Transactional Tries[*]

Michael Schröder

Vienna University of Technology
mc.schroeder@gmail.com
http://github.com/mcschroeder

Abstract Software Transactional Memory (STM) immensely simplifies concurrent programming by allowing memory operations to be grouped together into atomic blocks. But a common problem with STM is contention. Many standard data structures, when used in a transactional setting, cause unreasonably high numbers of conflicts. I present a contention-free STM data structure for Haskell: the transactional trie. It is based on the lock-free concurrent trie, but lifted into an STM context. It uses well-considered local side-effects to eliminate unnecessary conflicts while preserving transactional safety.

1 Introduction

It is a widely held opinion that concurrent programming is difficult and error-prone. Low-level synchronization mechanisms, such as locks, are notoriously tricky to get right. Deadlocks, livelocks, heisenbugs and other issues encountered when writing complex concurrent systems are usually hard to track down and often confound even experienced programmers.

To simplify concurrent programming, higher-level abstractions are needed. One such abstraction is *Software Transactional Memory* (STM). Briefly, this technique allows the programmer to group multiple memory operations into a single atomic block, not unlike a database transaction. When implemented in a high-level language such as Haskell, with its emphasis on purity and its strong static type system, STM becomes especially powerful.

The fundamental data type of STM is the transactional variable. A TVar stores arbitrary data, to be accessed and modified in a thread-safe manner. For example, I might define a bank account as

```
type Euro = Int
type Account = TVar Euro
```

and then use a function like

```
transfer :: Account → Account → Euro → STM ()
```

to safely — in the transactional sense — move money between accounts.

[*] This report is an abridged version of chapter 3 of my master's thesis [10].

But where do those accounts come from? If I am a bank, how do I represent the whole collection of accounts I manage, in a way that is transactionally safe? The obvious solution, and a common pattern, is to simply use an existing container type and put that type into a TVar:

```
type IBAN = String
type Bank = TVar (Map IBAN Account)
```

Since looking up an account from the Map involves a readTVar operation, the Map is entangled with the transaction, and I can be sure that when transferring money between accounts, both accounts actually exist in the bank at the time when the transaction commits.

The drawback of this pattern of simply wrapping a Map inside a TVar is that when adding or removing elements of the Map, one has to replace the Map inside the TVar wholesale. Thus all concurrently running transactions that have accessed the Map become invalid and will have to restart once they try to commit. Depending on the exact access patterns, this can be a serious cause of contention. For example, one benchmark running on a 16-core machine, with 16 threads each trying to commit a slice out of 200 000 randomly generated transactions, resulted in over 1.3 million retries. That is some serious overhead!

The underlying problem is that the whole Map is made transactional, when we only ever care about the subset of the Map that is relevant to the current transaction. If transaction A updates an element with key k_1 and transaction B deletes an element with key k_2, then those two transactions only conflict if $k_1 = k_2$; if k_1 and k_2 are different, then there is no reason for either of the transactions to wait for the other one. But the Map does not know it is part of a transaction, and the TVar does not know nor care about the structure of its contents. And so the transactional net is cast too wide.

The solution is to not simply put a Map, or any other ready-made container type, into a TVar, but to design data structures specifically tailored to the needs of transactional concurrency. In this report, I present one such data structure: the *transactional trie*.

2 Background: STM in Haskell

Here are the main data types and operations of STM in Haskell:

```
data STM a
instance Monad STM
atomically :: STM a → IO a
data TVar a
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
retry :: STM a
orElse :: STM a → STM a → STM a
```

Atomic blocks in Haskell are represented by the STM monad. Inside this monad, we can freely operate on transactional variables, or TVars. We can read them, write them and create new ones. When we want to actually perform an STM computation and make its effects visible to the rest of the world, we apply atomically to the computation. This function turns an STM block into a transaction in the IO monad that, when executed, will take place atomically with respect to all other transactions.

For example, the following code snippet increments a transactional variable named v:

```
atomically $ do x ← readTVar v
                writeTVar v (x + 1)
```

The use of atomically guarantees that no other thread can come in between the reading and writing of the variable. The sequence of operations happens indivisibly.

An important aspect of Haskell's STM implementation is that it is fully composable. Smaller transactions can be combined into larger transactions without having to know how these smaller transactions are implemented. An important tool to make this possible is the composable blocking operator retry. Conceptually, retry abandons the current transaction and runs it again from the top. In the following example, the variable v is decremented, unless it is zero, in which case the transaction blocks until v is non-zero again:

```
atomically $ do x ← readTVar v
                if x ≡ 0
                  then retry
                  else writeTVar v (x − 1)
```

In addition to retry, there is the orElse combinator, which allows "trying out" transactions in sequence. m1 `orElse` m2 first executes m1; if m1 returns, then orElse returns; but if m1 retries, its effects are discarded and m2 is executed instead.

STM is also robust against exceptions. The standard functions throw and catch act as expected: if an exception occurs inside an atomic block and is not caught, the transaction's effects are discarded and the exception is propagated.

For more background on Haskell's STM, including its implementation, see the original STM papers [3, 2]. For a more thorough exploration of not only STM but also other Haskell concurrency mechanisms, read Simon Marlow's excellent book on that topic [5].

3 Transactional Tries

The transactional trie is based on the concurrent trie of Prokopec, Bagwell, and Odersky [8], which is a non-blocking concurrent version of the hash array mapped trie first described by Bagwell [1].

A hash array mapped trie is a tree whose leaves store key-value bindings and whose nodes are implemented as arrays. Each array has 2^k elements. To look up a key, you take the initial k bits of the key's hash as an index into the root array. If the element at that index is another array node, you continue by using the next k bits of the hash as an index into that second array. If that element is another array, you again use the next k bits of the hash, and so on. Generally speaking, to index into an array node at level l, you use the k bits of the hash beginning at position $k * l$. This procedure is repeated until either a leaf node is found or one of the array nodes does not have an entry at the particular index, in which case the key is not yet present in the trie. The expected depth of the trie is $O(log_{2^k}(n))$, which means operations have a nice worst-case logarithmic performance.

Most of the array nodes would only be sparsely populated. To not waste space, the arrays are actually used in conjunction with a bitmap of length 2^k that encodes which positions in the array are actually filled. If a bit is set in the bitmap, then the (logical) array contains an element at the corresponding index. The actual array only has a size equal to the bit count of the bitmap, and after obtaining a (logical) array index i in the manner described above, it has to be converted to an index into the sparse array via the formula $\#((i - 1) \wedge bmp)$, where $\#$ is a function that counts the number of bits and bmp is the array's bitmap. To ensure that the bitmap can be efficiently represented, k is usually chosen so that 2^k equals the size of the native machine word, e.g. on 64-bit systems $k = 6$.

The *concurrent* trie extends the hash trie by adding *indirection nodes* above every array node. An indirection node simply points to the array node underneath it. Indirection nodes have the property that they stay in the trie even if the nodes above or below them change. When inserting an element into the trie, instead of directly modifying an array node, an updated copy of the array node is created and an atomic compare-and-swap operation on the indirection node is used to switch out the old array node for the new one. If the compare-and-swap operation fails, meaning another thread has already modified the array while we were not looking, the operation is retried from the beginning. This simple scheme, where indirection nodes act as barriers for concurrent modification, ensures that there are no lost updates or race conditions of any kind, while keeping all operations completely lock-free. A more thorough discussion, including proofs of linearizability and lock-freedom, can be found in the paper by Prokopec et al. [8]. A Haskell implementation of the concurrent trie, as a mutable data structure in IO, is also available [9].

The *transactional* trie is an attempt to lift the concurrent trie into an STM context. The idea is to use the lock-freedom of the concurrent trie to make a non-contentious data structure for STM. This is not entirely straightforward, as there is a natural tension between the atomic compare-and-swap operations of the concurrent trie, which are pessimistic and require execution inside the IO monad, and optimistic transactions as implemented by STM. While it is possible to simulate compare-and-swap using TVars and retry, this would entangle the indirection nodes with the rest of the transaction, which is exactly the opposite of what we want. To keep the non-blocking nature of the concurrent trie, the indirection nodes need to be kept independent of the transaction as a whole, which should only hinge on the actual values stored in the trie's leaves. If two transactions were to cross paths at some indirection node, but otherwise concern independent elements of the trie, then neither transaction should have to retry or block. Side-effecting compare-and-swap operations that run within but independently of a transaction are the only way to achieve this. Alas, the type system, with good reason, will not just allow us to mix IO and STM actions, so we have to circumvent it from time to time using unsafeIOToSTM. We will need to justify every single use of unsafeIOToSTM and ensure it does not lead to violations of correctness. Still, bypassing the type system is usually a bad sign, and indeed we will see that correctness can only be preserved at the cost of memory efficiency, at least in an STM implementation without finalizers.

4 Implementation

The version of the transactional trie discussed in this report is available on Hackage at `http://hackage.haskell.org/package/ttrie-0.1.2`. The full source code can also be found at `http://github.com/mcschroeder/ttrie`.

The module Control.Concurrent.STM.Map[1] exports the transactional trie under the following interface:

```
data Map k v
empty :: STM (Map k v)
insert :: (Eq k, Hashable k) ⇒ k → v → Map k v → STM ()
lookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
delete :: (Eq k, Hashable k) ⇒ k → Map k v → STM ()
```

Now let us implement it. As always, we begin with some types:[2]

```
newtype Map k v = Map (INode k v)
type INode k v = IORef (Node k v)
data Node k v = Array !(SparseArray (Branch k v))
              | List  ![Leaf k v]
data Branch k v = I !(INode k v)
                | L !(Leaf k v)
data Leaf k v = Leaf !k !(TVar (Maybe v))
```

The transactional trie largely follows the construction of a concurrent trie:

[1] The name of the trie's public data type is Map, instead of, say, TTrie. The more general name is in keeping with other container libraries and serves to decouple the interface from the specific implementation based on concurrent tries.

[2] The ! operator is a strictness annotation.

– The INode is the indirection node described in the previous section and is simply an IORef, which is a mutable variable in IO. To read and write IORefs atomically, we will use some functions and types from the `atomic-primops` package [6]:

```
data Ticket a
readForCAS :: IORef a → Ticket a
peekTicket :: Ticket a → IO a
casIORef :: IORef a → Ticket a → a → IO (Bool, Ticket a)
```

The idea of the Ticket type is to encapsulate proof that a thread has observed a specific value of an IORef. Due to compiler optimizations, it would not be safe to just use pointer equality to compare values directly.

– A Node is either an Array of Branches or a List of Leafs. The List is used in case of hash collisions. A couple of convenience functions help us manipulate such collision lists:

```
listLookup :: Eq k ⇒ k → [Leaf k v] → Maybe (TVar (Maybe v))
listDelete :: Eq k ⇒ k → [Leaf k v] → [Leaf k v]
```

Their implementations are entirely standard.
The Array is actually a SparseArray, which abstracts away all the bit-fiddling necessary for navigating the bit-mapped arrays underlying a hash array mapped trie. Its interface is largely self-explanatory:

```
data SparseArray a
emptyArray :: SparseArray a
mkSingleton :: Level → Hash → a → SparseArray a
mkPair :: Level → Hash → a → a → Maybe (SparseArray a)
arrayLookup :: Level → Hash → SparseArray a → Maybe a
arrayInsert :: Level → Hash → a → SparseArray a → SparseArray a
arrayUpdate :: Level → Hash → a → SparseArray a → SparseArray a
```

I will not go into the implementation of SparseArray. It is fairly low-level and can be found in the internal Data.SparseArray module of the `ttrie` package. Some additional functions are used to manipulate Hashes and Levels. Again, they are self-explanatory:

```
type Hash = Word
hash :: Hashable a ⇒ a → Hash

type Level = Int
down :: Level → Level
up :: Level → Level
lastLevel :: Level
```

– A Branch either adds another level to the trie by being an INode or it is simply a single Leaf.

The one big difference to a concurrent trie lies in the definition of the Leaf. Basically, a Leaf is a key-value mapping. It stores a key k and a value v. But the way it stores v determines how the trie behaves in a transactional context. Let us build it step by step:

1. Imagine if Leaf were defined exactly like in a concurrent trie:

   data Leaf k v = Leaf !k v

   Then an atomic compare-and-swap on an INode to insert a new Leaf would obviously not be safe during an STM transaction: other transactions could see the new value v before our transaction commits; and they could replace v by inserting a new Leaf for the same key, resulting in our insert being lost.

2. We can eliminate lost inserts by wrapping the value in a TVar:

   data Leaf k v = Leaf !k !(TVar v)

   Now, instead of replacing the whole Leaf to update v, we can use writeTVar to only modify the value part of the Leaf. If two transactions try to update the same Leaf, then STM will detect the conflict and one of the transactions would have to retry.
   Of course, if there does not yet exist a Leaf for a specific key, then a new Leaf will still have to be inserted with a compare-and-swap. In this case it is again possible for other transactions to read the TVar immediately after the swap, even though our transaction has not yet committed and may still abort. This can happen without conflict because the new Leaf contains a newly allocated TVar and allocation effects are allowed to escape transactions by design. Reading a newly allocated TVar will never cause a conflict.

3. To ensure proper isolation, the actual type of Leaf looks like this:

   data Leaf k v = Leaf !k !(TVar (Maybe v))

   By adding the Maybe, we can allocate new TVars with Nothing in them. A transaction can then insert a new Leaf containing Nothing using the compare-and-swap operation. Other threads will still able to read the new TVar immediately after the compare-and-swap, but all they will get is Nothing. The transaction, meanwhile, can simply writeTVar (Just v) to safely insert the actual value into the Leaf's TVar. If another transaction also writes to the TVar and commits before us, then we have a legitimate conflict on the value level, and our transaction will simply retry.

Now that we have the types that make up the trie's internal structure, we can implement its operations. We begin with the function to create an empty trie:

```
empty :: STM (Map k v)
empty = unsafeIOToSTM $ Map <$> newIORef (Array emptyArray)
```

It contains no surprises, although it has the first use of unsafeIOToSTM, which in this case is clearly harmless.

For the rest of the operations, let us assume we have a function

```
getTVar :: (Eq k, Hashable k) ⇒ k → Map k v → STM (TVar (Maybe v))
```

that either returns the TVar stored in the Leaf for a given key, or allocates a new TVar for that key and inserts it appropriately into the trie. The TVar returned by getTVar k m will always either contain Just v, where v is the value associated with the key k in the trie m, or Nothing, if k is not actually present in m. Additionally, getTVar obeys the following invariants:

Invariant 1: getTVar k_1 $m \equiv$ getTVar k_2 $m \iff k_1 \equiv k_2$
Invariant 2: getTVar itself does not read from nor write to any TVars.

Now we can define the trie's operations as follows:

```
insert k v m = do var ← getTVar k m
                  writeTVar var (Just v)
lookup k m = do var ← getTVar k m
                readTVar var
delete k m = do var ← getTVar k m
                writeTVar var Nothing
```

The nice thing about defining the operations this way, is that correctness and non-contentiousness follow directly from the invariants of getTVar. The first invariant ensures correctness. If we get the same TVar every time we call getTVar with the same key, and if that TVar is unique to that key, then STM will take care of the rest. And if, by the second invariant, getTVar does not touch any transactional variables, then the only way one of the operations can cause a conflict is if it actually operates at the same time on the same TVar as another transaction. Unnecessary contention is therefore not possible.

All that is left to do is implementing getTVar. Essentially, getTVar is a combination of the insert and lookup functions of the concurrent trie, just lifted into STM. It tries to look up the TVar associated with a given key, and if that does not exist, allocates and inserts a new TVar for that key. When inserting a new TVar, the structure of the trie has to be changed to accommodate the new element.

Let us look at the code:

```
getTVar k (Map root) = go root 0
  where
    h = hash k
```

The actual work is done by the recursive helper function go. It begins at level 0 by looking into the *root* indirection node. Note that throughout the iterations of go, the hash h of the key is only computed once.

```
go inode level = do
  ticket ← unsafeIOToSTM $ readForCAS inode
  case peekTicket ticket of
    Array a → case arrayLookup level h a of
      Just (I inode₂) → go inode₂ (down level)
      Just (L leaf₂@(Leaf k₂ var))
        | k ≡ k₂       → return var
        | otherwise   → cas inode ticket (growTrie level a (hash k₂) leaf₂)
      Nothing          → cas inode ticket (insertLeaf level a)
    List xs   → case listLookup k xs of
      Just var         → return var
      Nothing          → cas inode ticket (return ∘ List ∘ (:xs))
```

The use of unsafeIOToSTM here is clearly safe — all we are doing is reading the value of the indirection node. This does not have any side effects, so it does not matter if the transaction aborts prematurely. If the transaction retries, the indirection node is just read again — possibly resulting in a different value. It is also possible that the value of the indirection node changes during the runtime of the rest of the function — but that is precisely why we obtain a Ticket.

Depending on the contents of the indirection node, we either go deeper into the trie with a recursive call of go; return the TVar associated with the key; or insert a new TVar by using the cas function to swap out the old contents of the indirection node with an updated version that somehow contains the new TVar.

The cas function is also part of the **where** clause of getTVar:

```
cas inode ticket f = do
  var ← newTVar Nothing
  node ← f (Leaf k var)
  (ok, _) ← unsafeIOToSTM $ casIORef inode ticket node
  if ok then return var
       else go root 0
```

It implements a transactionally safe compare-and-swap procedure:

1. Allocate a new TVar containing Nothing.
2. Use the given function f to produce a *node* containing a Leaf with this TVar.
3. Use casIORef to compare-and-swap the old contents of the *inode* with the new *node*.
4. If the compare-and-swap was successful, the new *node* is immediately visible to all other threads. Return the TVar to the caller, who is now free to use writeTVar to fill in the final value.
5. If the compare-and-swap failed, because some other thread has changed the *inode* since the time we first read it, restart the operation — not with the STM retry, which would restart the whole transaction, but simply by calling go *root* 0 again.

All that is remaining now are the functions given for f in the code of go. Given a new Leaf, they are supposed to return a Node that somehow contains this new Leaf. In the case of the overflow list, this is just a trivial anonymous function that prepends the leaf into the List node. The insertLeaf function does pretty much the same, except for Array nodes:

```
insertLeaf level a leaf = do
  let a' = arrayInsert level h (L leaf) a
  return (Array a')
```

In case of a key collision, things are a slightly more involved. The growTrie function puts the colliding leaves into a new level of the trie, where they hopefully will not collide anymore:

```
growTrie level a h_2 leaf_2 leaf_1 = do
  inode_2 ← unsafeIOToSTM $ combineLeaves (down level) h leaf_1 h_2 leaf_2
  let a' = arrayUpdate level h (I inode_2) a
  return (Array a')
combineLeaves level h_1 leaf_1 h_2 leaf_2
   | level ⩾ lastLevel = newIORef (List [leaf_1, leaf_2])
   | otherwise =
     case mkPair level h (L leaf_1) h_2 (L leaf_2) of
       Just pair → newIORef (Array pair)
       Nothing → do
         inode ← combineLeaves (down level) h_1 leaf_1 h_2 leaf_2
         let a = mkSingleton level h (I inode)
         newIORef (Array a)
```

The use of casIORef here is once again harmless, as combineLeaves only uses IO to allocate new IORefs. The mkPair function for making a two-element SparseArray returns a Maybe, because it is possible that on a given level of the trie the two keys hash to the same array index and so the leaves cannot both be put into a single array. In that case, another new indirection node has to be introduced into the trie and the procedure repeated. If at some point the last level has been reached, the leaves just go into an overflow List node.

5 Memory efficiency

While the transactional trie successfully carries over the lock-freedom of the concurrent trie and keeps the asymptotic performance of its operations, it does have to make a couple of concessions regarding memory efficiency.

The first concession is that when looking up any key for the first time, the lookup operation will actually grow the trie. This is a direct consequence of using getTVar to implement the trie's basic operations. If getTVar does not find the Leaf for a given key, it allocates a new one and inserts it. One might wonder if it is possible to implement a lookup function that does not rely on getTVar. The following attempt is pretty straightforward and appears to be correct at first glance — although you might already guess from its name that something is not quite right:

```
phantomLookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
phantomLookup k (Map root) = go root 0
  where
    h = hash k

    go inode level = do
      node ← unsafeIOToSTM $ readIORef inode
      case node of
        Array a → case arrayLookup level h a of
          Just (I inode₂) → go inode₂ (down level)
          Just (L (Leaf k₂ var))
            | k ≡ k₂      → readTVar var
            | otherwise   → return Nothing
          Nothing         → return Nothing
        List xs  → case listLookup k xs of
          Just var        → readTVar var
          Nothing         → return Nothing
```

The problem with this simple implementation is that under certain circumstances it allows for *phantom reads*. Consider the following pair of functions:

```
f = atomically $ do v1 ← phantomLookup k
                    v2 ← phantomLookup k
                    return (v1 ≡ v2)

g = atomically (insert k 23)
```

Due to STM's isolation guarantees, one would reasonably expect that f always returns True. However, sometimes f will return False when g is run between the two phantomLookups in f. How is this possible? If you start out with an empty trie, then the first phantomLookup in f obviously returns Nothing. And it does so without touching any TVars, because there is no TVar for k at this point. Only when running g for the first time, will a TVar for k be created. The transaction inside f will now happily read from this TVar during the second phantomLookup and will not detect any inconsistencies, because this is the first time it has seen the TVar. This problem does not only occur on an empty trie, but any time we look up a key that has not previously been inserted. The only remedy is to ensure that there is always a TVar for every key, even if it is filled with Nothing, which is exactly what the implementation of lookup using getTVar does.

Granted, it seems as if these kinds of phantom lookups might not occur regularly in practice, and even if they did, they would probably cause no great harm. The overhead of always allocating a Leaf for every key that is ever looked up, on the other hand, seems much more troublesome. However, phantomLookup exhibits exactly the kind of seldom-occurring unexpected behavior that results in bugs that are incredibly hard to find. And having a lookup function that grows the trie is really only an issue in two cases:

1. when we expect the keys we look up to not be present a significant amount of the time; then a transactional trie is probably really not the right data structure. Although if one were to use phantomLookup instead of lookup, and if in this particular scenario phantom lookups are actually acceptable, then using a transactional trie could still be feasible.
2. when a malicious actor purposefully wants to increase memory consumption, i.e. a classic denial-of-service attack; then one can again counteract this by using phantomLookup, limited to those places that are susceptible to attack. For example, a login routine in a web application could first use phantomLookup to check if the user actually exists, before continuing with the transaction. Here the phantom lookup does not matter, because if the user does not exist the transaction is aborted anyway.

Thus, it makes sense to have the behavior of lookup be the default and provide phantomLookup for those select scenarios where it is actually an improvement.

The other trade-off the trie has to make regarding memory efficiency, is that the delete operation does not actually remove Leafs or compact the trie again. It merely fills a Leaf's TVar with Nothing. This frees up the values associated with the keys, which is the major part of a trie's memory consumption, but it does not delete the keys or compress the structure that has emerged in the trie, which might now be suboptimal given the trie's current utilization.

Again, for the common use case, this might not be a problem. Very often, we do not want to actually delete certain data, but merely mark it as deleted; or maybe delete the data, but mark the associated keys as having been previously in use in order to prevent reusing them. Think of unique user IDs, for example. In such a scenario, the overhead of the trie not actually deleting Leafs disappears.[3]

6 Evaluation

I empirically evaluated the transactional trie against similar data structures, measuring contention, runtime performance and memory allocation. The benchmarks were run on an Amazon EC2 C3 extra-large instance with Intel Xeon E5-2680 v2 (Ivy Bridge) processors and a total of 16 physical cores. Under comparison were three hashing-based container types: a transactional trie; a HashMap from the `unordered-containers` library [11], wrapped inside a TVar; and the STM-specialized hash array mapped trie from the `stm-containers` library [13].

The same random Text strings are used as keys for each container. Each benchmark consists of a number of random STM transaction. The transactions

[3] For the cases where we do want the trie to always be as compact a representation of its data as possible, there is an unsafeDelete operation, which really does remove Leafs and compresses the trie again. Alas, as its name suggests, unsafeDelete is not transactionally safe. It can be made safe by using an STM extension called *finalizers*. For a thorough description of finalizers and how they can be used to make unsafeDelete safe, see [10].

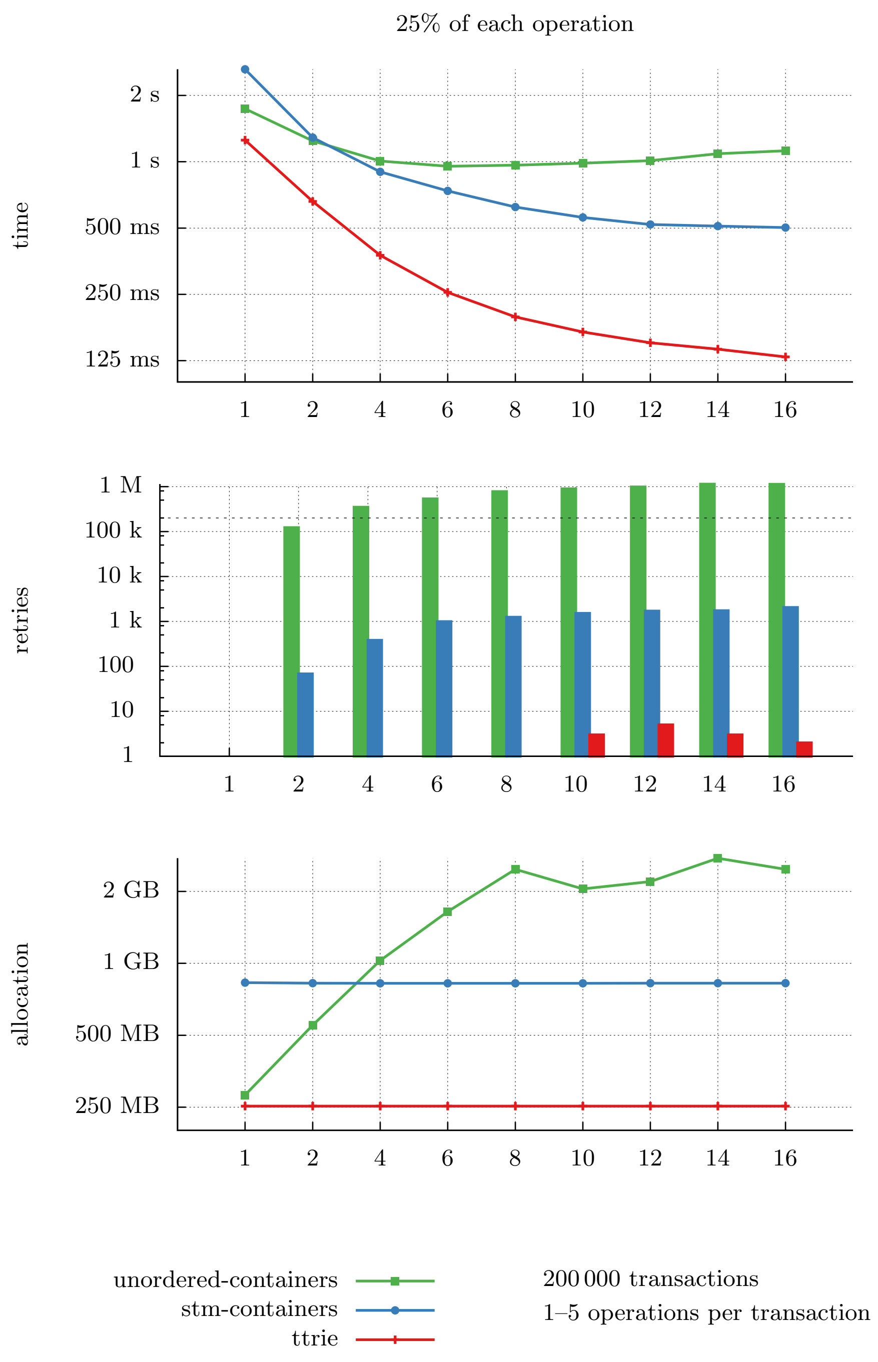

Figure 1. Benchmark comparing STM data structures

are split evenly over the number of threads in use. The time it takes to complete all transactions for a particular container is measured using the `criterion` and `criterion-plus` libraries [7, 12], which calculate the mean execution time over many iterations. To measure contention, the transactions are run again using the `stm-stats` library [4] to count how often the STM runtime system has to restart transactions due to conflicts. Finally, the transactions are run once more to measure the total amount of allocated memory, using GHCs built-in facilities for collecting memory usage statistics. The benchmark was compiled using GHC 7.8.3. For more details about test data generation and the exact benchmark setup, see the `ttrie` source distribution.

Figure 1 shows the results of a benchmark performing 200 000 random STM transactions, where each transaction performs 1–5 operations. Each operation (insert, lookup, update or delete) occurs equally likely in the mix of operations per transaction and the containers are prefilled with 1 000 000 entries.[4]

As we can see from the number of retries, the transactional trie exhibits no contention; the handful of retries it has to perform — 5 in the worst case — are due to legitimate conflicts. This is in stark contrast to `unordered-containers` and `stm-containers`: here, the number of spurious retries vastly overshadows the legitimate conflicts. In the worst case, the TVar-wrapped HashMap has to retry more than 1.3 million times for the 200 000 transactions to succeed. The run-time performance of `unordered-containers` begins to rapidly degrade at 4 threads, which is when the number of retries first exceeds the number of transactions.

Overall, `ttrie` is 2–4 times faster than `stm-containers`, allocating only a third of the memory; and 1.3–8.6 times faster than `unordered-containers`, allocating almost 10 times less memory.

References

[1] Phil Bagwell. *Ideal Hash Trees.* Tech. rep. LAMP-REPORT-2001-001. EPFL, 2001.

[2] Tim Harris and Simon Peyton Jones. "Transactional memory with data invariants." In: *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.* TRANSACT '06. Ottawa, Canada: ACM, 2006.

[3] Tim Harris et al. "Composable Memory Transactions." In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP '05. Chicago, IL, USA: ACM, 2005, pp. 48–60.

[4] David Leuschner, Stefan Wehr, and Joachim Breitner. *`stm-stats`: retry statistics for STM transactions.* Version 0.2.0.0. 2011. URL: `http://hackage.haskell.org/package/stm-stats-0.2.0.0`.

[5] Simon Marlow. *Parallel and Concurrent Programming in Haskell.* O'Reilly Media Inc., 2013.

[4] For additional benchmarks with different setups, see [10].

[6] Ryan Newton. *`atomic-primops`: A safe approach to CAS and other atomic ops in Haskell.* Version 0.6. 2014. URL: `http://hackage.haskell.org/package/atomic-primops-0.6`.

[7] Bryan O'Sullivan. *`criterion`: Robust, reliable performance measurement and analysis.* Version 1.0.2.0. 2014. URL: `http://hackage.haskell.org/package/criterion-1.0.2.0`.

[8] Aleksander Prokopec, Phil Bagwell, and Martin Odersky. *Cache-Aware Lock-Free Concurrent Hash Tries.* Tech. rep. EPFL-REPORT-166908. EPFL, 2011.

[9] Michael Schröder. *`ctrie`: Non-blocking concurrent map.* Version 0.1.0.2. 2014. URL: `http://hackage.haskell.org/package/ctrie-0.1.0.2`.

[10] Michael Schröder. "Durability and Contention in Software Transactional Memory." MSc Thesis. Vienna University of Technology, 2015. URL: `http://github.com/mcschroeder/thesis`.

[11] Johan Tibell and Edward Z. Yang. *`unordered-containers`: Efficient hashing-based container types.* Version 0.2.5.0. 2014. URL: `http://hackage.haskell.org/package/unordered-containers-0.2.5.0`.

[12] Nikita Volkov. *`criterion-plus`: Enhancement of the `criterion` benchmarking library.* Version 0.1.3. 2014. URL: `http://hackage.haskell.org/package/criterion-plus-0.1.3`.

[13] Nikita Volkov. *`stm-containers`: Containers for STM.* Version 0.2.3. 2014. URL: `http://hackage.haskell.org/package/stm-containers-0.2.3`.