Revisiting Software Transactional Memory in Haskell^{*}

Matthew Le

Rochester Institute of Technology, USA
ml9951@cs.rit.edu

Ryan Yates

University of Rochester, USA
ryates@cs.rochester.edu

Matthew Fluet

Rochester Institute of Technology, USA
mtf@cs.rit.edu

Abstract

Software Transactional Memory (STM) has become very popular in Haskell. Currently, there are nearly 500 packages on Haskell's package archive that directly use STM. Despite the widespread use in real world applications, Haskell's STM implementation has seen very few updates since its introduction in 2005.

In this work, we describe our efforts to redesign the STM implementation in the Glasgow Haskell Compiler (GHC), based on a TL2-like implementation that is able to support both `orElse` and `retry` without the use of traditional nested transactions. We argue that our implementation is simpler than the current GHC implementation while supporting opacity. We also demonstrate that our implementation performs better than the current GHC implementation on a number of benchmarks by multiple orders of magnitude for long-running transactions.

In an effort to measure the performance of `orElse` and `retry`, we present an STM-based work stealing scheduler. With `orElse` and `retry`, we are able to elegantly implement the scheduler in just a few lines of code. We have modified the `Par Monad`, a real-world Haskell package that provides deterministic parallelism, to use our STM-based work stealing scheduler and show that it is not only simpler but is able to perform as well as the current scheduler.

Categories and Subject Descriptors D.3.3 [Language Constructs and Features]: Concurrent programming structures; D.1.3 [Concurrent Programming]: Parallel programming; D.3.2 [Language Classifications]: Concurrent, distributed, and parallel languages

Keywords Transactions, Haskell

1. Introduction

Haskell provides an excellent context for implementing transactional memory. Its rich type system allows one to sidestep several issues that arise when implementing software transactional memory in other languages. For example, Haskell's type system is able to distinguish statically code that may perform IO and code executing in a transaction. Additionally, given the preexisting division between functional and imperative code, it is natural to separate between transactional and non-transactional accesses to memory,

^{*}This research was funded in part by the National Science Foundation under grants CNS-1065099, CCF-1337224, and CCF-1422649, and an IBM Canada CAS Research Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...\$15.00
<http://dx.doi.org/10.1145/2976002.2976020>

```
data TVar a = ...
instance Monad STM where ...

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

retry :: STM a
orElse :: STM a -> STM a -> STM a

atomically :: STM a -> IO a
```

Figure 1. STM Haskell's interface

effectively eliminating the privatization problem and the complexity that it introduces (Marathe et al. 2008).

The STM Haskell design and implementation has received much praise for its expressive interface that includes blocking on a precondition within a transaction (using the `retry` mechanism) and composing alternatives (using the `orElse` mechanism). Together, these features have made STM in Haskell very popular, and it is used in a wide array of publicly deployed projects. Despite its popularity, the implementation itself has seen only minor changes since its introduction in 2005 (Harris et al. 2005).

In this paper, we discuss our experience redesigning the STM implementation for Haskell in the industrial-strength Glasgow Haskell Compiler (GHC). The new implementation is based on TL2 (Dice et al. 2006) and is able to support all of the prior functionality included in its introductory paper, namely `retry` and `orElse`.

2. Background

While GHC's STM implementation sees significant use, most of the research in transactional memory (TM) has occurred outside of the Haskell community. Haskell's purity is a wonderful fit for TM, avoiding the need for significant effort to control effects in transactional execution. The TM community has worked hard to develop high performance implementations and study fundamental trade-offs in transaction execution. Broadly speaking, STM implementations differ in how they store and manipulate transaction metadata during execution, how they validate to ensure the system stays consistent, and how they commit transactions.

The STM functionality in Haskell is encapsulated in its own STM monad, which can be seen in Figure 1. Transactions operate on `TVars`, where a new `TVar` can be created using `newTVar`, which takes an initial value of type `a`. Reads and writes are performed using `readTVar` and `writeTVar` respectively. Actions of type `STM a` can be turned into IO actions with `atomically`. This also delineates a transaction where all the effects in the original STM action are seen atomically from any other thread. Separating the STM op-

erations into their own monad guarantees TVars are accessed transactionally and that no transaction unfriendly effects such as I/O can occur within a transaction.

STM Haskell additionally provides support for blocking if a precondition does not hold within a transaction. The `retry` mechanism is used to abort a transaction and wait until one of the TVars that it read has been modified. The `orElse` function is used for composing alternatives: (`orElse t1 t2`) will execute the `t1` transaction, and if it ends up calling `retry`, any writes from `t1` will be ignored and the `t2` transaction will be executed. If `t2` also ends up calling `retry`, then the entire `orElse` retries. These features are very commonly used in STM Haskell code. As an example, we could implement a queue that blocks a dequeue operation if the queue is empty as follows:

```
data TQueue a =
  TQueue (TVar [a]) (TVar [a])

helper :: TQueue a -> STM (Maybe a)
helper (TQueue read write) = do
  rs <- readTVar read
  case rs of
    (r:rs') -> do
      writeTVar read rs'
      return (Just r)
    [] -> do
      ws <- readTVar write
      case reverse ws of
        (w:ws') -> do
          writeTVar write []
          writeTVar read ws'
          return (Just w)
        [] -> return Nothing

dequeue :: TQueue a -> STM a
dequeue q = do
  x <- helper q
  case x of
    Nothing -> retry
    Just x -> return x
```

If `helper` finds that the queue is empty, we simply call `retry`, which (in the absence of any successful `orElse` alternatives) will abort the transaction and try again as soon as the `read` or `write` TVar, the two TVars read in `helper`, have been modified.

Blocking transactions with `retry` and composing transactions with `orElse` are features not included in most transactional memory implementations. While GHC's `orElse` feature support extends into the runtime system, it is mostly a convenience feature (a great convenience indeed!) giving the ability to compose potentially blocking transactional code without needing to be concerned with the internal implementation details of the transactions. The `retry` feature is more fundamental, especially for an efficient implementation. Section 4.4 shows how we extend an existing TM implementation from outside of the Haskell community to support `retry` and `orElse`.

3. Current STM Haskell Implementation

In this section we provide a brief overview of the current STM implementation in GHC. Additional details can be found in (Harris et al. 2005). The implementation spans three levels in GHC. First there is the Haskell language level functions which the application programmer uses (Figure 1). These Haskell level functions then make use of C-- (Peyton Jones et al. 1999) primitive operations which in turn make calls into the C run-time system, where the majority of the underlying STM lives.

```
typedef struct{
  Value_t  current_value;
  TSOQueue watch_queue;
  unsigned version;
}TVar;

typedef struct{
  TVar * tvar;
  Value_t expected_val;
  Value_t new_val;
  unsigned version;
}TRecEntry;

typedef struct TRecChunk_{
  struct TRecChunk_ * next_chunk;
  long next_entry_idx;
  TRecEntry entries[CHUNK_SIZE];
}TRecChunk;

typedef struct TRec_{
  struct TRec_ * enclosing_trec;
  TRecChunk * current_chunk;
}TRec;
```

Figure 2. STM Haskell Metadata

Figure 3. STM Haskell Metadata diagram

3.1 Metadata

Figures 2 and 3 show the metadata used. A TVar contains the value stored as well as a queue of thread state objects (TSOs). In GHC, a TSO contains the identity of a thread, including its numerical thread id, transactional log, call stack, etc. The queue is used to implement the blocking semantics of `retry` described in Section 3.3.

The TRec is a nested structure, where a new level is added each time an `orElse` is entered. At each level of the TRec there is a chunked linked list, where each chunk (currently) has 16 entries. Each entry contains the TVar that was accessed, the value observed in the TVar at its first access (`expected_val`), and a new value to be written into the TVar upon a successful commit (`new_val`).

Like many other functional languages, Haskell uses a generational garbage collector, which can take advantage of the fact that the vast majority of the data created is immutable. In order to deal with mutable heap objects, we must mark them with a special header so the garbage collector can place them on a *remember set*. This remember set is then treated as a root when a GC occurs. Since a chunk in the TRec is mutated every time we create a new entry, we must mark each chunk as mutable. This can incur additional overhead in the garbage collector and can keep data live longer than necessary. In Section 4 we show how to implement a chunked log without having to mark the chunks as mutable.

3.2 TVar Access

Each time a write is performed, the thread scans the TRec starting from the current nesting level. If the entry is found in the current level, then the `new_val` field of the entry is updated. If the TVar is found in an outer level of the TRec, then a new entry in the current level of the TRec is created using the `expected_val` field of the previously found entry and an updated `new_val` field. If the entry is not found at any level of the TRec, then a new entry is created with the `expected_val` field set to the current contents of the TVar and the `new_val` field set to the written value. When reading from a TVar, the TRec is scanned starting from the current nesting level. If the entry is found in the current level, then the `new_val` of the entry is returned. If the TVar is found in an outer level of the TRec, then a new entry is created in the current level of the TRec using the `expected_val` and `new_val` fields of the previously found entry returning the `new_val`. If the entry is not found at any level of the TRec, then a new entry is created with the `expected_val` and `new_val` fields set to the current contents of the TVar returning the current contents. The copying of TRec entries into the current level ensures that nested transactions can be properly validated.

3.3 Commit, orElse, and retry

When committing a transaction, the TRec is traversed. For each entry, the current contents of the TVar is compared to the `expected_val` stored in the entry. Write entries are distinguished by a difference in the `expected_val` and `new_val`. The TVar for each write entry is locked by writing the TRec into the `current_val` field. When reading from a TVar, a transaction can tell if it is locked by checking the header of the heap object. If it has the distinct TRec header, then it is locked and the thread spins until it becomes unlocked. The transaction is valid if all the expected values in the TRec match the actual values in TVars and no updates happen to these TVars while the check is in progress. To achieve this, a version number is stored on each TVar which is incremented on every update. On the first pass through the log when locks are acquired for writes, a version number is stored for each read. A second pass after all the locks are acquired checks the reads again and ensures the version has not changed and the expected value still holds. Updating and unlocking each write TVar is simply performed by writing the new value into the TVar then incrementing the version. Any threads enqueued on the TVar are woken up. If the log is invalid, then the transaction is aborted and all locks are released. It is important to note that there is a narrow window of opportunity for a pair of transactions with overlapping access sets that are committing at the same time to both abort due to seeing a locked TVar from the other transaction.

In order to implement `orElse`, nested transactions are used. Upon entering an `orElse`, a new level in the TRec is created. If the first alternative calls `retry`, then the reads of the nested log are merged into the enclosing parent log, discarding any write entries, and the alternate code path is executed. This way if the `retry` propagates to the top level, the transaction will appropriately block on the TVars read in both branches of the `orElse`. If either branch

of the `orElse` succeeds, then the nested log is validated. If the log remains valid, then the reads and writes of the nested log are merged into the enclosing parent log. If the log has become invalid, then the nested transaction is aborted and attempted a subsequent time. If a `retry` propagates to the top level of the transaction, then the thread is enqueued on every TVar in the log and returns to the scheduler.

3.4 Opacity

Opacity (Guerraoui and Kapalka 2008) is a strong correctness property which, among other things, requires that transactions always observe a consistent state of memory. The current implementation of STM Haskell does not support opacity, inducing a number of complications throughout (and beyond) the STM implementation. For example, a consequence of observing inconsistent state can be a thread entering an infinite loop. To counter this, the thread scheduler validates the thread's TRec (assuming there is one) every time it is scheduled. This occasional validation overhead, potential lost execution time, and allocation to doomed transactions is unfortunate as is the added complexity to the implementation by having interdependencies between scheduling and STM. Opacity, however, essentially requires validation at every read. We will see in the next section how TL2 avoids this high cost while still providing opacity.

4. TL2 STM Haskell Implementation

Transactional Locking II (TL2) (Dice et al. 2006) is an STM implementation that serializes transactions with respect to a global version clock. Not only is it a highly efficient STM, but it is also able to provide opacity at a low cost. We explored several options for replacement implementations and TL2 was straight forward to implement, had properties we wanted to explore, and had similarities to the existing implementation that simplified our implementation. In this section we describe our implementation of a TL2-like STM that is able to support `orElse` and `retry`.

4.1 Metadata

Figures 4 and 5 show all the necessary metadata for the STM. Unlike the existing STM which has no separate global structures, TL2 uses a global version clock for serializing all transactions. The TVar metadata is used for versioning, locking, and supporting blocking. Per-thread records track accesses in a similar way to the existing STM, but with a different implementation.

Together with the value, each TVar contains a stamp indicating the last time it was written. An even-numbered stamp indicates a version number while an odd number is used to indicate a locked TVar. The odd number can also indicate which particular thread is holding the lock such as $((\text{threadID} \ll 1) \mid 1)$. As with the current STM Haskell implementation, each TVar also maintains a queue of thread state objects to support the blocking semantics of `retry`.

Each thread then maintains a private TRec which contains a local time stamp (`read_version`) indicating when the transaction began, a read set that contains all TVars read from during the transactions, and a write set that maintains a set of pairs of TVars and values to be written.

With some help from the garbage collector, we are able to avoid marking the read set chunks as mutable. Every time that a chunk is allocated, we set the `size` field to `CHUNK_SIZE`. We are then able to add entries to the chunk via mutation until the chunk gets processed by the garbage collector (or becomes full). Heap objects in GHC Haskell all have a pointer to a static info table shared among all like heap objects. Many heap objects have special handling in garbage collection dictated by information found in the info table. The existing STM transactional records have special handling for

```

alt1 t1 t2 = do
  x <- readTVar t1
  writeTVar t2 x
  retry

alt2 t1 t2 = do
  x <- readTVar t2
  writeTVar t1 x

txn t1 t2 =
  writeTVar t1 0
  writeTVar t2 1
  orElse (alt1 t1 t2) (alt2 t1 t2)

```

Figure 7. Example using `retry` and `orElse`

Figure 8. Corresponding stack/write set layout when `retry` is called

```

unsigned version_clock = 0;

typedef struct{
  Value_t value;
  TSOQueue * watch_queue;
  unsigned stamp;
  unsigned old_stamp;
}TVar;

typedef struct RS_Chunk_{
  struct RS_Chunk_ * next_chunk;
  unsigned next_entry_idx;
  unsigned size;
  TVar * entries[CHUNK_SIZE]
}RS_Chunk;

typedef struct{
  TVar * tvar;
  Value_t value;
  WriteSet * next;
}WriteSet;

typedef struct{
  unsigned read_version;
  RS_Chunk * read_set;
  WriteSet * write_set;
}TRec;

```

Figure 4. TL2 Metadata

Figure 5. TL2 Metadata diagram.

chunks because the number of entries in each chunk changes as the program runs and the collector must know the semantics of the structure to successfully follow the pointers it contains.

We improve on the existing implementation by setting the `size` field to `next_entry_idx` when garbage collection evacuates the object as depicted in Figure 6. This effectively “seals” the chunk and processes *only* the used entries with no additional cost. The next time we try to add an entry, we will compare the `next_entry_idx` field to the `size` field; seeing the chunk is “full” and we will allocate a new chunk. This allows us to enjoy the cache performance of the chunked linked list we saw in the original STM Haskell implementation while maintaining the invariants of a generational garbage collector, specifically, avoiding traversing mutable chunks when collecting the youngest most frequent collections.

Figure 6. When garbage collecting read set chunks, “seal” the chunk by setting `size` to `next_entry_idx` and copy only the used entries

While our benchmarks may not benefit directly from this optimization, given that there is very little cost to this change, we expect it to help in many real-world applications. Consider an application that uses STM to coordinate infrequent work triggered by some user or network input. A transaction may initially run, traversing a large data structure and blocking with `retry`. Even though this data structure may only change from this infrequent user input, without sealing the transactional record will be traversed on every minor collection. This will slow down all threads by making this stop-the-world collection slower. With sealing, they are not traversed avoiding polluting the cache with memory that will likely not need to be accessed.

4.2 TVar Access

Each time a write is performed, we add an entry to the write set containing the TVar and the value to be written. Currently, the write set is simply implemented as a linked list; however, it could just as easily be extended to use the chunked list we are using for the read set. If a TVar is written multiple times, then we will have duplicate entries in the write set. This is essential for our implementation of `retry` described in Section 4.4. The user should largely be in control of poor performance due to extensive duplicate writes so we feel this is not a corner case that merits significant optimization.

When reading, we first check if the TVar exists in the write set. If so, we simply return the associated value. If it does not exist in the write set, then we read the value from the TVar, check that it is not locked, and check that the version number is less than or equal to our read version. We then record the TVar in our read set and return the value. If we find that the TVar is locked or out of date, then we try to perform a time stamp extension (Riegel et al. 2007). We do this by sampling the version clock. If we are able to validate our entire read set, then we can continue with the transaction with the newly sampled version.

4.3 Commit

When committing (see Figure 9), we first acquire locks for all TVars in our write set by writing an odd number into the `stamp` location. If we encounter an entry that we have already locked, it is a duplicate entry and we simply drop it from the list. Note that since we scan the write set in reverse chronological order, if a subsequent write to the same TVar is encountered, it is necessarily a younger write that should be overwritten. If any locks cannot be acquired, then the transaction is aborted. If locks are successfully acquired, then we obtain a *write version* by performing an atomic fetch-and-add (by 2) on `version_clock`. We then validate the read set by checking that each TVar is either not locked or locked by this committing thread. Finally we check that each TVar still has a stamp older than our `read_version`, aborting if any check fails. Once all these checks pass, we can perform the necessary updates for our write set, waking up any threads stored in the corresponding watch queues. Locks are released by writing our write version into the `stamp` field of each TVar written.

We employ two additional optimizations described in (Dice et al. 2006):

1. If the write version that we receive is greater than our read version by 2, then we know that no one else could have possibly committed in the meantime, so we do not validate the read set.
2. If our write set is empty, we avoid the commit process altogether, since checking time stamps in-flight is enough to guarantee serializability.

4.4 retry and orElse

Upon entering an `orElse`, we push a frame onto the stack that contains a pointer to the current write set (implemented as an im-

mutable linked list) and the alternate code path. If we successfully complete the first branch of the `orElse`, then we simply return, popping the `orElse` frame off the stack. If `retry` is called, then we walk the stack looking for an `orElse` frame. If we find one, then we reset our write set to the write set stored in the frame and execute the alternate code path. Note that the read set is never reset; reads from both alternatives of the `orElse` are accumulated in order to support the blocking semantics of `retry`.

Figure 7 contains a simple program making use of `orElse` and `retry` with the corresponding stack layout in Figure 8 when we reach the `retry`. We start by writing to `t1` and `t2` and then enter an `orElse`. Upon entering the `orElse`, we push a frame onto the stack that points to the first write to `t2`. When we execute `alt1`, we perform another write to `t2` and then call `retry`. At this point, we will traverse the stack looking for an `orElse` frame. We will then update the write set in the TRec to point to the write set stored in the `orElse` frame and jump to the alternate code path, `alt2`.

In the event that an `orElse` frame is not found when performing a `retry`, we will enqueue the thread on every TVar that is in the read set and yield to the scheduler. When the thread is woken up, it will restart the transaction from the beginning.

We have found this implementation of `retry` and `orElse` to be substantially simpler than the current implementation. We do not need to manage a nested transactional log, which simplifies many of the other operations and greatly reduces the overhead of `retry` and `orElse`. In the current STM implementation, we needed to validate the log and merge with the enclosing TRec when successfully completing an `orElse` and needed to merge our read entries with the enclosing log when executing a `retry`. It is worth noting that the non-destructive, functional representation of our logs is the critical difference that allows us to implement these features so efficiently and effortlessly.

4.5 Opacity

By checking timestamps and locks at each read throughout the transaction, we are able to support opaque transactions. This removes the need to validate the log when scheduling threads and raising exceptions. Additionally, by being able to eagerly detect violations during a transaction, we are able to abort doomed transactions which can save a considerable amount of time for long running transactions. Opacity is a trade-off, however, and may not always improve performance if the runtime system and programs can tolerate execution after inconsistent reads and before the detection of the inconsistency. Using a global version to boost the opacity performance (avoiding unneeded validations) can be a source of contention, especially on multi-socket machines. We hope in future work to explore other systems that avoid this particular bottleneck and other systems that do not give opacity but have much better overhead.

5. Evaluation

Our benchmark machine is a Dell PowerEdge R815 machine, equipped with 48 cores and 128 GB of physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-67. The 48 cores are provided by four 12 core AMD Opteron 6172 “Magny Cours” processors. Each core operates at 2.1 GHz and is equipped with 64 KB of instruction and data L1 cache and 512 KB of L2 cache; each processor is equipped with two 6 MB L3 caches (each of which is shared by six cores).

Figure 10 illustrates the performance of various concurrent data structures. For each, we tested the two STM implementations on two different workloads: 90% reads and 10% reads, where the remaining portion of operations are an even split of insert and delete operations. Each benchmark is the average of five executions at each thread level.

```

lock_value = (thread_id << 1) | 1

bool acquire_locks(write_set) {
  for (tvar in write_set) {
    stamp = tvar->stamp
    if (stamp == lock_value)
      continue
    if (stamp & 1 != 0
        || stamp > trec->read_version
        || cas(&tvar->stamp, stamp,
               lock_value) != stamp) {
      release_locks(trec->write_set)
      return false
    }
    tvar->old_stamp = stamp
  }
  return true
}

update(write_ver, write_set) {
  for (entry in write_set) {
    entry->tvar->value = entry->value
    entry->tvar->stamp = write_ver
  }
}

bool validate(read_set) {
  for (tvar in read_set) {
    stamp = tvar->stamp
    if (stamp != lock_value
        && (stamp > trec->read_version
            || (stamp & 1 == 1))) {
      release_locks(trec->write_set)
      return false
    }
  }
  return true
}

bool commit(trec)
{
  if (acquire_locks(trec->write_set)) {
    write_ver = fai(&version_clock, 2)
    if (write_ver == trec->read_version+2
        || validate(trec->read_set)) {
      update(write_ver, trec->write_set)
      return true
    }
  }
  return false
}

```

Figure 9. Pseudocode for TL2 commit.

5.1 Red-Black Tree and Skip List

Figures 10(a) and 10(b) contain the results for a red-black tree and skip list. These benchmarks focus on throughput in a steady-state. Insert and delete operations are a balanced portion of all operations and the number of keys is fixed to 100,000 leading to an expected size of 50,000 elements. Note that at that size, half of the insertions will be duplicate and half the deletions will not find the element so it is expected that in addition to the percentage of read operations, half of the potential write operations will be read-only. The number of threads is the number of Haskell threads that are spawned to perform transactions and matches the “-N” parameter given to the runtime system. TL2 is able to outperform GHC’s STM on both read-heavy and write-heavy workloads. The separation becomes much more apparent for the 90% read case because we are able to elide read set validation for lookup operations. Also, since we do not need to increment the version clock for read-only transactions, we reduce contention on the version clock.

5.2 Linked List

Figure 10(c) contains the results of recreating the benchmark presented in (Sulzmann et al. 2008), where the authors compare the performance of various concurrent linked list implementations in Haskell. The paper includes four different approaches. The first is a lock-free version that uses compare-and-swap (CAS) to delete and insert entries, where each node is stored in an `IORef`—a basic mutable reference cell. Second is one that uses hand-over-hand locking, or lock-coupling. Each node is stored in an `MVar`, which is a mutable cell that can take two states, full and empty. If a thread tries to “take” the contents of an empty `MVar`, it blocks until it becomes full. If a thread attempts to write to a full `MVar`, then it blocks until it becomes empty. The third implementation is a straightforward STM implementation, where the entire operation is encapsulated in a transaction. Lastly, in the “dissected” STM implementation, each individual iteration of the list traversal is protected by a transaction.

Items are inserted in the list by simply appending onto the end of the list, and lookups and deletions begin by scanning from the beginning. As a result, the list is unordered and permits duplicate entries.

In (Sulzmann et al. 2008) the authors show that the straightforward STM approach is not a feasible solution, performing multiple orders of magnitude more slowly than the alternative implementations. This is mainly due to the unified log rather than split read and write sets. The log becomes very large in this application, and in each iteration of a traversal, we must search the entire monolithic log for a previous write. Sulzmann et al. show that the “dissected” version is able to perform much better, mainly because it keeps the log to a constant size, bounding the search space of a lookup for a previous write. Our experiments confirm these results.

When we move to a TL2-based STM, the straightforward STM approach is able to perform quite competitively with the other implementations across a range of workloads. In the case where 90% of the operations are lookups, the TL2 implementation is able to perform exceptionally well, matching the performance of the CAS implementation. This is due mainly to two factors. First, since we have split the read and write sets, each iteration of the traversal looks at only the write set (which is empty). Second, for the lookup transactions we do not need to validate the read set, which can become quite large. As we scale up to more threads, we see performance starting to marginally degrade. This is most likely due to the fact that we are going to have more in-flight writing transactions. With more writing transactions in the system, we end up incrementing the version clock more frequently, which eliminates the ability to elide read set validation for the (few) writing transactions.

5.3 Work Stealing

The Par Monad (Marlow et al. 2011) is one of the preferred means of writing deterministic parallel applications in Haskell. It has seen widespread use and serves as a basis for recent research on lattice-

Figure 10. Throughput performance with transactions performing repeated transactions with a random operation on a shared data structure. SF stands for Straight Forward, D for Dissected, and MLC for MVar Lock Coupling.

based deterministic data structures (LVars) (Kuper and Newton 2013). The Par Monad is written completely in Haskell and makes use of a simple work stealing scheduler using IORefs to coordinate task stealing and MVars to put schedulers to sleep when no work is available to be stolen and woken up when new work has been produced.

In order to measure the performance of `retry` and `orElse`, we have rewritten the Par Monad to make use of an STM-based work stealing scheduler. Each scheduler thread has a scheduling deque. When threads try to pop or steal work from a deque that is empty, we call `retry`. We can then implement the work stealing operation as follows:

```
steal :: [Deque] -> STM Task
steal [] = do
  done <- readTVar shutDown
  if done
  then return Exit
  else retry
steal (d:ds) =
  popTop d 'orElse' steal ds
```

This will attempt to steal from each deque until it finds one that is nonempty. If all deques are empty, then due to the semantics of `retry`, the thread will be put to sleep until one of the TVars that it read from changes (i.e., some thread posts work to its deque). This very elegantly implements the work stealing semantics with-

out having to explicitly coordinate going to sleep and waking up threads.

The Par Monad comes with a number of simple benchmark programs that we have used to compare our STM based scheduler, using both TL2 and GHC's current STM, to the current non-STM scheduler.

As can be seen from Figure 11, the TL2-based scheduler is able to match the original work stealing scheduler almost perfectly. GHC's STM is relatively competitive for Mandelbrot but performs very poorly as the thread count increases for the other benchmarks. The Mandelbrot benchmark is written in a divide-and-conquer style, where in each iteration we divide the workload in half across the rows. This is able to distribute work among the schedulers fairly effectively, making successful steal attempts more likely when needed.

MiniMax uses alpha-beta searching of a game tree to find the best move in a 4×4 game of X's and O's and NQueens calculates the number of solutions to the N -queens problem for a board of size 14×14 . Both of these benchmarks exhibit irregular nested parallelism. As we scale up to the larger thread counts, GHC's STM is unable to compete due to the fact that threads looking for work need to visit more deques leading to large `orElse` chains.

GHC's STM is able to perform quite well on NBody for the lower thread counts, but once we scale up, it abruptly falls apart. This benchmark is written to have the main thread spawn all of the tasks, and no task spawns any further sub-task. Thus, there is only

Figure 11. Work Stealing Benchmark

ever one deque in the system that produces any work, and all other threads are competing to steal with it. For the lower thread counts, the main thread ends up running on the first Capability (GHC’s “virtual CPU”), however, when we move past 20 threads, it ends up getting migrated to one of the higher numbered capabilities.

The work stealing scheduler in the Par Monad currently does not implement randomized work stealing. Instead, each thread gets an ordered list of the scheduling deques from 1 to N and every thread attempts its steals in the same order. When the main work-producing thread is scheduled on the first capability, every other thread is able to access it on their first attempt. When the main work-producing thread gets moved to one of the higher numbered capabilities, the only successful steals end up happening much further down the line, leading to large chains of `orElse`s.

This is a problem for GHC’s STM for two reasons. First, the long chain of `orElse`s introduces a non-trivial amount of overhead due to the fact that each time `retry` is called, we must merge the reads from the nested log into the parent log so that we can appropriately wait if the `retry` propagates to the top-level. Second, we have noticed a livelocking phenomenon in which stealers continually abort each other such that no thread is able to make progress. This happens due to validation of the nested log when an `orElse` successfully completes. This validation acquires locks on all written TVars and merges with the enclosing log. When validating the

top level transaction, we also acquire locks on our written TVars and validate the log. Thus, a transaction writing in an `orElse` must “win” twice when acquiring locks for the written TVars. The stealing threads end up making close to zero progress and the work-producing thread ends up having to execute all of the tasks, degrading down to sequential performance.

Our TL2-based STM does not have this issue because we only need to lock written TVars once. When two stealers compete for the top of the deque, one is guaranteed to win every time. Additionally, since we do not have nested logs, there is no need to merge reads when we call `retry`.

We expect that a rewrite of the Par Monad to make use of randomized work stealing would lead to improved performance for GHC’s current STM. If threads are not reading from deques in the same order every time, it is likely that the livelocking phenomenon would be reduced. That said, the overhead of merging reads at every `retry` is still going to impose a non trivial overhead, making STM Haskell a non-viable option for this application.

It is worth noting that validating the log when successfully completing an `orElse` is not strictly necessary for correctness. If we were to remove this check, we would most likely avoid the livelocking issue. However, the commit protocol would need to be modified in some way to deal with the nesting when committing a top-level transaction.

6. Conclusion

Our new STM implementation shows significant performance improvements over the existing GHC implementation on a range of benchmarks including one real-world Haskell application. Leveraging the TL2 algorithm and extending it with support for Haskell's unique composable blocking features resulted in a simpler implementation avoiding traditional nested transactions and nested log structure.

References

- D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Stockholm, Sweden, 2006. Springer-Verlag.
- R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, Salt Lake City, UT, USA, 2008. ACM.
- T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, Chicago, IL, USA, 2005. ACM.
- L. Kuper and R. R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, Boston, Massachusetts, USA, 2013. ACM.
- V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 67–74, Portland, Oregon, USA, 2008. IEEE Computer Society.
- S. Marlow, R. R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, Tokyo, Japan, 2011. ACM.
- S. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In *Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, PPDP '99, pages 1–28, Coimbra, Portugal, 1999. Springer-Verlag.
- T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, San Diego, California, USA, 2007. ACM.
- M. Sulzmann, E. S. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in Haskell. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, DAMP '09, pages 37–46, Savannah, GA, USA, 2008. ACM.