Revisiting Software Transactional Memory in Haskell\textsuperscript{1}

Matthew Le\textsuperscript{2}, Ryan Yates\textsuperscript{3}, and Matthew Fluet\textsuperscript{2}

Rochester Institute of Technology\textsuperscript{2}
University of Rochester\textsuperscript{3}

Haskell Symposium, 9-23-2016

\textsuperscript{1}This work was funded in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Studies.
Transactional memory is the joining of two ideas:

- The ability to express what should be **atomic** without saying how.
- An implementation that uses **speculation** to optimistically execute and try again if needed.
Haskell STM adds two ideas:

- Efficient transaction **blocking** with **retry**. Wait until conditions may be right, then try again.
- **Composition** with **orElse**. If conditions are not right for one computation, try a different one.
Implementations

- Current GHC fine-grain locking implementation [Harris et al., 2005] based on OSTM [Fraser, 2004] (*STM-Fine*).
- Our new implementation based on TL2 [Dice et al., 2006, Riegel et al., 2007] (*TL2*).
Red–Black Tree

(AMD Opteron™ 48-cores across 4-sockets)

Red–Black Tree 90% Reads

TXns/second

Threads

0 1 2 3 4 5 6 7 8

STM-Fine
TL2
Skip List

(AMD Opteron™ 48-cores across 4-sockets)

Skip List 90% Reads

-10^6

TXns/second

<table>
<thead>
<tr>
<th>Threads</th>
<th>STM-Fine</th>
<th>TL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linked List\(^2\) (AMD Opteron\(^\text{TM}\) 48-cores across 4-sockets)

![Linked List 90% Reads](image)

- TXns/second
- Threads
- STM-Fine
- TL2
- CAS

\[^2\text{[Sulzmann et al., 2008]}\]
Workstealing

(AMD Opteron™ 48-cores across 4-sockets)

---

NQueens

STM-Fine
TL2
Original

Threads

Execution Time (seconds)

1 12 24 36 48

NBody

Threads

1 12 24 36 48

---

$^3$[Marlow et al., 2011]
Implementation

Features we must support:

- Atomicity
- Speculation
- retry
- orElse

```haskell
  t = atomically $ do
      x <- readTVar tx
      y <- readTVar ty

      orElse
          (writeTVar tx (x+1) >> when (x == y) retry)
          (writeTVar ty (y+1))
```
Properties

- STM-Fine
  - No Global Bottleneck
  - Lazy Validation

- TL2
  - Global Bottleneck
  - Opacity
t1 = atomically $ do
  x <- readTVar tx
  y <- readTVar ty
  ...
  -- f is big when x > y
  let big = BS.replicate (f x y) 0
  ...

t2 = atomically $ do
  -- invariant: x < y
  modifyTVar ty succ
  modifyTVar tx succ
Future work

- Improve metadata representation for STM-Fine.
- Fix memory effect bugs with STM-Fine.
- Continue to evaluate implementations.
- Explore TL2 with other improvements
  - TStruct
  - HTM
Thanks!
In STM execution, reads and writes to TVars are tracked in a transactional record (TRec).

Execution continues under the assumption that there have been no conflicts. A conflict is where two threads access the same location and at least one is a write.
Existing Haskell STM Implementation

- At the end of the transaction:
  - The RTS \textit{validates} that reads still match the values in the TVars.
  - And \textit{commits} by performing writes atomically.
- If validation fails, start over.
- Similar to OSTM [Fraser, 2004].
  - No global bottlenecks.
  - Read-only transactions do not acquire locks.
  - OSTM is non-blocking, GHC’s STM can livelock.
- A transaction can explicitly block using `retry`.
- Blocking transactions can be composed using `orElse`.
  
  In `orElse a b` the first branch `a` is attempted if it reaches `retry` it is abandoned and execution continues with `b`.
Implementation of retry and orElse

- retry works like an exception caught by atomically or orElse.
  - When atomically is reached, the thread validates, assigns itself to watch lists, and blocks.
  - When orElse is reached, the read set from the first branch is merged, and execution continues with the second branch.
Haskell STM Metadata Structure

Node
- key
- value
- parent
- left
- right
- color

TVar
- value
- watch

Watch Queue
- thread
- next
- prev

Watch Queue
- thread
- next
- prev

TRec
- prev
- index
- tvar
- old
- new
- tvar
- old
- new
- ...
- tvar
- old
- new
Red–Black Tree Example
commit(TRec* trec) {
    if (validate(trec)) {
        if (read_check(trec)) {
            update(trec)
            return true
        }
    }
    return false
}
bool validate(TRec* trec) {
    for (e in trec) {
        if (is_write(e)) {
            if (!lock(e)
                || e->value != e->tvar->value) {
                release_locks(trec)
                return false;
            }
        } else {
            e->version = e->tvar->version
        }
    }
}
bool read_check(TRec* trec) {
    for (e in trec) {
        if (is_read(e)) {
            if (e->value != e->tvar->value || e->version != e->tvar->version) {
                release_locks(trec)
                return false
            }
        }
    }
    return true
}
update(TRec* trec)
{
    for (e in trec) {
        if (is_write(e)) {
            e->tvar->version++
            e->tvar->value = e->new_value
        }
    }
}
bool acquire_locks(write_set) {
    foreach (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) {
    foreach (entry in write_set) {
        entry->tvar->value = entry->value
        entry->tvar->stamp = write_ver
    }
}

bool validate(read_set) {
    foreach (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
}
Transactional locking II.

Practical lock-freedom.
PhD thesis, University of Cambridge Computer Laboratory.

Composable memory transactions.
In Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP), pages 48–60, Chicago, IL.

A monad for deterministic parallelism.
In Proceedings of the 4th ACM Symposium on Haskell, Haskell ’11, pages 71–82, Tokyo, Japan. ACM.

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. ACM.

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. ACM.