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 made it significantly easier to write correct concurrent programs in 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 runtime 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 concurrent data structures built with `TStruct` out-scale and out-perform their `TVar`-based equivalents.

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

Keywords Haskell; transactions

ACM Reference Format:
Ryan Yates and Michael L. Scott. 2017. Improving STM Performance with Transactional Structs. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7–8, 2017 (Haskell'17),* 11 pages.
https://doi.org/10.1145/3122955.3122972

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. [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].

Haskell'17, September 7–8, 2017, Oxford, UK

ACM ISBN 978-1-4503-5182-9/17/09…$15.00
https://doi.org/10.1145/3122955.3122972

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 working on compiler and language support to make programming with `TStruct` as easy as programming with `TVars`, but we expect support to be possible (see preview in Appendix A), and the performance results reported here confirm the effort to be 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; no locks for read-only transactions).
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 4 we explain how our extensions ensure program correctness (strict serializability). 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 discussion of future work in Section 7.

2 Background

While most of the details in Harris et al. [5] remain true of GHC's STM implementation, some details are missing that are relevant to our work. In this section we give an overview of the existing implementation (with some simplification and abstraction to make the description more clear).

```
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 TVars.

```
struct TVar {
  Header       header
  Word         version
  WatchList*   watchList
  HeapObject*  value
}
struct TStruct {
  Header       header
  Word         lock
  Word         lockCount
  Word         version
  WatchList*   watchList
  Word         words
  Word         ptrs
  union { Word word, HeapObject* ptr } payload[]
}
```

Figure 2. STM implementation for TVars and TStructs.

2.1 STM Interface

GHC Haskell's STM API is given in Figure 1. The STM type resembles 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 top-level global variables, as atomically currently cannot be nested under unsafePerformIO. The retry and orElse combinators support condition synchronization (blocking) and composition of transactions.

2.2 STM Implementation

The run-time system supports STM at several levels. We focus on the library level (written in C) and the garbage collection support. Transactions proceed in two phases: execution and commit. The execution phase must track both read and write accesses to TVars, recording writes and providing the new values in subsequent reads. The commit phase double-checks to make sure that the transaction observed a consistent view of memory, acquires locks for TVars being updated, and performs the actual updates. If problems are encountered in either phase, 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 first 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. The structure of a TVar heap object is shown at the top of Figure 2.

After a transaction finishes execution, it is validated by comparing TRec entry values with the values in the TVars. It is committed by acquiring locks and writing values from TRec entries back into the TVars. Details about validation and commit are given in Section 4.

3 Adding Transactional Structs

In this section we discuss some problems with TVars and explain how TStruct overcomes them. 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 many 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, each of which can be written and read transactionally. We can then such store fields as key, value, and color as words in the structure and pointers to other nodes as pointer fields. Perhaps more important than saving space, TStructs avoid

```
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`.

indirection. By keeping the words and pointers close together they allow us to touch fewer cache lines than we would if we followed pointers to get to 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.

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. Figure 2 (bottom) shows the structure of a `TStruct` heap object. The size of a `TStruct` never changes and for many uses it will be known at compile time. As part of ongoing efforts, we are working to exploit compiler knowledge of `TStruct` layout 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 follows the pointer fields as it would in a `SmallMutableArray#`. In our initial implementation we noticed an increase in time spent garbage collecting over a `TVar` version of the same structure. We fixed this by enabling eager promotion [11] for `TStruct` objects, a generational GC feature that is disabled for other mutable primitive objects. A common mutable array based workload will loop over array entries computing new values that will be live only until replaced in the next loop iteration. In this context the array will be long lived, but the values held by the array will not and should not be promoted eagerly. As we are using `TStruct` to build concurrent data structures, it will be common for the values written to a `TStruct` to have the same lifetime as the `TStruct` itself. We suspect that `TVars` do not suffer from opting out of eager promotion because the indirections that point to `TVars` eagerly promote the `TVar`. With `TStruct` we have removed the indirection and do not get this side benefit.

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, however, 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 `TStructs` are immutable so we have pure functions that give the number of pointers and words. While simple, the current API requires the use of unsafe features; this limitation could be removed with language extensions (see Appendix A) or type-indexed products.

For some data structures, we provide data structure specific initialization actions that are non-transactional. When `TVars` 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. Non-transactional writes are also not atomic and are only used for initialization before the `TStruct` is visible to multiple threads. In future work we would like to explore an API that gives static guarantees that these non-transactional accesses happen only on private structures. An example of code written using `TStruct` primitive operations can be found in Appendix A.

3.3.2 Run-Time System Details

To support `TStruct`, the existing STM runtime is augmented with a separate list of `TRec` entries to track `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

As noted above, 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 `TVars` and `TStructs`; subsequent reads are satisfied from—and writes recorded in—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 puts into effect the changes in the `TRec`, but only if the state of shared memory matches

the view recorded in the TRec. That is, commit should happen only 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 in a successfully committing transaction is the same as the view during execution, that this view is internally consistent, and that other transactions will be prevented from committing conflicting changes concurrently.

The commit phase comprises 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 must never both reach their update step (two transactions conflict if they use the same memory location and at least one writes to it).

4.2 TVar Commit

Pseudocode for the existing GHC STM commit operation appears in Figure 4. As in OSTM, the value word in a TVar is also the lock variable. As TVars hold only references to heap objects, a locked state can be indicated by referencing a special heap object that cannot be referenced by user code. As described in Section 2.2, the special heap object used is the locking transaction's TRec. In a simplification of OSTM, GHC does not share access to TRecs 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. When a transaction first reads a TVar it must load the current value from the TVar. If that load sees a TRec it spins, waiting for the locking transaction to complete. 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, TVars contain a version field which is incremented with every update. The validation step needs to do three things: acquire locks for TVars 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 TVars in the read set. The read check step ensures that the view of memory seen during both execution and validation is 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 TVars, 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 the validate step, when the expected value, zero, stored in the TRec for x, failed to 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 the validate step, together the values (again). For the 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 the validate step, we narrow the window for conflicting commits considerably.

4.4 TStruct Commit

In TVars, the lock and the value were conflated. We cannot use the same technique in TStructs because they include 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. Pseudocode for TStruct commit appears in Figure 4. An additional field is included in TStructs for a lock count, as multiple fields in a TStruct may be written in a transaction. Validation must now handle the possibility that multiple entries—reads and writes—will be 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 already be locked by this transaction, however. If it is, 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 each entry in the read set for a matching value and for 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 update steps for that location can interleave. Updates first acquire the lock in the validate step, then increment the version and unlock by writing the new value in the update step. 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_0 that is committing and only reads x by considering two parallel timelines, one for T_0 and one for the ordered updates to x, as shown in Figure 6. Transaction T_0 will value check x, read the version, then value check x again in the validate step. Later in the read check step, T_0 will value check x and then check that the version matches. During the span between T_0's reads of x's version, we know immediately

```
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 pseudocode for `TVars` on the left and `TStructs` on the right.

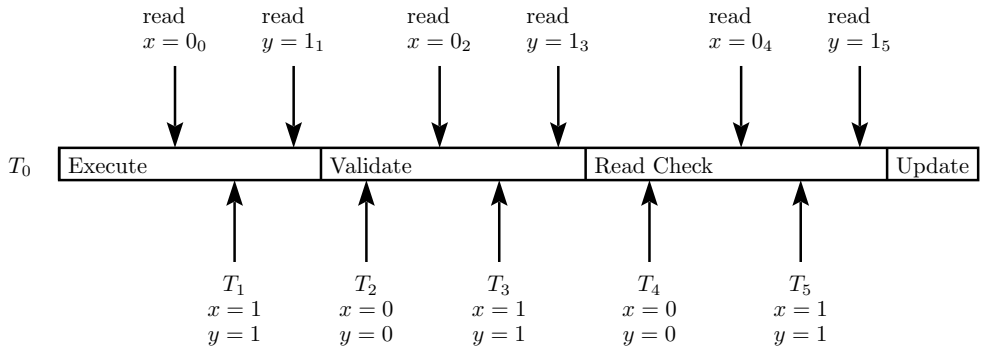


Figure 5. Timeline illustrating a sequence of commits that could lead to committing with an inconsistent view of memory. This diagram is from the perspective of T_0's execution and commit. Reads of shared memory (with values subscripted with the corresponding version number) are noted above the line;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

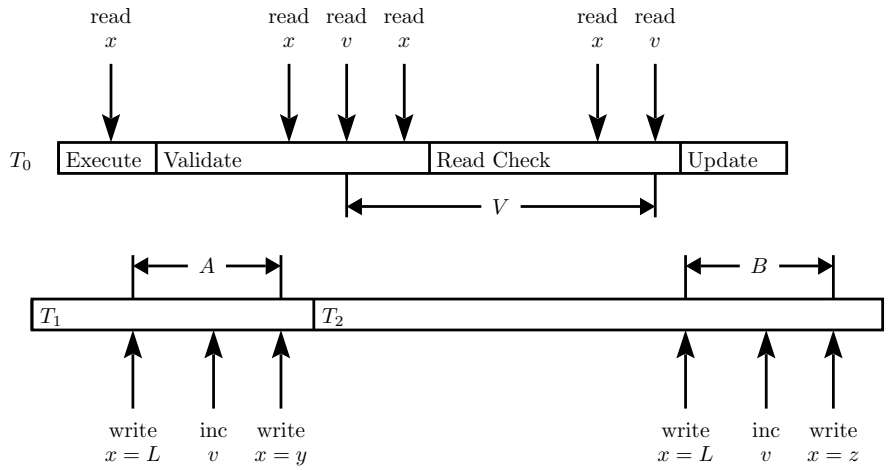


Figure 6. Timeline of possible interactions with a committing read-only transaction T_0 and other transactions T_1 and T_2 with respect to a `TVar` with value/lock field x and version field v. Ranges A and B cannot overlap with each other due to locking x and any overlap of these ranges with a read of x in T_0 will be detected. If any increments of v happen between T_0's reads of v (range V) it will also be detected.

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_0'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_0 has a span in which no updates have happened, and the entire read set is visited in the validate step before it is visited again in the read check step, 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_0's written locations, because those locations were locked. The updates can then be ordered after T_0. 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_0'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 field 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 has the same implication as with `TVars`: the view of memory matches the view seen in execution, and no conflicting updates can happen until after a point at which the locks were 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 tables, and hashed array mapped tries. We chose these structures because each demands a unique set of features from our `TStruct` implementation. Experience with these structures helped to guide our implementation to be robust and featureful.

5.1 Red-Black Tree

Red-black trees achieve $O(\log n)$ operations by "coloring" each node, requiring the same number of black nodes on every path from the root to a leaf, and forbidding consecutive red nodes on any such path (thereby ensuring that no two paths differ in length by more than a factor of two). Among the data structures we implemented, the red-black tree has the simplest constituent nodes: each contains only a key and value, a color, and pointers to the parent and two children. Moreover each node points only to nodes of the same type and layout, and every node is the same size. Code for the red-black tree is quite complex, however, due to its rebalancing operations, which provide good worst-case performance without relying on probabilistic outcomes.

Significant effort has been devoted to concurrent red-black trees, including transactional versions [2, 3, 12]. The transactional memory approach makes it easy to write concurrent data structures that are difficult to express with fine-grain locks, while maintaining the potential for a high degree of concurrency. Moreover, transactions allow an arbitrary set of operations to be composed into a single atomic operation. Even with high concurrency, however, transactional memory introduces instrumentation and (in Haskell) indirection overhead that can significantly compromise performance. `TStructs` allow transactions to operate on nodes containing both

references and values, thereby decreasing memory overhead and increasing locality of reference.

5.2 Skip List

Skip lists [18] achieve performance comparable to that of a balanced tree by relying on randomization rather than rebalancing. This strategy leads to substantially simpler code. Like red-black trees, skip lists have been a fruitful target for concurrency research, including transactional versions [7].

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 both inter-thread contention and the overhead of transactional accesses; we therefore employ a separate generator for each thread and ensure that each is on a separate cache line. We must be careful with the generator accesses, which are non-transactional, due to the implementation of `retry`, which waits for a change to one of the variables in the transaction's read set before reawakening the thread. If the decision to execute `retry` were predicated on a non-transactional access, deadlock could occur, as the run-time system has no way to tell that the non-transactional state has changed. We use `unsafeIOToSTM` to perform non-transactional accesses, and take care not to leak information from the state of the random number generator. Note that nondeterminism is already common in transactions, since the schedule of transaction execution may influence 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 than the red-black tree nodes due to the varying number of levels. 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 [15] 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 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 only change one location. Insertions look for a free bucket among the two locations and often will be done with a small change at the chosen 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. On an insertion or deletion, a new bucket is created, copying appropriate entries. In the `TStruct`-based implementation, we have an immutable array of pointers to mutable `TStruct` buckets. Insertions and deletions simply update entries in the bucket. The `TStruct` buckets are a 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) [1] is commonly used in Haskell in its pure functional form as the underlying structure in the `unordered-containers` package for the `Map` and `Set` abstractions [20].

The "Hashed" part of the HAMT name indicates that the key is hashed before indexing to ensure a uniform distribution, thereby avoiding (as in a skip list) the need for rebalancing. The "Trie" part of the name indicates that the key is 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. While the maximum depth of the trie is fixed for a given length of key, levels tend to become more and more sparse as one moves down the tree (especially with a good hash function), and lower levels may not be needed at all.

Variable sparsity argues for a representation that avoids allocating space for unused pointer fields. 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 typically requires an entirely new node. With immutable nodes such replacement is expected. There is ample room for intermediate designs, however, with some 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.

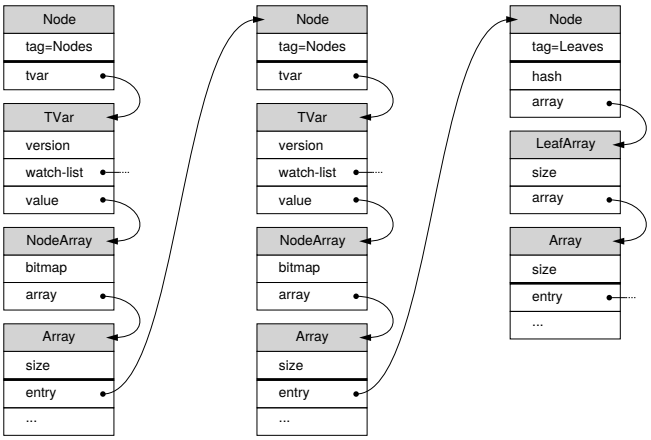
5.4.1 Transactional Implementation

We use an existing Haskell implementation of HAMT found in the `stm-containers` package [21], 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 our `TVar`-based HAMT and corresponding code is given in Figure 7a. 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 `Bitmap` field in `NodeArray`.

In our `TStruct`-based HAMT, nodes are either a `Nodes` level or `Leaves` leaf, as shown in Figure 7b. 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.

5.4.2 HAMT Comparison

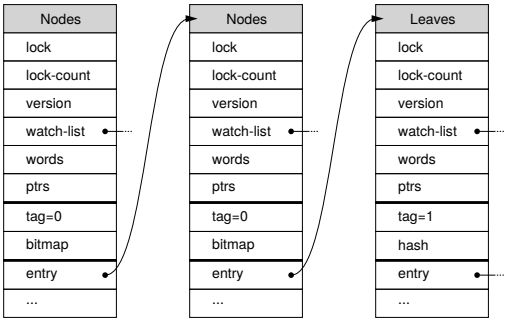
The HAMT falls somewhere in between the red-black tree and skip list in complexity. Most of the difficult aspects of HAMT lie in the data representation. Here `TStruct` makes things somewhat simpler




```
data Node a = Nodes (TVar (NodeArray a))
            | Leaf Hash a
            | Leaves Hash (LeafArray a)

data NodeArray a = NodeArray Bitmap (Array (Node a))
data LeafArray a = LeafArray Size   (Array a)
```

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




```
-- Conceptually:
data Node a = Nodes  Bitmap (Array (Node a))
            | Leaves Hash   (Array a)
-- Actual type:
data Node a = Node (TStruct# RealWorld Any)
```

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

Figure 7. The `TVar`-based (a) and `TStruct`-based (b) node data types and diagrams showing two level nodes and a leaf node of an HAMT.

although (in the absence of compiler support) with significantly less safety—more on this in Section 7. HAMT nodes come in several forms and sizes.

Prokopec et al. [16, 17] have explored concurrent versions of the HAMT. Interestingly, their implementation includes an extra indirection, as in the `TVar`-based version. This indirection facilitates low-cost snapshots. A Haskell version of this concurrent HAMT can be found in the `ctrie` package [19]; we include it in the performance experiments of Section 6.

6 Performance Evaluation

Our modifications were made to the 8.0.2 version of GHC. Results were obtained on a 2-socket, 36-core, 72-thread Intel Xeon E5-2699 v3 running Fedora 24. 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. The best performance was achieved by first filling all the cores on one socket, then moving to the second socket, and finally using the second hardware context (hyperthread) on each core.

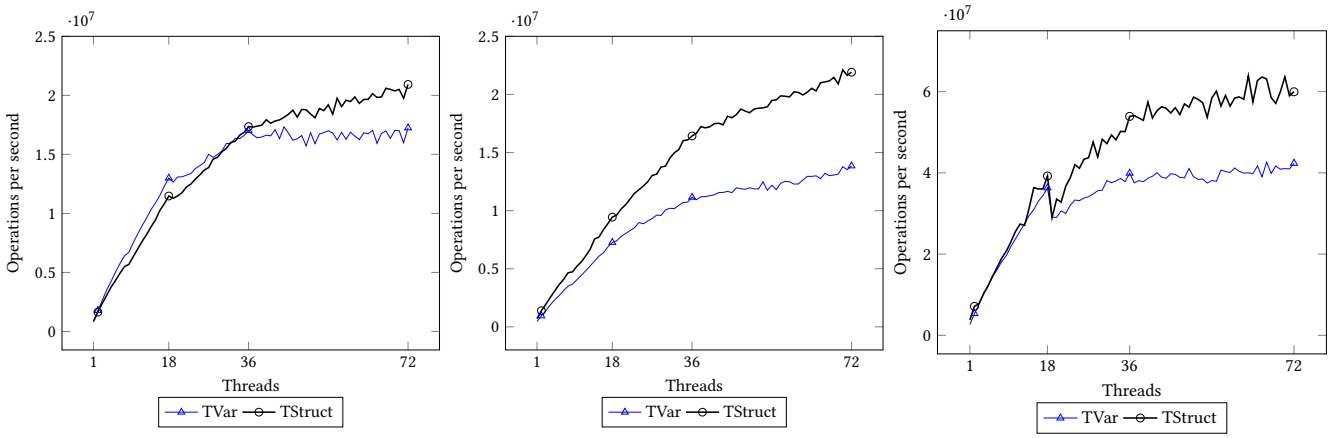
6.1 Data Structure Throughput

Our microbenchmarks focus on steady-state data structure throughput. Figures 8 and 9 plot this throughput for a mix of operations on data structures representing a set which initially has 50,000 entries. During execution, each thread repeatedly performs a transaction that searches for a random key (from a key space of 100,000) 90% of the time, inserts a random key 5% of the time, and deletes a random key the remaining 5% of the time. Due to the mix of operations, the key space, and the initial occupancy, the structure is expected to keep its size regardless of the length of the run. Given this stability, we can expect half of insertions and deletions to follow a read-only path (with respect to the Haskell transaction), where an insertion finds that the entry in question already exists, or a deletion finds that it does not.

For all four data structures, `TStruct` results in noticeably better performance, especially at high thread counts. Details vary with the structure. In the red-black tree, `TVars` actually out-perform `TStructs` by a small amount until hyperthreads are introduced, after which `TVar` performance is flat but `TStruct` continues to improve, eventually outperforming `TVar` by 21%. Several factors may contribute to this behavior, including false conflicts introduced by `TStructs`. For example, consider a transaction that is updating the color of a grandparent node g after an insertion. In the `TVar` version this color update does not conflict with another transaction that reads g. In the `TStruct` version there can be a conflict if the read (either the initial read or a read during commit) happens while the updating transaction is committing. The whole `TStruct` is locked for the write to the color field. We hope to explore different lock granularity for `TStruct` in future work.

The skip list and cuckoo hash table implementations show even larger benefits for `TStruct`, peaking at 58% and 41% increases in throughput respectively. The decreased memory use due to eliminated indirection allows the benchmark to benefit from hyperthreads despite the shared L1 cache.

Finally, in HAMT, `TStruct` outperforms `TVar` by 23% on a single thread and 15% at 18 threads, where throughput peaks. Neither the second socket nor the use of hyperthreads enables additional throughput for HAMT.



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

Figure 8. 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).

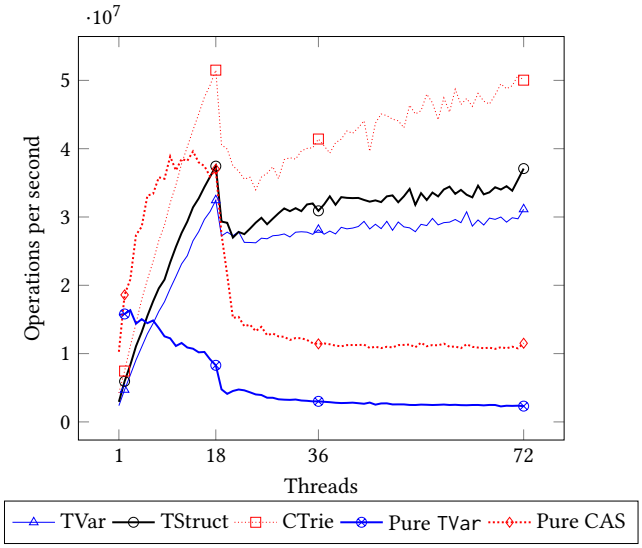


Figure 9. Operations on HAMT as in Figure 8. Also included are a fine-grain `IORef`-based concurrent CTrie and pure functional structures accessed transactionally (Pure `TVar`) or through an `IORef` (Pure CAS).

For comparison purposes, we also include in Figure 9 the performance of three alternative algorithms. The first is a concurrent implementation of HAMT (CTrie) that uses compare-and-swap (CAS) operations on `IORefs` in each node. The other two are pure functional implementations from the `unordered-containers` package, one accessed through an `IORef` and the other through a `TVar`. The `IORef` is updated using `atomicModifyIORefCAS` from the `atomic-primops` package [13]. This operation attempts a speculative evaluation of the update to the data structure before performing a CAS. If unsuccessful, it falls back to `atomicModifyIORef`, which atomically swaps in a thunk of the update before evaluating. The `TVar` version is transactional, supporting the same features and composition as any other transactional version, but with global granularity. Note that neither CTrie nor Pure CAS supports transactional blocking or composition, focusing instead on single data structure performance. As expected, these two alternatives provide the highest throughput at low thread counts. CTrie has better scalability, beating out Pure CAS after 12 threads and continuing to improve until moving to the second socket. The cost of the global bottleneck increases significantly with the second socket, greatly degrading performance. The pure `TVar` version fails to scale, with increased threads only harming performance.

We also instrumented the run-time system to track the amount of memory allocated during a transaction. Our methods were very lightweight, simply taking a snapshot of the heap pointer at the beginning and end of a transaction and recording amounts requested from `allocate` (such as STM metadata). Values from certain corner cases—such as when GC happens in the middle of a transaction—are detected and ignored, leading to modest imprecision in our measurements. When developing our benchmarks, we used this instrumentation to guide our coding and to reduce allocations of unneeded intermediate structures such as wrappers around unlifted heap objects. We focused our attention on the read path of each benchmark as it was significantly less code and it was executed the majority of the time. In all cases the `TStruct` version of each benchmark allocated less: 32% as much as `TVar` in HAMT; 52% as much in red-black tree; 41% as much in skip list; 10% as much in the cuckoo hash table.

Without eager promotion, the `TStruct` red-back tree and HAMT both had peak performance matching that of their respective `TVar` versions. In both cases if we ignored GC time and calculated throughput as transactions committed per second of mutator time, the `TStruct` versions had higher throughput. For the cuckoo hash table and skip list, eager promotion made no significant difference to performance. This is unsurprising, at least in the cuckoo hash case: the mutable buckets contain only references to keys and values, not to further structure.

7 Future Work

While we have seen performance improvements for applications with `TStruct`, we have not been satisfied with the code that had to

be written to achieve this. We are therefore working on compiler support for transactional structs, to improve the quality of generated code and to provide better safety and simplicity to programmers. As mentioned in Section 3.3.1, non-transactional initialization of `TStructs` can be guaranteed to be safe in common scenarios. In ongoing work we are exploring an API that exposes these accesses safely. Specifically, we are building on a recent proposal by Simon Marlow to add data types with mutable fields [10]. In this proposal, data constructors with mutable fields are `IO`, `ST`, or `STM` actions, while pattern matching on a constructor introduces references to mutable fields rather than to values. These references are represented internally as the pairing of an offset with a pointer to the heap object. Additional actions allow reading and writing to fields within the proper context. Simple extensions to generalized algebraic data type (GADT) syntax give a clean way to express these data types with the context required for their access and creation. Some details and a small example are given in Appendix A.

We also hope to explore more data structures that can benefit from transactional structs. This will likely lead us to explore improved transactional array support as well. There are several variations on 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 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 also compromise performance due to increased conflicts.

Previous work has suggested that performance may be improved by using a different data structure for the transactional record [9]. Additionally, the granularity of STM metadata could be explored for further improvements. A related 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 optimization 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, in which transactions are first attempted using hardware transactional memory. The idea was that reduced indirection would reduce the frequency with which transactions overflowed their limited hardware buffers. Along the way we discovered that `TStruct` improved performance of software-only transactional memory 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.

While we get better or similar performance by enabling eager promotion for `TStructs`, we would like to better characterize when this will be a benefit and when it will harm performance. For example we expected some benefit for skip lists but we saw none. With a better characterization we may be able to develop GC policies that can suit a broad set of workloads.

Given our focus on microbenchmarks, it is not yet clear how our results will translate to 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. Typical current applications use a pure functional data structure and gain mutation by accessing the whole structure through a single mutable cell, with appropriate synchronization (usually `atomicModifyIORef`). This pattern works well on low core counts, but fails to scale as the single cell inevitably becomes a bottleneck [14].

A STM Mutable Fields

In writing our data structure implementations with `TStruct` we sacrificed safety in several ways. In this appendix we look at the HAMT node representation and some of the safety lost, and then show how an extension to the mutable fields proposal can recover safety while giving opportunities for better code generation.

Figure 7b shows the `TStruct` node representation for HAMT. Each `Nodes` will point to either another `Nodes` level or a `Leaves`, while `Leaves` will always point to values held by the data structure. Normally we would use a sum type to encode these possible children but this would introduce an indirection, as user declared sum types cannot be mutable. Instead we keep a tag field to indicate the constructor as the first word in the `TStruct`, as seen in the diagram in Figure 7b. This tag field is immutable; the child references may change as children are updated. It would be costly to access immutable fields transactionally so we provide two API's for accessing `TStructs`, transactional and non-transactional (suffixed `NT`). Consider the following code for a `contains` function on a `TStruct` HAMT based set:

```
contains :: Key → Level → Node Key → STM Bool
contains k level (Node tstruct#) = do
  tag ← readTStructWordNT# tstruct# 0#
  case tag of
    0 → do — A Nodes object
      let i = hashIndex level (hash k)
      bitmap ← readTStructWordNT# tstruct# 1#
      if testBit bitmap i
        then do
          let (I# p#) = position bitmap i
          child ← readTStructPtr# tstruct# p#
          contains k (level+1) (Node (unsafeCoerce# child))
        else return False
    1 → do — A Leaves object
      h ← readTStructWordNT tstruct# 1#
      if hash k == h
        then find k tstruct#
        else return False
```

This representation sacrifices safety in four ways:

1. While our use of a non-transactional read of the bitmap field is safe, nothing enforces the immutability of that field. For example, we could consider implementing deletion of a child from `Nodes` where the child array is compacted and the bitmap is updated. This change would require a transactional read.
2. The tag, bitmap, and hash fields are always accessed with constant offsets that must be correct. No bounds check is performed and out of bounds accesses would lead to undefined behavior.

3. Similarly the value from the tag field must be handled consistently in several places.
4. We use the highly dangerous `unsafeCoerce#` as we keep references to unlifted `TStructs` in `Nodes` objects and references to lifted values in `Leaves`. Even small modifications to this code can quickly lead to mistakes.

The mutable fields proposal [10] with a few extensions addresses all of these issues. First we extend the available contexts for mutation to include the `STM` monad. We also allow a single mutable array field, marked by the `mutableArray` keyword, as the last entry in a constructor. This will be represented as an array of fields at the end of the object. The size of this array and initial values are provided at creation time. When a `mutableArray` is pattern matched, the name is bound to a reference to the array that can be accessed by a transactional API (in the context of STM). We also allow sum types mixing mutable and immutable constructors. The HAMT data declaration with our mutable fields syntax would be the following:

```
data Node a where
  Nodes  :: Bitmap → mutableArray (Node a) → STM (Node a)
  Leaf   :: Hash → a → Node a
  Leaves :: Hash → mutableArray a → STM (Node a)
```

We recover the singleton `Leaf` constructor from the original version found in Figure 7a. If we had included this in our `TStruct` version we would have lost any benefit in `TStruct` metadata overhead. With this `Node` type we can write `contains` as follows:

```
contains :: Key → Level → Node Key → STM Bool
contains k level (Nodes bitmap ns) = do
    let i = hashIndex level (hash k)
    if testBit bitmap i
      then do
        child ← readArrayRef ns (position bitmap i)
        contains k (level+1) child
      else return False
contains k _ (Leaf _ k') = k == k'
contains k _ (Leaves h ls)
  | hash k /= h' = False
  | otherwise    = find k ls
```

What were previously non-transactional accesses (safety issue 1) are now fields in a data constructor which are inherently immutable and safely accessible, even outside STM. The `case` statement matching on the tag field is now a pattern match giving us bindings (safety issue 2) and exhaustiveness warnings (safety issue 3). Finally we have no need for `unsafeCoerce#` as our data declaration expresses exactly the types we need for each constructor (safety issue 4). We are optimistic that an implementation based on mutable fields will give us the performance benefits of `TStruct` while still allowing concise, robust, and safe code.

Acknowledgments

This work was funded in part by the US 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.

References

[1] Phil Bagwell. 2001. *Ideal hash trees.* Technical Report. School of Computer and Communication Sciences, EPFL. http://lampwww.epfl.ch/papers/idealhashtrees.pdf.

[2] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Bangalore, India.

[3] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge Computer Laboratory.

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

[5] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Chicago, IL, USA.

[6] Timothy L. Harris, James R. Larus, and Ravi Rajwar. 2010. *Transactional Memory* (second ed.). Morgan & Claypool, San Francisco, CA.

[7] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, USA.

[8] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers.

[9] Matthew Le, Ryan Yates, and Matthew Fluet. 2016. Revisiting Software Transactional Memory in Haskell. In *Proc. of the 9th Intl. Symp. on Haskell*. Nara, Japan.

[10] Simon Marlow. 2016. Mutable Constructor Fields. (2016). https://github.com/simonmar/ghc-proposals/blob/mutable-fields/proposals/0000-mutable-fields.rst

[11] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. 2008. Parallel generational-copying garbage collection with a block-structured heap. In *Proc. of the 7th Intl. Symp. on Memory Management*. Tucson, AZ, USA.

[12] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*. Seattle, WA, USA.

[13] Ryan R. Newton. 2016. atomic-primops: A safe approach to CAS and other atomic ops in Haskell. (2016). http://hackage.haskell.org/package/atomic-primops

[14] Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. 2015. Adaptive Lock-free Maps: Purely-functional to Scalable. In *Proc. of the 20th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*. Vancouver, BC, Canada.

[15] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004).

[16] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Cache-Aware Lock-Free Concurrent Hash Tries.* Technical Report. School of Computer and Communication Sciences, EPFL. https://infoscience.epfl.ch/record/166908.

[17] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In *Proc. of the 17th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*. New Orleans, LA, USA.

[18] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990).

[19] Michael Schröder. 2013. ctrie: Non-blocking concurrent map. (2013). http://hackage.haskell.org/package/ctrie

[20] Johan Tibell. 2012. unordered-containers: Efficient hashing-based container types. (2012). http://hackage.haskell.org/package/unordered-containers

[21] Nikita Volkov. 2016. stm-containers: Containers for STM. (2016). https://hackage.haskell.org/package/stm-containers