Using hardware transactions in the GHC runtime system

Ryan Yates and Michael L. Scott

University of Rochester

Haskell Implementers Workshop, 8-30-2015

---

1This 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.
Outline

- Haskell TM.
- Our implementations using HTM.
- Performance results.
- Future work.

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.
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 conflicting writes.
At the end of the transaction:

- The RTS *validates* that reads still match the values in the TVars.
- And *commits* by performing writes atomically.
- If validation fails, start over.
Existing Haskell STM Implementations

Coarse-grain Lock (STM-Coarse)

- Serialize commits with a global lock.
- Similar to NOrec [Dalessandro et al., 2010, Dalessandro et al., 2011, Riegel et al., 2011].

Fine-grain Locks (STM-Fine)

- Lock for each TVar.
- Two-phase commit.
- Similar to OSTM [Fraser, 2004].
Hardware Transactional Memory

- Available from Intel and IBM.
- Hardware tracks reads and writes and watches for conflicts.
- No guarantee that a transaction will commit, requires a software fallback.
- Intel implementation targets small existing critical sections.
  - Implementation leverages cache-coherence and is limited by the L1-cache.
  - Gives performance of fine-grain locking with the simplicity of a single global lock.
Haskell with HTM

Hardware Commit (HTM-Coarse, HTM-Fine)

- Like coarse-grain and fine-grain lock STMs, but using hardware transactions to perform the commit.
- If the hardware transaction execution repeatedly fails due to conflict, try the commit in software by taking the locks.
- Leverage the TRec for other features: retry and orElse.
Hybrid TM (Hybrid)

- Execute the full Haskell transactions in hardware. Reads and writes are performed directly to TVars.
- If running the full transaction fails due to conflict, fallback to HTM-Coarse.
There’s more!

- Constant space metadata tracking for retry (except for the GC write barrier).
- Lowered implementation level of TM runtime support.
- Mutable unboxed values.
- Remove indirection in mutation.
- Flexible transactional variable granularity.
data Node = Node !(STMMutableArray# RealWorld Node) | Nil

unsafeReadNode :: Node -> Word -> STM Node
unsafeReadNode marr (W# w#) = STM $ \s# ->
    readTArray# (unNode marr) w# s#
{-# INLINE unsafeReadNode #-}

unsafeWriteNode :: Node -> Word -> Node -> STM ()
unsafeWriteNode marr (W# w#) a = STM $ \s# ->
    case writeTArray# (unNode marr) w# a s# of
      s2# -> (# s2#, () #)
{-# INLINE unsafeWriteNode #-}
How fast can we go?
### Machine

- Intel® Xeon™ E5-2699 v3 two socket, 36-core, 72-thread

### Tests

- Red-Black tree with concurrent inserts (5%), deletes (5%), and lookups (90%) measuring throughput at steady state.
Results (50,000 nodes)  
(Intel® Xeon™ E5-2699 v3 two socket, 36-core)
Results (5,000 nodes) (Intel Xeon E5-2699 v3 two socket, 36-core)

Tree operations per second vs Threads

- HashMap
- STM-Coarse
- HTM-Coarse
- Hybrid
- STM-Fine
- HTM-Fine
- Hybrid-TStruct
Results (5,000 nodes) (Intel® Xeon™ E5-2699 v3 two socket, 36-core)

- Threads: 1, 8, 18, 36, 54, 72
- Tree operations per second: 0 to $4 \times 10^7$

Graph showing:
- HashMap
- STM-Coarse
- HTM-Coarse
- Hybrid
- STM-Fine
- HTM-Fine
- Hybrid-TStruct
Results (5,000 nodes) (Intel® Xeon™ E5-2699 v3 two socket, 36-core)

Tree operations per second

HashMap
STM-Coarse
HTM-Coarse
Hybrid
STM-Fine
HTM-Fine
Hybrid-TStruct

Threads

1 8 18 36 54 72
Results (5,000 nodes)

(Intel® Xeon™ E5-2699 v3 two socket, 36-core)
Results (5,000 nodes)

(Intel® Xeon™ E5-2699 v3 two socket, 36-core)

Tree operations per second

- HashMap
- STM-Coarse
- HTM-Coarse
- Hybrid
- STM-Fine
- HTM-Fine
- Hybrid-TStruct

Threads

1 8 18 36 54 72
Results (5,000 nodes) (Intel® Xeon™ E5-2699 v3 two socket, 36-core)

Tree operations per second vs. Threads

- **HashMap**
- **STM-Coarse**
- **HTM-Coarse**
- **Hybrid**
- **STM-Fine**
- **HTM-Fine**
- **Hybrid-TStruct**
Summary

- We have a much better understanding of the performance issues.
- Performance on a concurrent set is competitive.
- **HTM** is a useful and **flexible** tool that helps performance.
- We have a roadmap for future improvements.

Slides: [http://goo.gl/8wVAR5](http://goo.gl/8wVAR5)  
Paper: [http://goo.gl/Er29ef](http://goo.gl/Er29ef)
Experiments with ensuring cache-alignment of STM heap objects has yet to yield improved performance.

Some work with Intel’s emulator (SDE)\(^2\) indicated that memory accesses to the stack were causing false conflicts. Fiddling with stack sizes makes this disappear.

Supporting retry

Existing retry Implementation

- When retry is encountered, add the thread to the watch list of each TVar in the transaction’s TRec.
- When a transaction commits, wake up all transactions in watch lists on TVars it writes.
Supporting retry

Hardware Transactions

- Replace watch lists with bloom filters for read sets.
- Support read-only retry directly in HTM.
- Record write-set during HTM then perform wake-ups after HTM commit.
Wakeup Structure

- Committed writer transactions search blocked thread read-sets in a short transaction eliding a global wakeup lock.
- Committing HTM read-only retry transactions atomically insert themselves in the wakeup structure by writing the global wakeup lock inside the hardware transaction.
  - Releases lock when successfully blocked.
  - Aborts wakeup transaction (short and cheap).
  - Serializes HTM retry transactions (rare anyway).
Results retry

(Intel® Xeon™ E5-2699 v3 two socket, 36-core)
t1 = atomically (a ‘orElse‘ b)

- Atomically choose second branch when the first retry.
- No direct support in hardware for a partial rollback.
- If the first transaction does not write to any TVars, there is nothing to roll back.
  - Keep a TRec while running the first transaction.
  - Or rewrite the first transaction to delay writes until after the choice to retry.
Haskell STM Metadata Structure

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

TVar
- value
- watch

Watch Queue
- thread
- next
- prev

TRec
- prev
- index
- tvar
- old
- new

Watch Queue
- thread
- next
- prev

...
Haskell HTM Metadata Structure

<table>
<thead>
<tr>
<th>Node</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TVar</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>hash</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HTRec</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>read-set</td>
<td></td>
</tr>
<tr>
<td>write-set</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wakeup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>read-set</td>
<td></td>
</tr>
<tr>
<td>thread</td>
<td></td>
</tr>
</tbody>
</table>

...
Haskell Before TStruct

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

TVar
- value
- hash

Node
- key
- value
- parent
- left
- right
- color
Haskell with TStruct

Node
lock
hash
key
value
color
parent
left
right

Node
lock
hash
key
value
color
parent
left
right
Haskell STM TQueue Implementation

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

dehqueue :: TQueue a -> a -> STM ()
dehqueue (TQueue _ write) v = modifyTVar write (v:)

enqueue :: TQueue a -> STM a
enqueue (TQueue read write) =
  readTVar read >>= \case
    (v:vs) -> writeTVar read vs >>= return v
    [] -> reverse <$> readTVar write >>= \case
      [] -> retry
      (v:vs) -> do writeTVar write []
        writeTVar read vs
        return v
Haskell STM Implementation

Fairly standard commit protocol, but missing optimizations from more recent work.

<table>
<thead>
<tr>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Coarse grain: perform writes while holding the global lock.</td>
</tr>
<tr>
<td>- Fine grain:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Broken code that we are not allowed to write!

```haskell
transferBad :: TVar Int -> TVar Int -> Int -> STM ()
transferBad accountX accountY value = do
    x <- readTVar accountX
    y <- readTVar accountY

    writeTVar accountX (x + v)
    writeTVar accountY (y - v)

    if x < 0
        then launchMissiles
    else return ()
```
Broken code that we are not allowed to write!

```haskell
thread :: IO ()
thread = do
    transfer a b 200
    transfer a c 300
```
GHC’s runtime support for STM is written in C.

Code is generated in Cmm and calls into the runtime are essentially foreign calls with significant extra overhead.

We avoid this by writing the HTM support in Cmm.

Typeclass machinery could allow deeper code specialization.
Lazy Evaluation

- Lazy evaluation may lead to false conflicts due to the update step that writes back the fully evaluated value.
- One solution could be to delay performing updates (to shared values) until after a transaction commits.
- Races here are fine as any update must represent the same value.
Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory.

NOrec: Streamlining STM by abolishing ownership records.
In Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP), pages 67–78, Bangalore, India.

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

In Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 53–64, San Jose, CA.