a significant performance overhead. Nevertheless, if we were to provide this implementation in a library, asynchronous-exception safety would be an essential property. The version we describe here is therefore useful only to illustrate the relative performance of hand-over-hand locking with `MVar` against the other algorithms.

## 4. Empirical Results

We measured the absolute performance of each of the algorithms on an 8-core multiprocessor (a dual quad-core Intel Xeon 1.86GHz with 16GB of memory). We were using GHC 6.10.1.

To benchmark each algorithm, we randomly generated a test data consisting of an initial list of 3000 elements and 8 lists of 3000 operations in the ratio of 2 finds to 1 delete to 4 inserts[2]. Each benchmark run was measured by loading the test data into memory, generating the initial linked list, and then starting the clock before creating 8 threads, each performing one of the pre-generated lists of operations. When all threads have finished, the clock was stopped and the elapsed time reported. We took the average of 3 runs of each benchmark.

In the figures we use the following abbreviations for each algorithm:

- **STM**. The naive STM implementation (Section 3.1.1).
- **LAZY**. Dissected STM transactions (Section 3.1.2).
- **MLC**. Hand-over-hand locking using MVars (Section 3.2).
- **MLCusingSTM**. Identical to the MLC algorithm, but using the STM implementation of MVars (Section 2.2).
- **CAS** Lock-free compare-and-swap implementation using `IORef` and `atomicModifyIORef` (Section 3.3).
- **CASusingSTM** Identical to the CAS algorithm, but using STM to implement compare-and-swap.

Although GHC includes a parallel garbage collector, our benchmarks were conducted with it disabled for the following reason: we discovered that the linear linked-list structure in the heap apparently does not allow for optimal parallel garbage collection. As such, the parallel collector usually did not improve performance of the programs, and at times even made things worse.

GHC's garbage collector currently performs poorly when there are a large number of `TVar`s in the heap: it visits all the `TVar`s in each GC cycle, rather than just those that have been mutated since the last GC. In contrast, it is more efficient in its treatment of `MVar`s and `IORef`s. While this is a real issue that does affect STM programs, it is also not an inherent limitation of STM, and is likely to be fixed in a future version of GHC. For thease reasons we decided to avoid unfairly biasing the results against STM, and explicitly set a larger heap size (32MB) which reduced the number of GC cycles and the overall GC cost.

We wish to highlight that the programs in these benchmarks revealed some issues in GHC's optimiser, which prevented us from obtaining the best absolute performance for these programs. In particular, the linked-list data structures have an extra layer of boxing: it should be possible to eliminate the `IORef`, `TVar` and `MVar` boxes inside each linked list node. When we tried this GHC did indeed eliminate the boxes, but unfortunately this led to poor code being generated for other parts of the algorithm, and the resulting performance was worse. Fortunately these issues were common across all the benchmarks, so while the absolute performance of these benchmarks is not likely to be optimal, the relative performance between the algorithms is still consistent.

The relative performance of each algorithm is given in Figure 3. Note that the Y axis is a logarithmic scale: there is approximately an order of magnitude difference between STM and the cluster of algorithms in the middle (LAZY, MLC, MLCusingSTM, CASusingSTM), and another order of magnitude faster is the CAS implementation. Figure 4 shows how each algorithm's performance scales with the number of processor cores.

[2] In retrospect, this is probably an unrealistic workload, being too heavily weighted to writing operations. In future work we intend to experiment with different workloads.

### 4.1 Analysis

We expected the naive STM implementation to perform poorly - indeed it scales worse as the length of the list increases, so we could make it look arbitrarily bad by simply running larger tests. This much is not surprising; the problem is well-known. The LAZY (dissected STM) implementation improves on this as expected, and performs more competitively against the hand-over-hand locking implementations, with good scalability results.

We expect hand-over-hand locking to scale poorly, because multiple threads operating on the list cannot overtake each other, since each thread always has at least one list node locked. Similarly, we expect STM to scale poorly, because each transaction that modifies the list will tend to invalidate every other transaction on the list. However, thanks to STM's optimistic concurrency and the fact that not every transaction modifies the list, STM still scales slightly better than hand-over-hand locking.

The absolute performance of `MVar` is better than STM because STM has a certain amount of per-transaction overhead, so on a single processor we see MLC performing better than any of the STM-based implementations. However the poor scalability of MLC results in its slower performance against CASusingSTM at 4 cores and above.

The `MVar` implementation can suffer from poor scheduling decisions when the number of threads exceeds the number of cores. When a thread holding a lock is descheduled, this will block many of the other threads in the system, until the original thread is scheduled again and releases the lock. This is supported by the fact that the performance of both MLC and MLCusingSTM jumps when we reach 8 cores (see Figure 4), when the number of cores equals the number of threads, eliminating this scheduling effect.

The difference between CAS and CASusingSTM (nearly an order of magnitude) is somewhat surprising, but it can be explained by the overhead of list traversal. Traversing list nodes in the CAS algorithm uses only `readIORef`, which is a single memory read and is therefore extremely cheap. In contrast, CASusingSTM must read the pointers between list nodes using `readTVar`, which entails performing a complete transaction, a significantly larger overhead. However, since these transactions are just reading a single `TVar`, the transaction is superfluous: it should never have to retry. In practice however, these transactions *do* sometimes retry. This is because in GHC, validating a transaction consists of attempting to lock each of the `TVar`s involved in the transaction, and if any of these `TVar`s are already locked by another CPU, the transaction is retried. As a result, even transactions that consist of a single `readTVar` can fail and retry. Of course, it would be rather straightforward to optimise GHC's STM implementation to handle this special case by spotting these degenerate transactions at commit-time, but a simpler approach is to provide a new operation

```
readTVarIO :: TVar a -> IO a
```

which just returns the current value of the `TVar`, with no transaction overhead. There is only one minor complication: if the `TVar` is currently locked by another STM transaction in the process of committing, `readTVarIO` has to wait. This can be done with a simple busy-wait loop.

We implemented this operation and measured the effect of using it in CASusingSTM, and it does indeed improve the efficiency of CASusingSTM dramatically, to within 30–40% of CAS (more than a factor of 7 improvement). The remaining difference is presumably due to the overhead of the STM transactions required to perform the real atomic operations.

Interestingly, making this improvement to CASusingSTM also reduced its scaling properties: the improved version only scaled by just over a factor of 2 on 8 processors, whereas the unoptimised CASusingSTM scales by nearly a factor of 6 on 8 processors. This implies that the overhead (expensive list traversals via `readTVar`) we eliminated accounts for a significant portion of the parallelism achieved from running with more processors.

The CAS algorithm also scales by only around a factor of 3 on 8 processors, so we believe that both CAS and the optimised CASusingSTM are experiencing the hardware-imposed overheads due to multiple processors modifying shared data structures.

For completeness, the performance of a purely sequential implementation (using `IORef` with no atomic operations, and no lazy delete) is around 2% better than CAS on a single processor with this hardware. So on two or more processors, CAS easily outperforms the sequential implementation.

## 5. Related Work

There has been considerable amount of prior work on concurrent linked list algorithms and their efficient parallel implementation. State of the art implementations employ non-blocking algorithms which either use (dissected) STM operations or CAS (atomic compare-and-swap) operations. For example, see [3] and the references therein. We have transferred these ideas to the Haskell setting using GHC's synchronization primitives. We have yet to compare measurements/analysis results which we expect will be a challenging task due to differences in the run-time environment and computer architecture used. For example, the algorithms in [3] are implemented in C++ and tested on a Sun Sparc many core architecture.

In the context of Haskell, the only related work we are aware of concentrates exclusively on the performance of STM [13, 10]. The work in [13] extends the Haskell STM interface with an unsafe `unreadTVar` primitive which supports the early release of transactional variables. Such an extension avoids false retries in case of long-running transactions, but obviously requires great care by the programmer. A comprehensive study of the performance of GHC's STM mechanism is given in [10]. One of the benchmarks measures an algorithm equivalent to our LAZY (dissected STM), and measurements show that a significant amount of run-time is spent on STM operations such as validate and commit. As we have shown here, we can drastically improve the performance of an STM-based algorithm by introducing a safe `readTVarIO` primitive (see CASusingSTM).

## 6. Conclusion

We have presented a total of six algorithms for a concurrent singly-linked list: STM, dissected STM, hand-over-hand (using either `MVar` or STM), and compare-and-swap (using either `IORef` or STM).

If we consider performance alone, the compare-and-swap algorithm using `IORef` is superior to all the others by almost an order of magnitude. So ignoring other concerns, highly-tuned concurrent data structures should use this method. However, we have also shown that if a new operation `readTVarIO` is added, it is possible to achieve close to the performance of `IORef` using STM (Section 4.1), and in some cases this is likely to be an attractive solution, since it retains the ability to use transactions for complex manipulations of the data structure, cases where using compare-and-swap is likely to be overly difficult to get right.

If composability is required, then the pure STM algorithm seems the only option, and to improve its performance one possibility is to search for optimisations within the STM framework, either using automatic techniques or help from the programmer [13]. However, recent work [7] suggests that it is possible to recover the benefits of a high-level STM abstraction with the performance of a low-level implementation (e.g. either based on atomic compare-and-swap or using locks). This seems like an attractive option which we plan to pursue in future work.

One clear result is that on its own, hand-over-hand locking is not a good choice. Not only does it scale very poorly, but it is non-compositional and the implementations we presented here do not include the exception-safety that would be necessary if we were to provide it as a library (Section 3.5), and adding this would reduce its performance further.

## Acknowledgements

We thank the reviewers for their comments on a previous version of this paper.

## References

[1] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. pages 129–139, 1988.

[2] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proc. of PLDI'08*, pages 68–78. ACM Press, 2008.

[3] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[4] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[5] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of PPoPP'05*, pages 48–60. ACM Press, 2005.

[6] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *LNCS*, volume 2180, pages 300–314. Springer-Verlag, 2001.

[7] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. of PPoPP '08*, pages 207–216. ACM Press, 2008.

[8] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proc. of POPL'05*, pages 378–391. ACM Press, 2005.

[9] S Marlow, SL Peyton Jones, A Moran, and J Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM Press.

[10] C. Perfumo, N. Sönmez, S. Stipic, O. S. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *Proc. of 5th Conference on Computing Frontiers*, pages 67–78. ACM Press, 2008.

[11] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[12] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, pages 295–308. ACM Press, 1996.

[13] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero. `unreadTVar`: Extending haskell software transactional memory for performance. In *Proc. of Eighth Symposium on Trends in Functional Programming (TFP 2007)*, 2007.

## A. Table of Code Fragments

```
data List a = Node { val :: a,
                     next :: TVar (List a) }
            | Null
            | Head { next :: TVar (List a) }

data ListHandle a
     = ListHandle { headList :: TVar (TVar (List a)),
                    tailList :: TVar (TVar (List a)) }

newList :: IO (ListHandle a)
newList =
 do null <- atomically (newTVar Null)
    hd <- atomically (newTVar (Head {next = null }))
    hdPtr <- atomically (newTVar hd)
    tailPtr <- atomically (newTVar null)
    return (ListHandle {headList = hdPtr,
                        tailList = tailPtr})

addToTail :: ListHandle a -> a -> IO (TVar (List a))
addToTail (ListHandle {tailList = tailPtrPtr}) x = do
  tPtr <- atomically ( do
            null <- newTVar Null
            tailPtr <- readTVar tailPtrPtr
            writeTVar tailPtr
                      (Node {val = x, next = null})
            writeTVar tailPtrPtr null
            return tailPtr
           )
  return tPtr

find ::  Eq a => ListHandle a -> a -> IO Bool
find (ListHandle {headList = ptrPtr})  i =
  atomically (
       do ptr <- readTVar ptrPtr
          Head {next = startptr} <- readTVar ptr
          find2 startptr i)
   where
    find2 :: Eq a => TVar (List a) -> a -> STM Bool
    find2 curNodePtr i = do
      curNode <- readTVar curNodePtr
      case curNode of
        Null -> return False
        Node {val = curval, next = curnext} ->
           if (curval == i) then return True
           else find2 curnext i

delete :: Eq a => ListHandle a -> a -> IO Bool
delete (ListHandle {headList = ptrPtr})  i =
  atomically (
      do startptr <- readTVar ptrPtr
         delete2 startptr i)
   where
    delete2 :: Eq a => TVar (List a) -> a -> STM Bool
    delete2 prevPtr i = do
     prevNode <- readTVar prevPtr
     let curNodePtr = next prevNode
          --head/node have both a next field
     curNode <- readTVar curNodePtr
       case curNode of
         Null -> return False
         Node {val = curval, next = nextNode} ->
           if (curval /= i)
            then delete2 curNodePtr i -- keep searching
            else
              -- delete element (ie delink node)
              do  case prevNode of
                    Head {} -> do
                       writeTVar prevPtr
                              (Head {next = nextNode})
                       return True
                    Node {} -> do
                       writeTVar prevPtr
                              (Node {val = val prevNode,
                                     next = nextNode})
                       return True
```

**Table 1.** STM Linked List Straightforward Version

```
data List a = Node { val :: a
                   , next :: TVar (List a) }
            | DelNode { next :: TVar (List a) }
            | Null
            | Head { next :: TVar (List a) }

find :: Eq a => ListHandle a -> a -> IO Bool
find lh x = searchAndExecute lh x $
               \_ _ -> return (return True)

delete :: Eq a => ListHandle a -> a -> IO Bool
delete lh x = searchAndExecute lh x $
               \curPtr curNode ->  do
                  writeTVar curPtr
                     (DelNode {next = next curNode})
                  return (return True))

searchAndExecute
    :: Eq a
    => ListHandle a
    -> a
    -> (TVar (List a)
        -> List a
        -> STM (IO Bool))
    -> IO Bool

searchAndExecute (ListHandle { headList = head }) x apply =
  do startPtr <- atomically (readTVar head)
     go startPtr
 where
  go prevPtr = loopSTM $ do
      prevNode <- readTVar prevPtr
      -- head/node/delnode all have next
      let curPtr = next prevNode
      curNode <- readTVar curPtr

      case curNode of
        Node {val = y, next = nextNode } ->
          | x == y ->
                -- node found and alive
               apply curPtr curNode
          | otherwise ->
               -- continue
               return (go curPtr)

        Null ->  -- reached end of list
               return (return False)

        DelNode { next = nextNode } ->
            -- delete curNode by setting the next
            -- of prevNode to next of curNode
           case prevNode of
             Node {} -> do
                writeTVar prevPtr
                     (Node {val = val prevNode,
                            next = nextNode})
                return (go prevPtr)
             Head {} -> do
                writeTVar prevPtr (Head {next = nextNode})
                return (go prevPtr)
             DelNode {} ->
                -- if parent deleted simply move ahead
                return (go curPtr)

loopSTM :: STM (IO a) -> IO a
loopSTM stm = do
  action <- atomically stm
  action
```

**Table 2.** Dissected STM Linked List Functions

```
data List a = Node { val :: a
                   , next :: MVar (List a) }
            | Null
            | Head { next :: MVar (List a) }

find :: Eq a => ListHandle a -> a -> IO Bool
find (ListHandle { headList = head }) x =
  let go prevPtr prevNode =
           do let curPtr = next prevNode -- head/node have all next
              curNode <- takeMVar curPtr
              case curNode of
                Node {val = y, next = nextNode } ->
                   if (x == y)
                   then -- node found
                      do putMVar prevPtr prevNode
                         putMVar curPtr curNode
                         return True
                   else
                      do putMVar prevPtr prevNode
                         go curPtr curNode -- continue
                Null -> do putMVar prevPtr prevNode
                           putMVar curPtr curNode
                           return False -- reached end of list
  in do startPtr <- readIORef head
        startNode <- takeMVar startPtr
        go startPtr startNode

delete :: Eq a => ListHandle a -> a -> IO Bool
delete (ListHandle { headList = head }) x =
  let go prevPtr prevNode =
        do do let curPtr = next prevNode -- head/node have all next
              curNode <- takeMVar curPtr
              case curNode of
                Node {val = y, next = nextNode } ->
                   if (x == y)
                   then -- delink node
                     do case prevNode of
                          Node {} -> do putMVar prevPtr (Node {val = val prevNode,
                                                               next = nextNode})
                                        putMVar curPtr curNode
                                        return True
                          Head {} -> do putMVar prevPtr (Head {next = nextNode})
                                        putMVar curPtr curNode
                                        return True
                   else do putMVar prevPtr prevNode
                           go curPtr curNode -- continue
                Null -> do putMVar curPtr curNode
                           putMVar prevPtr prevNode
                           return False -- reached end of list

  in do startPtr <- readIORef head
        startNode <- takeMVar startPtr
        go startPtr startNode
```

**Table 3.** Lock Coupling Based Linked List

```
data List a = Node { val :: a
                   , next :: IORef (List a) }
            | DelNode { next :: IORef (List a) }
            | Null
            | Head { next :: IORef (List a) }

atomCAS :: Eq a => IORef a -> a -> a -> IO Bool
atomCAS ptr old new =
   atomicModifyIORef ptr (\ cur -> if cur == old
                                   then (new, True)
                                   else (cur, False))

find :: Eq a => ListHandle a -> a -> IO Bool
find (ListHandle { headList = head }) x =
  let go prevPtr = do
         prevNode <- readIORef prevPtr
         let curPtr = next prevNode     -- head/node/delnode have all next
         curNode <- readIORef curPtr
          case curNode of
             Node {val = y, next = nextNode } ->
               if (x == y) then return True        -- found
               else go curPtr                      -- continue
             Null -> return False
             DelNode {next = nextNode } ->
                case prevNode of
                   Node {} -> do b <- atomCAS prevPtr prevNode (Node {val = val prevNode,
                                                                      next = nextNode})
                                 if b then go prevPtr
                                  else go curPtr
                   Head {} -> do b <- atomCAS prevPtr prevNode (Head {next = nextNode})
                                 if b then go prevPtr
                                  else go curPtr
                   DelNode {} -> go curPtr            -- if parent deleted simply move ahead
  in do startPtr <- readIORef head
        go startPtr


delete :: Eq a => ListHandle a -> a -> IO Bool
delete (ListHandle { headList = head }) x =
  let go prevPtr = do
         prevNode <- readIORef prevPtr
         let curPtr = next prevNode
         curNode <- readIORef curPtr
         case curNode of
           Node {val = y, next = nextNode } ->
             if (x == y) then
                 do b <- atomCAS curPtr curNode (DelNode {next = nextNode})
                    if b then return True
                     else go prevPtr       -- spin
                   else go curPtr -- continue
           Null -> return False
           DelNode {next = nextNode } ->
                 case prevNode of
                         Node {} -> do b <- atomCAS prevPtr prevNode (Node {val = val prevNode,
                                                                            next = nextNode})
                                       if b then go prevPtr
                                        else go curPtr
                         Head {} -> do b <- atomCAS prevPtr prevNode (Head {next = nextNode})
                                       if b then go prevPtr
                                        else go curPtr
                         DelNode {} -> go curPtr    -- if parent deleted simply move ahead
  in do startPtr <- readIORef head
        go startPtr
```

**Table 4.** CAS Based Linked List