Improving STM Performance with Transactional Structs*

Ryan Yates
Computer Science Department
University of Rochester
Rochester, NY, USA
ryates@cs.rochester.edu

Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY, USA
scott@cs.rochester.edu

ABSTRACT

Software transactional memory (STM) has been an important and useful feature for Haskell. Its performance, however, is limited by several inefficiencies. While safe concurrent computations are easy to express in Haskell's STM, concurrent data structures suffer unfortunate bloat in the implementation due to an extra level of indirection for mutable references as well as the inability to express unboxed mutable transactional values. We address these deficiencies by introducing `TStruct` to the GHC run-time system, allowing strict unboxed transactional values as well as mutable references without an extra indirection. Using `TStruct` we implement several data structures, discuss their design, and provide benchmark results on a large multicore machine. Our benchmarks show that some concurrent data structures built with `TStruct` significantly out-perform and out-scale their `TVar`-based equivalents.

1. INTRODUCTION

The Haskell programming language, as implemented by the Glasgow Haskell Compiler (GHC), has many innovative features, including a rich run-time system to manage the unique needs of a pure functional language with lazy evaluation. Since its introduction by Harris et al. in 2005 [5], GHC's STM has grown increasingly popular. Most uses are not performance critical, but rather focus on ensuring correctness in the face of concurrency from user interaction or system events. Transactional memory (TM) based concurrent data structures are less common and little effort has been invested in the sort of performance tuning that has characterized STM work for imperative languages [6, chap. 4].

In comparison to most of those imperative implementations, GHC's TM is unusual in its use of explicit transactional variables called `TVars`. Inspecting or manipulating these variables outside of the context of a transaction is not allowed. There is no special compiler support for STM beyond the existing type system. STM is supported instead by the run-time system. Inside transactions, execution is restricted to operations on `TVars` and the usual pure functional computations. `TVar` operations consist of creation (with an initial value), reading, and writing.

In our work we expand from `TVars` to `TStructs`, allowing users to express transactional computations on structures with word and reference fields. This change can significantly reduce the memory overhead of TM data structures, speed up execution, and in some cases reduce contention by decreasing the number of synchronization operations. At this time we are still missing compiler and language support to make programming with `TStruct` as easy as programming with `TVars`, but we expect support to be possible and this work shows that the performance improvements to be gained make the effort worthwhile.

In this paper we

1. describe extensions to GHC's fine-grain locking STM to support transactional structures containing a mix of words and pointers while maintaining the features (`retry` and `orElse`) and properties of the STM (no global bottlenecks and read-only transactions take no locks).

2. implement several data structures with both `TStruct` and `TVar` to explore where performance improves or degrades.

3. provide results from data structure microbenchmarks on a large multicore machine.

Section 2 provides background information on GHC's existing fine-grain locking STM implementation as well an overview of the `stm` library's interface for writing transactions. In Section 3 we describe deficiencies in the existing implementation, and introduce the `TStruct` interface and implementation as a means of addressing these deficiencies. In Section 5 we present four concurrent data structures built using `TStruct`, and characterize the behavior of their methods. We describe our benchmarking techniques and present performance results in Section 6. We finish with related work, future work, and conclusions in Sections 7 and 8.

2. BACKGROUND

2.1 STM Interface

GHC Haskell's STM API is given in Figure 1. The `STM` type is implemented the same as the `IO` type but with only the `TVar`-manipulating actions. The `atomically` function takes an `STM` action and gives an `IO` action which, when executed, will perform the transaction atomically, so that other threads see either all or none of the transaction's effects. New variables are made with `newTVar`, taking an initial value for the variable. The `IO` variant is useful for creating global variables.

*This work was funded in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Studies.

```
instance Monad STM where ...

data TVar a = ...
instance Eq (TVar a) where ...

newTVarIO :: a → IO (TVar a)
newTVar   :: a → STM (TVar a)

readTVar  :: TVar a → STM a
writeTVar :: TVar a → a → STM ()

atomically :: STM a → IO a

retry  :: STM a
orElse :: STM a → STM a → STM a
```

Figure 1: API for STM with `TVar`s.

```
struct TVar {
  Header       header
  Word         version
  WatchQueue*  queue
  HeapObject*  value
}

struct TRecEntry {
  TVar*        tvar
  HeapObject*  old_value
  HeapObject*  new_value
  Word         version
}
```

Figure 2: STM implementation for `TVar`s.

The `retry` action and `orElse` combinator allow for blocking transactions and composing alternatives. When `retry` is executed the transaction is blocked and waits for one of the `TVar`s that it has read to change values before attempting again to execute. The `orElse` combinator tries the first action and either commits it or, if it executes `retry`, undoes all its effects and executes the second action instead. Note that `orElse` atomically chooses between the actions, so for the result to be the result of the second action it must hold for the duration of the whole transaction that the first action results in `retry`.

2.2 STM Implementation

The run-time system supports STM at several levels. We focus on the run-time library level (written in C) and the garbage collection support. Transactions proceed in three phases: execution, validation, and commit. Execution must track both read and write accesses to `TVar`s, recording writes and providing the new values in subsequent reads. Validation ensures that the transaction observed a consistent view of memory. Commit acquires locks for `TVar`s being updated and performs the updates. If there are any failures in these steps, the transaction will discard its work and start again from the beginning.

When a transaction starts, a transactional record (`TRec`) is created. The `TRec` maintains a chunked linked list of entries recording each `TVar` read, the value seen when encountered, and any new value written to the `TVar`. Executing `readTVar` will first search the `TRec` for a matching entry and use any new value as the value read. If there is no entry for the `TVar` then a new entry is created and the value is read directly from the `TVar`. The `value` field in each `TVar` can double as a lock variable: the locking thread stores a pointer to its transactional record instead of the actual value. When adding a new `TRec` entry we must first check to see if the pointer already refers to a `TRec`. If so, we spin until the value changes. We will see later that locks are never held for unbounded time, so deadlock is not possible. Performing a `writeTVar` is similar to reading: we start by searching for an entry or adding a new one; then we record the value being written in the `TRec` entry.

After a transaction finishes execution, it is validated by comparing `TRec` entry values with the values in the `TVar`s. Details about validation and commit are given in Section 4.

3. ADDING TRANSACTIONAL STRUCTS

In this section we discuss some problems with `TVar`s and how `TStruct` gives the expressiveness to overcome many of these problems. We also give details of our implementation and the various parts of GHC that were modified.

3.1 Indirection with TVars

Consider a red-black tree. A node of such a tree will typically consist of pointers to children, a parent pointer, fields for key and value, and a field for the color of the node. The simplest insertion will search to find the insertion location, make a new node, and link it to its parent. Linking mutates the pointer field of the parent node. When a rebalance is needed, however, several pointers will be reassigned as well as color fields. We could choose to keep the color fields as pure values and make new nodes whenever the color changes, but this can be difficult to manage as each new node must be relinked. Making the color mutable by storing the color value in a `TVar` adds significant memory overhead and indirection. Each `TVar` must point to a heap object, not an unboxed value. To store the color, we have a pointer to a `TVar` in the node object and a pointer in the `TVar` to a boxed value, a significant amount of overhead for one bit of information.

3.2 Mutable Unboxed Values

We avoid some of the indirection problems with `TVar`s by introducing a new built-in transactional structure we call `TStruct`. Every `TStruct` can be allocated with a fixed number of word sized fields and pointer fields, each of which can be written and read transactionally. We can then store fields like key, value, and color as words in the structure and pointers to other nodes as pointer fields. Perhaps more important than the space saving is avoiding indirection. By keeping the words and pointers close together we are likely to need to touch fewer cache lines than if we must follow pointers to get values.

Unfortunately, pointer fields of a `TStruct` often still entail a level of indirection to accommodate sum types like `Node`, which may be either true nodes or `Nil`. For the true node case, the implementation is simply an indirection word that points to the `TStruct` for that node. GHC is expected soon to be able to unpack objects like our `TStruct` into sum types, and we hope to leverage that ability to avoid the final indirection. This may aid in `TVar` based data structures as well, but will not, in that case, serve to reduce or coalesce STM metadata.

```
data TStruct a = ...

instance Eq (TStruct a) where ...

newTStructIO :: Int → Int → a → IO (TStruct a)
newTStruct   :: Int → Int → a → STM (TStruct a)

readTStruct  :: TStruct a → Int → STM a
writeTStruct :: TStruct a → Int → a → STM ()

readTStructWord  :: TStruct a → Int → STM Word
writeTStructWord :: TStruct a → Int → Word
                 → STM ()

lengthTStruct      :: TStruct a → Int
lengthTStructWords :: TStruct a → Int
```

Figure 3: API for `TStruct`.

3.3 Implementation Details

3.3.1 Haskell API

Our implementation of `TStruct` is based on GHC's small array support, specifically the `SmallMutableArray#` type. Each `TStruct` has three parts: metadata, words, and pointers. The metadata includes size fields that indicate the number of word and pointer fields, together with STM metadata that mirrors that of a `TVar`: a lock word, a lock counter, and a version number. The size of a `TStruct` never changes and for many uses could easily be known at compile time. Future work will explore exploiting this for better performance. For now we make use of "unsafe" read and write operations to avoid bounds checks when appropriate. Garbage collection of `TStruct` objects simply follows the pointer fields as it would in a `SmallMutableArray#`.

A simple API for working with `TStruct` is given in Figure 3. The `newTStruct` actions create a new struct with parameters for number of words and number of pointers and an initializing value for the pointers. Note that we are limited to one type of pointer. Nothing in the implementation requires this restriction and we use this simple API along with `unsafeCoerce` to build a richer API specific to particular data structures. Transactional reading and writing work similarly to `TVar` but with an index. Out of range indices will raise an exception. Lengths in `TStruct`s are immutable so we have pure functions that give the number of pointers and words.

In addition, for some data structures, we make data structure specific initialization actions that are non-transactional. When `TVar`s are created there is only one field to initialize and this initialization is done *non-transactionally*. That is, the write is not delayed until commit, but is immediately set in the `TVar` (since that `TVar` is not yet visible to any other thread). With `TStruct` there are several fields that may need initialization. In future work we would like to explore an API that gives static guarantees that these non-transactional accesses happen only on private structures.

3.3.2 Run-time System Details

To support `TStruct`, the existing STM runtime is augmented with a separate list of `TRec` entries for tracking `TStruct` accesses. The `TStruct` entries contain an additional field to indicate the accessed index within the `TStruct`. The offset can be compared with the number of pointer fields to determine if the access is a word access or a pointer access (this is essential for garbage collection to correctly handle TRecs). Details about the commit are in Section 4.

4. STM CORRECTNESS WITH TSTRUCT

Haskell's STM has its roots in the OSTM of Fraser and Harris [3, 4]; `TStruct` builds on this implementation. In this section we will show that our `TStruct` implementation (and the original GHC STM implementation) are strictly serializable, meaning that for any concurrent STM execution there exists some total order on transactions that is consistent with "real time" (if transaction T_1 finishes before transaction T_2 in the implementation, then T_1 precedes T_2 in the total order), and that would have produced the same results if executed sequentially.

4.1 Commit Overview

We can think of transactions as executing in two phases. The first phase executes the code of the transaction and builds the transactional record (`TRec`). Interaction with shared memory in this phase consists only of the initial reads of `TVar`s and `TStruct`s; subsequent reads are satisfied from the `TRec`. At the end of this phase the `TRec` captures the data that were read and the changes to shared state that would need to occur to make the transaction "happen."

The second phase is the commit, which effects the changes in the `TRec`, but only if the state of shared memory matches the view recorded in the `TRec`. That is, commit should only happen if it would be consistent with stopping the world and performing the execution while directly mutating shared memory. We must show that the view of shared memory at commit is the same as the view during execution, that this view is a consistent view of memory, and other transactions will be prevented from committing conflicting changes concurrently.

The commit phase has three steps, *validate*, *read check*, and *update*. If either validate or read check fails, the transaction will release its locks and restart without having made any changes. If the update step is reached, the transaction will always make its changes and then release its locks. This implies that two conflicting transactions cannot both reach their update step (a conflict is where two transactions use the same memory location and at least one writes to it).

4.2 TVar Commit

Psuedocode for the existing GHC STM implementation appears in Figure 4. As in OSTM, the value word in a `TVar` is also the lock variable. As `TVar`s only hold references to heap objects a locked state can be indicated by referencing a special heap object that cannot be referenced by user code. In a simplification of OSTM, however, the GHC code does not share access to `TRec`s among threads: OSTM leverages shared access to ensure that when conflicting transactions are committing at the same time one of them succeeds (thus ensuring lock-free progress). GHC's STM admits the possibility of livelock: when a `TRec` pointer is read from a `TVar`, the current thread releases any locks it holds and retries. This read barrier is safe because, as we shall see later, locks are held only for a finite number of steps (assuming OS threads are scheduled with some fairness).

In addition to the value/lock field, `TVar`s contain a version field which is incremented with every update. The valida-

tion step does three things: acquire locks for `TVar`s in the write set, check that the view of memory seen during execution is still the state of memory, and record version numbers for all the `TVar`s in the read set. The read check step ensures that the view of memory seen during both execution and validation is in fact consistent, and further that no other committing transactions conflict. Finally the update increments version numbers, writes new values, and releases locks.

4.3 The need for Read Check

Consider the timeline given in Figure 5. Transaction T_0 reads two `TVar`s, x and y, initially both zero. Between its reads of x and y transaction T_1 fully executes and commits, updating both x and y to one and giving T_0 an inconsistent view of memory with $x = 0$ and $y = 1$. If T_0's commit were to continue without any other transactions doing work this inconsistency would be discovered in validate where the expected value, zero, stored in the `TRec` for x, would not match the value, one, in the `TVar`. Other transactions, however, can commit while T_0 is validating, leading to validation seeing the *same inconsistent view* of memory as execution. The read check detects this by checking the version numbers stored in validate and the values again. For the value check in read check to succeed x must be set back to zero (by T_4 in the diagram), but this cannot happen without the version number also increasing (subscript 4 in the read of x in read check).

We might be tempted to store the version numbers during execution. This, however, would remove an important benefit of value-based validation. Consider a program in which multiple threads are handling events that arrive in a series of queues ordered by priority, and an execution in which a thread T sees all the highest priority queues empty and starts handling a low-priority event. While T is handling its event, new high-priority events arrive and are quickly handled by other worker threads. When T starts validation all the high-priority queues are again empty, but their version numbers have all been incremented. As a result, T will be unable to commit, even though logically it should be able to. By storing version numbers in validate the window for conflicting commits is narrowed considerably.

4.4 TStruct Commit

With `TVar`s the lock and the value were conflated. We cannot use the same technique with `TStruct`s as they have word values in addition to references. Instead we can conflate the lock and the version number. Odd values indicate that the `TStruct` is locked, with the high order bits identifying the thread that holds the lock. Even values indicate that the `TStruct` is unlocked, with the high order bits specifying a version number. Psuedocode for `TStruct` commit appears in Figure 4. An additional field is included in `TStruct`s for a lock count, as multiple fields in a `TStruct` may be written in a transaction. Validation must now handle there being multiple entries, reads and writes, that are protected by a single lock and a single version number. Reads will check the value, but in contrast to the `TVar` case this will not simultaneously check the lock status. A separate check for the lock is needed. The `TStruct` could be locked by this transaction, however. If it is locked by this transaction we must not treat the lock field as a version number! When the lock is first acquired, the version number is stored for later use in unlocking (by writing that version number incremented by two). If the committing transaction does not hold the lock, the entry is a read, the old value matches, and the lock is not held, then we record the version number for the `TStruct`. In the read check we again check all the entries in the read set for matching values and either a lock held by this transaction or a matching version number (the version number cannot change if we hold the lock). When updating we write the new value first and then unlock with the next version number.

4.5 Correctness

Consider a single `TVar` location x and the ways in which validation, read check, and updates to that location can interleave. Updates first acquire the lock in validate, then increment the version and unlock by writing the new value in update. Because updates are guarded by locking and unlocking the location we know that all updates are ordered. We can understand the interaction of these updates with a transaction T that is committing and only reads x by considering two parallel timelines, one for T and one for the ordered updates to x. Transaction T will value check x, read the version, then value check x again in validate. Later in the read check it will value check x then check that the version matches. For the span between T's reads of x's version we know immediately that an increment to x's version will be detected. This leaves only a few possible combinations that may happen. An update could start between the value check and the version read in validate, locking x and incrementing its version immediately before T's version read. With x locked, however, it will fail the value check before the second version read, so the lock release must happen before this value check. The value check will then fail because the value will be different (if it isn't, there is no conflict). The only remaining possibility is a second update to x that puts the old value back. The value, however, will not be updated until after the version is incremented. Therefore if we successfully reach the end of the read check for x we know that no updates have happened to x between the version reads and that x has the same value seen in execution.

Given that each read-only location of T has a span where no updates have happened, and the entire read set is visited in validate before it is visited again in the read check, there exists some point in the intersection of those spans where all the values hold and all the locks for the write locations are held. We do not know that this point exists until the end of the read check. Even if values have changed at that point we know that the updates could only have happened in a transaction that did not access any of T's written locations, because those locations were locked. The updates can then be ordered after T. Any updates that would lead to an inconsistent view of memory are ruled out because they would require some update to the version before T's read check finishes.

For `TStruct` we have equivalent properties but several details change, because it is the lock and the version that are conflated instead of the lock and the value. Updates write values first, then versions, because the version is written at lock release. The version is still the value that is increasing with every update and updates are still ordered due to the lock acquire and release. The span between version reads to a read set location x will detect any overlap with an update. Successfully reaching the end of the read check then

```
commit(TRec* trec) {
  validate(trec)
  read_check(trec)
  update(trec)
}

bool value_check(entry* e) {
  return  e→tvar→value == e→old_value
}


validate(TRec* trec) {
  for (e in trec) {
    if (is_write(e)) {
      abort_if(!try_lock(e)
            || !value_check(e))
    } else {
      abort_if(!value_check(e))
      e→version = e→tvar→version
      abort_if(!value_check(e))
    }
  }
}



read_check(TRec* trec) {
  for (e in read_set(trec)) {
    abort_if(!value_check(e)
          || e→tvar→version ≠ e→version)
  }
}



update(TRec* trec)
{
  for (e in write_set(trec)) {
    e→tvar→version++
    e→tvar→value = e→new_value
  }
}
```

```
commit(TRec* trec) {
  validate(trec)
  read_check(trec)
  update(trec)
}

bool value_check(entry* e) {
  return e→tstruct→payload[e→index]
           == e→old_value
}

validate(TRec* trec) {
  for (e in trec) {
    if (is_write(e)) {
      abort_if(!try_lock(e) || !value_check(e))
    } else {
      abort_if(!value_check(e))
      version_lock = e→tstruct→lock
      if (version_lock ≠ this_lock) {
        abort_if(is_locked(version_lock)
                  || !value_check(e))
        e→version = version_lock
      }
    }
  }
}

read_check(TRec* trec) {
  for (e in read_set(trec)) {
    version_lock = e→tstruct→lock
    abort_if((version_lock ≠ e→version
                && version_lock ≠ this_lock)
          || !value_check(e))
  }
}

update(TRec* trec)
{
  for (e in write_set(trec)) {
    e→tstruct→payload[e→index] = e→new_value
    e→tstruct→lock = e→tstruct→version+2
  }
}
```

Figure 4: Commit psuedocode for `TVar`s on the left and `TStruct`s on the right.

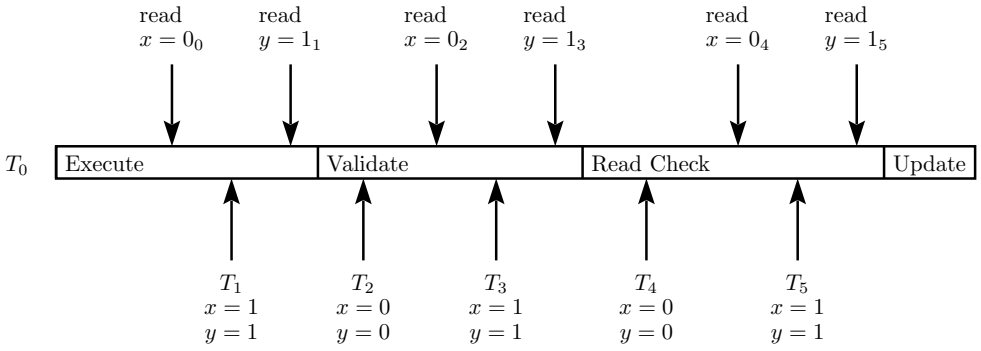


Figure 5: Timeline illustrating a sequence of commits that could lead to committing with an inconsistent view of memory. This diagram is from the perspective of T_0's execution and commit. Reads of shared memory (with values subscripted with the corresponding version number) are noted above the line while successful commits from other transactions and their updates are below the line. A consistent view of memory will always have both x and y with the same value.

has the same implication as with `TVar`s: the view of memory matches the state seen in execution and no conflicting updates happen until after a point where the locks are held and simultaneously the values matched. Updates to multiple locations in a `TStruct` result in the lock being held longer—not multiple acquires or releases—so this does not introduce any additional complexity for correctness.

5. DATA STRUCTURES WITH TSTRUCT

In this section we discuss our `TStruct`-based implementations of four data structures: red-black trees, skip lists, cuckoo hash, and hashed array mapped tries.

5.1 Red-Black Tree

In Section 3.1 we discussed our red-black tree node for both `TVar` and `TStruct` versions. Of the data structures we implemented, the red-black tree was the simplest in terms of the node design. Each node only points to nodes of the same type and layout and every node is the same size. It may, however, have the most complex code due to handling rebalancing. It presents an interesting case as it is a simple data structure while still providing good worst case performance not relying on probabilistic outcomes. Despite it simplicity, it is complex enough to drive extensive research in the context of concurrent data structures including with transactional memory [2, 3, 9].

The transactional memory approach gives great flexibility by allowing easy expression of concurrent data structures that are very difficult to otherwise express while approaching the fairness properties of a tailored solution. Indeed, transactional memory allows an arbitrary composition of individual operations into atomic operations, a much more difficult task then the, until recently elusive, wait-free algorithm. Recent work by Natarajan, Savoie, and Mittal has derived a wait-free algorithm for concurrent red-black trees [10] giving an interesting point of comparison for TM.

For transactional memory the difficulty comes in performance, some of which may only come with data structure specific improvements. Transactional structs addresses an important common part of many data structures, the node representation. By allowing the expression of whole nodes with both references and values we decrease the memory overhead and increase the locality of the data.

5.2 Skip List

Skip lists have been a fruitful target for concurrency research [7, 15]. Unlike red-black trees, skip lists do not rebalance and instead rely on randomization to achieve performance comparable to that of a balanced tree with high probability. This strategy greatly simplifies the structure and keeps operations mostly localized—much as in a concurrent linked list.

The idea with a skip list is to maintain a hierarchy of ordered linked lists with the lowest layer containing all the nodes and each layer above containing a subset of the previous list. Searching the list can start at the top level which has the fewest nodes. If the key is not found it can move down a level and start searching from where it left off in the previous list. In this way it can "skip" over large portions of the list refining its way down to the key. Pugh has shown [15] that if the probability of a node appearing in the next layer up is some fixed $p < 1$ then the expected cost of search will be $O(\log n)$, with the particular value of p balancing search costs with storage space.

A skip list implementation requires a source of pseudorandom numbers. While we could keep the state for a random number generator in a transactional variable, we want to avoid the overhead of transactional accesses. We also want to avoid any contention on the state of the number generator, so we keep a separate state for each thread and ensure that each is on a separate cache line. Non-transactional mutable state is excluded from transactions by simply restricting the actions available with the STM type to manipulating `TVar`s, `retry`, `orElse`, and the normal `Monad` operations. One cannot, for instance, read from an `IORef` in a transaction. If non-transactional state were included, the current implementation could deadlock if that state was used to make the decision to execute `retry`. This is because `retry` is implemented by waiting for a change to one the variables in the transaction's read set. The transaction will wake up only when a change is made to one of those variables. We use `unsafeIOToSTM` to perform non-transactional access and take care not to leak information from the state of the random number generator. Note that non-determinism is already common in transactions, since the schedule of transaction execution may determine program outcome.

A skip list node is implemented as a single `TStruct` with the key and value in word slots and levels of pointers in pointer slots. The number of words is fixed in this use of `TStruct` while the number of pointers varies from node to node. Skip list nodes are slightly more complicated then the red-black tree nodes due to the varying number of levels in each node. The code is much simpler, however, with the only difficult aspect being the source of random numbers.

5.3 Cuckoo Hash Table

The Cuckoo hash table [12] is an open addressing hash structure in which a pair of hash functions is used to give two locations for a particular key. On insertion if both locations are full, one of the existing entries will be evicted to make room for the new entry. The evicted item will then go to its alternate location, possibly leading to further evictions. If the chain of evictions is too long, the table is resized. Our implementation follows the concurrent Cuckoo hash table described by Herlihy and Shavit [8], with a pair of tables, one for each hash function.

In a concurrent setting the Cuckoo hash table is appealing because lookups need to look in only two locations and deletions need only to additionally change one location. Insertions look for a free bucket among the two locations and often will be done with a small change at that location: updating the size and writing the value into the bucket.

Our `TVar`-based implementation is structured as an array of mutable `TVar`s that reference immutable buckets. When an insertion or deletion happen, a new bucket is made, copying appropriate entries. In the `TStruct`-based implementation, we have an immutable array of pointers to mutable `TStruct` buckets. Insertions and deletions simply update a few entries in the bucket. The `TStruct` buckets are of fixed size, containing a size field, keys as words, and values as pointers.

5.4 Hashed Array Mapped Trie

The Hashed Array Mapped Trie (HAMT) data structure [1] is commonly used in Haskell in its pure functional form as the underlying structure in the `unordered-containers`

package from Johan Tibell for the `Map` and `Set` abstractions [17]. An HAMT avoids several of the usual performance problems of tree-like data structures. In a "Trie" the bits of the key are broken into fixed size chunks of n bits each. Each chunk is an index for a level of the tree. The corresponding node at that level can be indexed by the chunk to find the next node for that key. Nodes can either be levels or leaves where the levels point to further nodes and leaves contain key-value pairs. As an example consider the key $42 = 101010_2$ in a trie with $n = 3$ bits per level. Each node will have $2^3 = 8$ entries with the root indexed by the first three bits 010 and (if needed) the next level indexed by 101, the third and sixth entries in the nodes respectively.

The "Hashed" part of the HAMT name indicates that the key is hashed before indexing to ensure a uniform distribution, avoiding (with high probability) the need for rebalancing. Given a uniform distribution of hash values, levels should become more and more sparse as one moves down the tree, leading to the desire for a more compact representation. The "Array Mapped" part of the HAMT name indicates a technique that does just that, by storing a population bitmap with 2^n bits and as many pointers to lower levels as there are bits set in the bitmap. A trie level node in our example above with two children would have to have six wasted entries, where the "Array Mapped" scheme would need only 8 bits to indicate the dead ends. A trade-off with array mapping is that adding or removing an entry will require an entirely new node. In the context of immutable data structures this is expected. There is a lot of opportunity for intermediate designs that allocate nodes with a bit of extra space for anticipated growth. In a similar vein, mutation can be used for removal by marking dead ends rather then removing nodes. We leave the exploration of these designs to future work.

5.4.1 The Population Count Instruction

Array mapping can benefit greatly from the `popcnt` or "population count" hardware instruction supported in most current architectures. The instruction counts the number of bits set to 1 in a word. This allows quick indexing into the compact array by first masking the bitmap to only contain set bits that precede the desired index, then counting the number of those bits set. For example, if we want to look at the entry in the array at index 4 and our bitmap is 01110101_2 we will want to look at index 2 in the compact array. We find that by first masking with $2^4 - 1 = 00001111_2$ yielding 00000101_2 then counting set bits with `popcnt` giving 2. This 2 is the number of exiting entries in the array *before* our entry, telling us how many slots in the compact array to skip before our entry.

5.4.2 Implementation with TVars

We use an existing Haskell implementation of HAMT from Nikita Volkov found in the `stm-containers` package [18] with the minor change of ensuring that insertions of duplicate keys leave the existing value rather than replacing it (thus allowing the transaction to remain read-only). The layout of the data structure and corresponding code is given in Figure 6a. Each `Node` is a sum type with a `Nodes` constructor for levels and two leaf constructors, one for single entries and the other for entries with hash collisions. Mutation in this structure happens only at the `TVar` referenced in the `Nodes` constructor. The bitmap for array indexing is given in the `Indices` typed field in `WordArray`.

In the `TStruct` based implementation, nodes are either a `WordArray` level or `SizedArray` leaf similar to the code in Figure 6b. Mutation happens in the array part when, for instance, a child is replaced by an expanded node on insert and the parent reference is updated to the new child. To remove unneeded indirection in this structure we implement the whole node as a `TStruct` with an explicit tag field as seen in Figure 6b.

5.4.3 HAMT Comparison

The HAMT falls somewhere in between the red-black tree and skip list in complexity. Most of the difficult aspects of HAMT lie in the data representation. Here `TStruct` makes things somewhat simpler although (in the absence of compiler support) with significantly less safety. HAMT nodes come in several forms and sizes.

6. PERFORMANCE EVALUATION

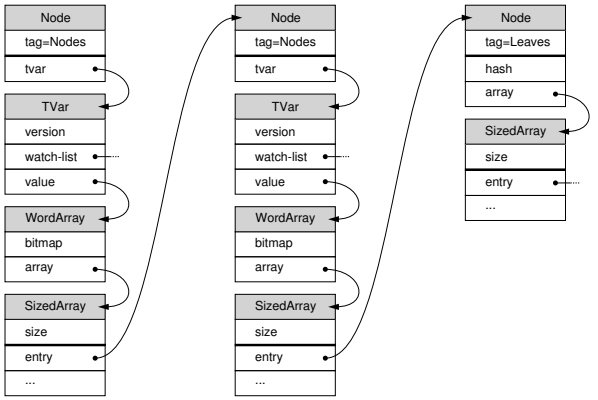
Results were obtained on a 2-socket, 36-core, 72-thread Intel Xeon E5-2699 v3 system. To achieve consistent results we augmented GHC's thread pinning mechanism to allow assignment of Haskell execution contexts to specific cores, and experimented with different assignments as we increased the number of threads used in our experiments. The best performance was achieved by first filling all the cores on one chip then moving to the second chip and finally using the SMT threads.

6.1 Data Structure Throughput

Our benchmarking work has focused on data structure steady state throughput performance. Figure 7d shows the throughput of a mix of operations on a data structure representing a set which initially has 50,000 entries. When the benchmark runs, each thread performs a transaction which will search for a random key (from a key space of 100,000) 90% of the time, insert a random key 5% of the time, and delete a random key the remaining 5% of the time. Due to the mix of operations the structure is expected to keep its size regardless of the length of the run. Given this and the size of the key space, we can expect half of insertions and deletions to follow a read-only (with respect to the Haskell transaction) path where the entries already exist in the structure in the case of insertion and the where the entries do not exist in the case of deletion.

For comparison in the HAMT case we include performance with a concurrent implementation (`ctrie`) that uses compare-and-swap operations on `IORef`s. Here the `TStruct` implementation outperforms the `TVar` implementation significantly, with 4.2 times the throughput of `TVar` on a single thread and 6.4 times the throughput of `TVar` on 36 threads. Using SMT thread does not benefit total throughput for `TStruct`. The skip list and cuckoo hash table implementations also show benefits for `TStruct`, though smaller in magnitude.

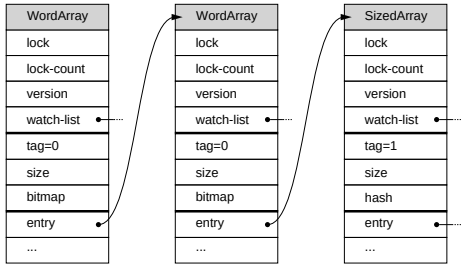
Our red-black tree does not perform as well with `TStruct`. Several factors may be causing this including false conflicts introduced by `TStruct`s or the benefits being limited by the small constant size of nodes and outweighed by the increased overhead. One significant difference between where `TStruct` works and does not is that, due to the design, HAMT nodes are only ever mutated in one field in each transaction. In future work we hope to gain a more detailed understanding




```
data Node a = Nodes (TVar (WordArray a))
            | Leaf Hash a
            | Leaves Hash (SizedArray a)

data WordArray  a = WordArray
Bitmap (Array (Node a))
data SizedArray a = SizedArray Size    (Array a)
```

(a) The `TVar`-based node data type.




```
data Node a = WordArray  Size Bitmap (Array (Node a))
            | SizedArray Size Hash   (Array a)
```

(b) The `TStruct`-based node data type.

Figure 6: The `TVar`-based (a) and `TStruct`-based (b) node data types and diagrams showing two level nodes, and a leaf node of an HAMT.

of the reasons behind the performance of our various data structures.

7. FUTURE WORK

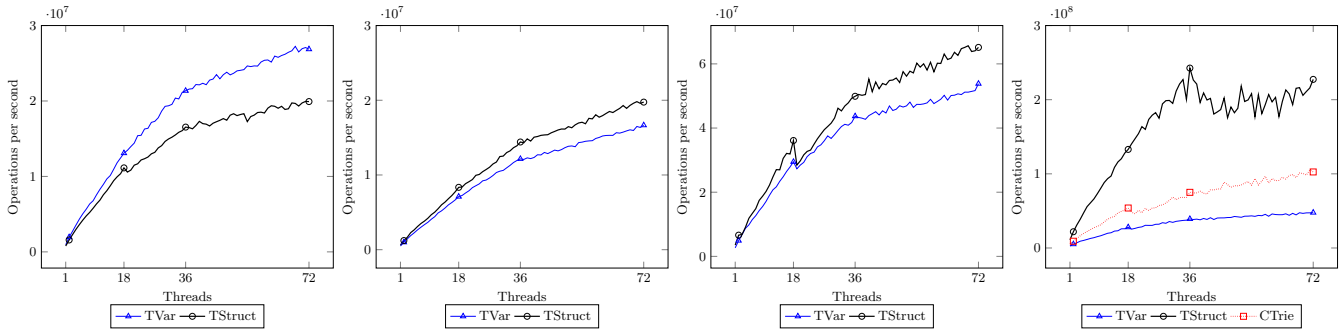
While we are seeing significant performance improvements for some applications with `TStruct`, we are not satisfied with the code that must be written to achieve this. Compiler support for expressing transactional structs could improve the quality of generated code and provide better safety and simplicity to programmers. As mentioned in Section 3.3.1, non-transactional initialization of `TStruct`s can be guaranteed safe in common scenarios. In future work we will explore an API that exposes these accesses safely. Specifically we hope to build on a recent proposal by Simon Marlow to add data types with mutable fields [**?**]. In this proposal data constructor with mutable fields are `IO`, `ST`, or `STM` actions while pattern matching on a constructor introduces references to mutable fields rather then values. These references are simply the pairing of an offset with a pointer to the constructor heap object. Additional actions allow access for reading and writing to fields within the proper context. Simple extensions to GADT syntax give a clean way to express these data types.

We also hope to explore more data structures that can benefit from transactional structs. This will likely lead to exploring improved transactional array support as well. There are several variations to the HAMT data structure that we hope to explore, given that we have more freedom to perform mutation in the context of STM and `TStruct`s. For instance, we may be able to avoid allocating new nodes and copying when an item is deleted by instead marking the entry as deleted with a sentinel value or a deletion bitmap. We could also explore over-allocating some levels of the HAMT, trading compact nodes for the expectation that nodes high in the tree will later become saturated. Of course these may lead to poor performance due to increased conflicts.

Another direction we have begun to explore is `TStruct` alignment. By aligning all allocations and GC copy operations of `TStruct` heap objects we can avoid false conflicts that increase inter-core communication and degrade performance. This trades off some space to internal fragmentation, but may improve performance for some concurrent workloads.

Our original motivation for `TStruct` was to improve performance of a hybrid transactional memory implementation where transactions are first attempted using hardware transactional memory. Along the way we discovered that `TStruct` improved performance of software-only transactional memory greatly on some data structures. In future work we hope to find ways to use hardware transactions to yield additional performance improvements, and to understand the factors that lead to good and poor performance of Haskell code in hardware transactions.

A concern that we have with our work is how well it translates to performance improvements in real-world applications. Few existing applications make significant use of STM data structures, even though STM is widely used for synchronization—`retry`-based condition synchronization in



(a) Red-black tree. (b) Skip list. (c) Cuckoo Hash Table. (d) HAMT.

Figure 7: Operations on data structures with roughly 50,000 entries where 90% of the operations are lookups and the rest are split between insertions and deletions (note the differing scales on the y-axis).

particular. It is unclear if STM data structures are avoided simply due to their poor performance. Applications will typically use a pure functional data structure and gain mutation by referring to the whole structure from a single mutable cell. Threads then access this cell with appropriate synchronization (usually `atomicModifyIORef`) to update the reference to a new data structure. This pattern works well on low core counts, but fails to scale as the single cell inevitably becomes a bottleneck [11].

8. CONCLUSION

We have shown that we can extend GHC's fine-grain locking STM to support transactional structures. Given this support we have explored the implementation of several data structures and their performance on microbenchmarks on a large multicore machine. On the hashed array mapped trie in particular this leads to substantial performance improvements.

9. REFERENCES

[1] P. Bagwell. Ideal hash trees. Technical report, 2001. http://lampwww.epfl.ch/papers/idealhashtrees.pdf.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, Bangalore, India, Jan. 2010.

[3] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[4] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. on Computer Systems (TOCS)*, 25(2):5, 2007.

[5] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, Chicago, IL, June 2005.

[6] T. L. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, San Francisco, CA, second edition, 2010.

[7] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[9] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[10] A. Natarajan, L. H. Savoie, and N. Mittal. *Concurrent Wait-Free Red Black Trees*, pages 45–60. Springer International Publishing, Cham, 2013.

[11] R. R. Newton, P. P. Fogg, and A. Varamesh. Adaptive lock-free maps: Purely-functional to scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 218–229, New York, NY, USA, 2015. ACM.

[12] R. Pagh and F. F. Rodler. *Cuckoo Hashing*, pages 121–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[13] A. Prokopec, P. Bagwell, and M. Odersky. Cache-aware lock-free concurrent hash tries. Technical report, 2011. https://infoscience.epfl.ch/record/166908.

[14] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*, volume 47, pages 151–160. ACM, 2012.

[15] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[16] M. Schröder. ctrie: Non-blocking concurrent map, 2013. https://hackage.haskell.org/package/ctrie.

[17] J. Tibell. unordered-containers: Efficient hashing-based container types, 2012. https://hackage.haskell.org/package/unordered-containers.

[18] N. Volkov. stm-containers: Containers for stm, 2016. https://hackage.haskell.org/package/stm-containers.