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.

CCS Concepts
• Computing methodologies → Concurrent programming languages; • Software and its engineering → Concurrent programming languages; Concurrent programming structures;

Keywords
Haskell; transactions; hashed array mapped trie

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 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.

Our contributions are the following:

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 and properties of the STM.
2. Implement several data structures with both TStruct and TVar versions 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 as an overview of the stm library’s interface for writing transactions. In Section 3 we detail our implementation and interface for transactional structures and include descriptions of deficiencies in the existing implementation that motivate the features. Our concurrent data structures are described along
with discussion of some general properties of the structures relating to concurrency in Section 4. Performance evaluation and description of our benchmark techniques is in Section 5. We finish with related work, future work, and conclusions in Sections 6, 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 where other threads either see all or none of the effects from the transaction. New variables are made with newTVar, taking an initial value for the variable. The IO variant is useful for creating global variables.

The retry and orElse features allow for blocking transactions and composing alternatives. When retry is executed the transaction is blocked and waits for one of the TVars that it has read to change values before attempting again to execute. The orElse function will try the first action, if it executes retry all the effects of that action will be ignored and the second action will be executed. 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 but we will focus just on the run-time library level written in C and the garbage collection support. Execution of transactions proceeds in three phases: running, validation, and commit. Running must track both read and write accesses to TVars, 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 TVars being updated and performs the updates. If there are any failures in these steps, the transaction will simply 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. Writing 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 running, it is validated by comparing TRec entry values with the values in the TVars (see Figure 3). If any differences are seen, the transaction is aborted. We also record the version number from the TVar in the entry. If the entry has a new value from our transaction we acquire the lock using a compare-and-swap (CAS) atomic operation. If we successfully get through all the entries we are then at a point where all the locks are held and we have version numbers for all the read entries. We then go through all the read entries and check that the value still matches and that the stashed version number matches the version in the TVar. This is to prevent committing after observing a partial update from another transaction. Once this read check is done we can perform our updates by incrementing the version in each TVar with a written entry and unlocking it by writing the new value into the value field.

3. ADDING TRANSACTIONAL STRUCTS

In this section we discuss some problems with TVars 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, then make a new node and link it to its parent. This 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 when node 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 `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 `TVars` 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 that 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 in 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 with `TStruct` our pointers often still have a level of indirection as we have our pointers point to a Haskell sum data type `Node` which distinguishes between nodes and `Nil`. This means that our `Node` constructor is an indirection that simply 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 take advantage of that ability to avoid this indirection. This may aid in `TVar` based data structures as well, but will not reduce or coalesce STM metadata.

### 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` with a single 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 4. 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 entries 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. With TStruct there are several fields that may need initialization. We only need to be concerned about transactional access when other threads have access to the TStruct which must happen after the transaction that creates the structure is committed. 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 keeping an offset that indexes into the TStruct indicating the field of the array that was accessed. The offset can be compared with the number of pointer fields to determine if the access is a word access or pointer access. This is essential for garbage collection to correctly handle TRecs.

There are several differences in how TStruct must commit. With TVar the lock field is also the value field. Since TVar always point to heap objects, we can lock a TVar and know who owns it by making it point to a TRec. When another transaction tries to read from the TVar it must check first to see if it read a TRec. If so, it will spin, waiting see the unlocking update. With TStruct we have one lock for each object, not one lock for each field. We also need to support unboxed values making conflation of lock and field more difficult. To support these needs, we have a separate lock field and a lock counter for transactions that mutate more then one field. In addition we conflate the lock and version to aid in getting atomic reads of the lock state and version. We do this by using the least-significant bit to indicate the lock status. It is also convenient to use the rest of the bits to indicate lock ownership when the lock is held meaning that odd values in the lock field indicate that the TStruct is locked and tell what transaction holds the lock while even values give a version number.

As we go through the TRec at commit time we acquire locks for each entry that is a write and set the lock counter to zero and stash the version number that we saw in an additional field in the TStruct. If we already hold the lock, we increment the lock counter. When we hold the lock we know that no other transaction can write to this TStruct. If we cannot successfully lock the TStruct, or if the expected value in the record does not match the value in the field, we abort the transaction and restart it from the beginning. As we acquire locks for write entries we read version numbers for read entries and store them in the TRec entry. Once we have all of the locks we perform a read check to ensure that we have not observed a partial commit. The read check compares version numbers and values, and also checks that the TStruct remains unlocked. If the version numbers and values hold we have seen a consistent view of memory while we acquired locks and it matches the view of memory from execution. The transaction is now free to perform its updates.

Like the existing TVar commit, our TStruct commit protocol has no global bottlenecks and avoids the need for read-only transactions to acquire any locks. Reading from a TStruct, however, does not require checking the value to see if it is locked as we must with TVar.

4. 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.

4.1 Red–Black Tree

In Section 3.1 we discussed our red–black tree node for both TVar and TStruct versions. The red–black tree data structure has been studied extensively in the context of concurrent data structures including with transactional memory [2, 9]. Results have been mixed, however, due to the complexity of handling concurrency in a structure that includes rebalancing. Balanced trees provide good worst case performance and do not rely on probabilistic outcomes for good expected performance. Some work on concurrent balanced trees has focused on relaxing strict balancing and instead allowing some form of eventual balance. Depending on the application this might sacrifice the very property that makes a balanced tree critical. Recent work by Natarajan, Savoie, and Mittal has derived a wait-free algorithm for concurrent red–black trees [10]. 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 arbitrary an composition of individual operations into atomic operations, a much more difficult task then the, until recently elusive, wait-free algorithm. 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.

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.

4.2 Skip List

Skip lists have been a fruitful target for concurrency research [15, 7]. Unlike red–black trees, skip lists do not rebalance and instead rely on a probabilistic scheme for good expected performance similar to a balanced tree. This 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 random 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 random state so we keep a separate state for each thread and ensure that each state 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 TVars, retry, orElse, and the normal Monad operations. One cannot, for instance, read from an IORef in a transaction. If these non-transactional effects were included the current implementation could deadlock if that state were 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 thus 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. Search for a key starts at the root node at the top level and alternates moving across nodes following the links at the current level and down when the key is not found. If a key greater then the search key or the end node are encountered then the search moves down and continues at that lower level. Search complete when the key is encountered or we attempt to move down passed the bottom level. Inserting a new node starts with a search for the insertion location keeping a stack where the current node is pushed ever time the search moves down. Once the insertion location is found a new node is made with height given by a random number from the skip list distribution. The new node takes links from previous nodes given by the stack and writes their values into its level, then links the previous node to the new node. Deletion also starts with search and then begins unlinking with the previous node in the search. Unlinking continues down the previous node moving forward to a new previous node for each level down as necessary.

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.

### 4.3 Cuckoo Hash Table

The Cuckoo hash table [12] is an open addressing hash structure where 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 given in [8] with a pair of tables, one for each hash function.

In a concurrent setting the Cuckoo hash table is appealing because lookups only need to look in two locations and deletions only need 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 TVars 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 fixed size keeping a size field, keys as words, and values as pointers.

### 4.4 Hashed Array Mapped Trie

In this section we discuss the Hashed Array Mapped Trie data structure. This structure 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 data structures [17].

#### 4.4.1 The HAMT Data Structure

A Hashed Array Mapped Trie (HAMT) [1] 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 in a trie with $n=3$ bits per level. Each node will have $2^n = 8$ 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 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 only need to pay $2^n$-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.

#### 4.4.2 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 011101012.
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)

Figure 5: The TVar-based node data type and a diagram showing two level nodes and a leaf node of an HAMT.

we will want to look at index 2 in the compact array. We find that by first masking with $2^4 - 1 = 00001111$ yielding 00000101 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.

4.4.3 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 leaves 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 5. Each Node is a sum type with a Nodes constructor for levels and two leaf constructors, one for single entries and the other for 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 6. 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 6.

4.4.4 HAMT Comparison

Compared to the red–black tree and skip list implementations the HAMT falls somewhere in between in complexity. Most of the difficult aspects of implementing 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.

5. 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.

5.1 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 we include performance on our benchmark...
with a concurrent HAMT implementation (cterie) that uses compare-and-swap operations on IORefs. 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.

Our red–black tree does not perform as well with TStruct. Several factors may be causing this including false conflicts introduced by TStructs or the benefits being limited by the small constant size of nodes and outweighed by the increased overhead. The skip list and cuckoo hash table implementations improves slightly. 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 of the reasons behind the performance of our various data structures.

6. RELATED WORK

GHC’s original STM implementation is based on Fraser’s OSTM [3, 4] extended to support orElse and retry [5]. As mentioned in Sections 4.1 and 4.2, extensive work has explored concurrent red–black trees and skip lists. Concurrent versions of HAMT have also been studied [13, 14]. Interestingly their implementation leverages an extra indirection in the TVar version we worked to remove in order to gain low cost snapshots. A Haskell implementation given in the ctrie package [16] performs better then the TVar version, but worse then our TStruct implementation. With improvements to GHC’s low-level atomic operations it should be possible to make a version of ctrie that outperforms an STM HAMT.

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 TStructs can be guaranteed safe in common scenarios. In future work we will explore an API that exposes these accesses safely.

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 TStructs. 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 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 as shown in [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 data structure this leads to substantial performance improvements.

9. REFERENCES


