

An Update on Haskell H/STM¹

Ryan Yates and Michael L. Scott

University of Rochester

TRANSACT 10, 6-15-2015

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.



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



Slides: <http://goo.gl/0ZFJXJ> Paper: <http://goo.gl/Er29ef>



At last TRANSACT we reported around 280 open source libraries in the Haskell ecosystem that depend on STM. Now there are over 400.

Reasons for using Haskell STM:

- It is easy!
- Expressive API with `retry` and `orElse`.
- Trivial to build into libraries.

Haskell STM Example

```
transA = do
  v <- dequeue(queue1)
  return v
transB = do
  v <- dequeue(queue2)
  if someCondition(v)
    then return v
    else retry
...
mainLoop = do
  a <- atomically(transA 'orElse' transB 'orElse' ...)
  handleRequest(a)
mainLoop
```



Existing Haskell STM Implementations

- Glasgow Haskell Compiler (GHC), 7.8.
- Explicit transactional variables.
- Object based.
- Lazy value-based validation.

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].



Hybrid TM (Hybrid)

- Three levels for transactions (CF [Matveev and Shavit, 2013]).
 - Full transactions in hardware.
 - Software transaction, commit in hardware.
 - Full software fallback.

HLE Commit (HTM-Coarse, HTM-Fine)

- Like coarse-grain and fine-grain lock STMs, but using hardware transactions to elide locks around commit.



Red-black tree performance

Last year

200 nodes, 84,000 tree operations per second on 4 threads.

Now

50,000 nodes, 24,000,000 tree operations per second on 72 threads.

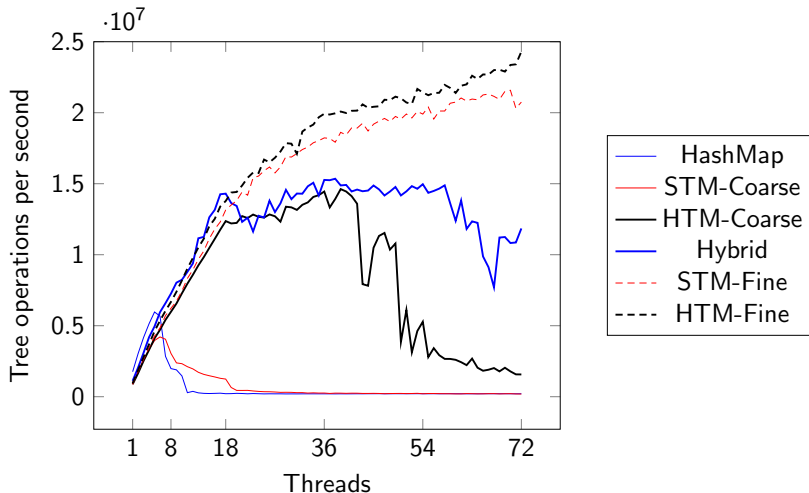


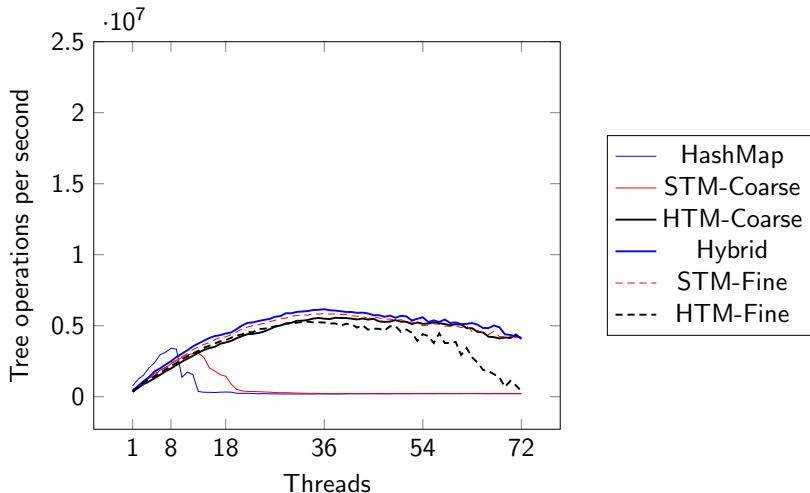
What changed?

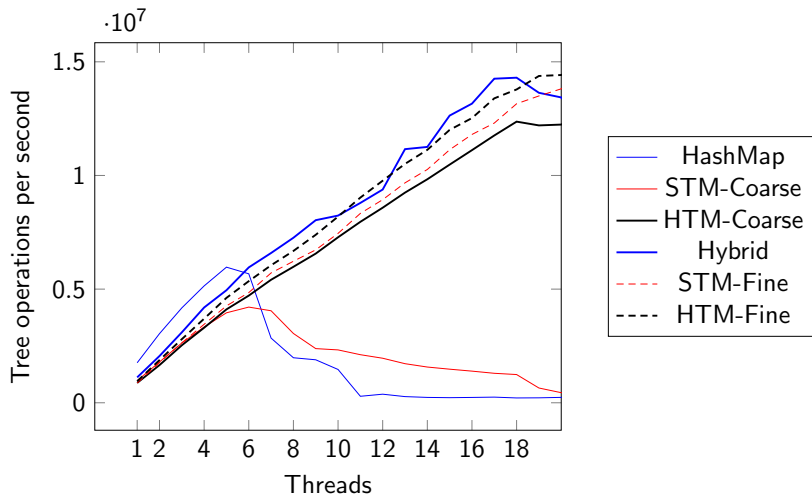
- Constant space² metadata tracking for retry.
- Lowered implementation level of TM runtime.
- Avoid reevaluating expensive thunks.
- Fixed PRNG.

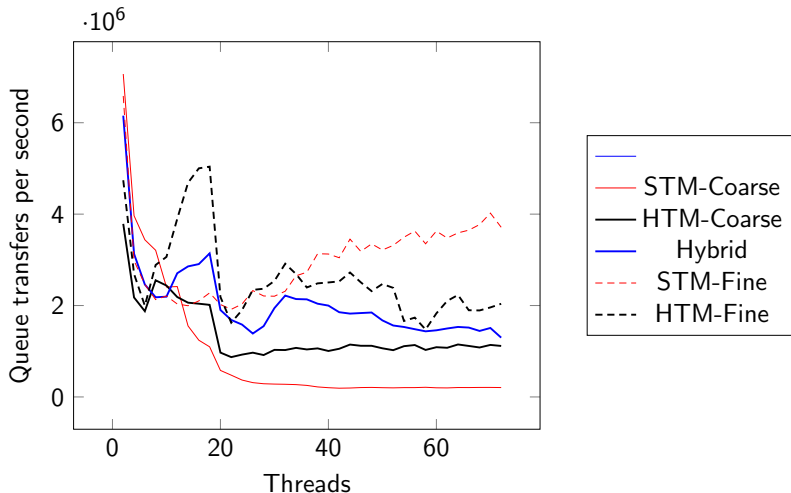
Also improved benchmarking for more accurate measurement.

²Nearly.



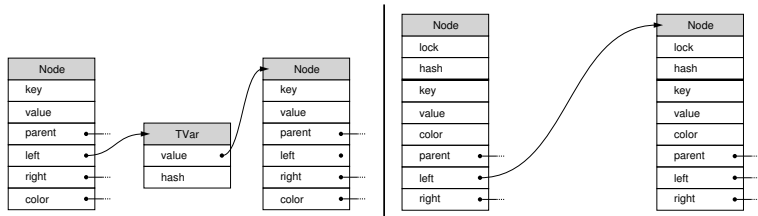






Future Implementation Work TStruct

- Flexible transactional variable granularity.
- Unboxed mutable variables.



Supporting orElse in Hardware Transactions

`t1 = atomically (a 'orElse' b)`

- Atomically choose second branch when the first retries.
- 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.



Summary

- We have a much better understanding of the performance issues.
- Performance on a concurrent set is competitive at scale.
- Good performance for infrequent `retry` use cases.
- HTM is a useful and flexible tool that helps performance.
- We have a roadmap for future improvements.



Slides: <http://goo.gl/OZFJXJ> Paper: <http://goo.gl/Er29ef>





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.



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).



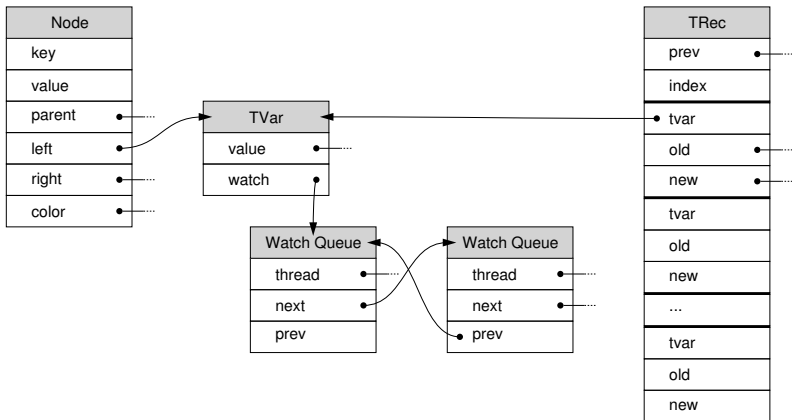
Existing orElse Implementation

Atomic choice between transactions biased toward the first.

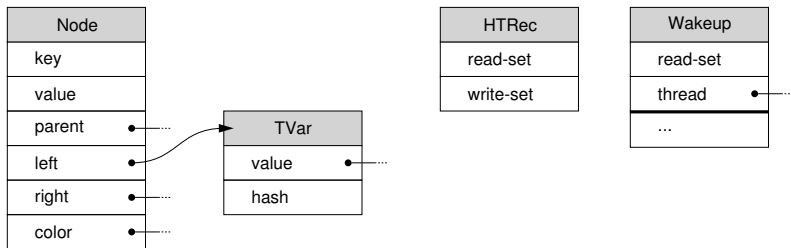
- Nested TRecs allow for partial rollback.
- If the first transaction encounters `retry`, throw away the writes, but merge the reads and move to the second transaction.



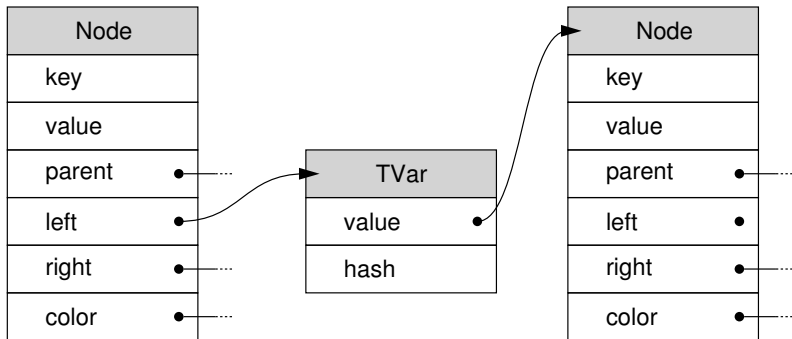
Haskell STM Metadata Structure



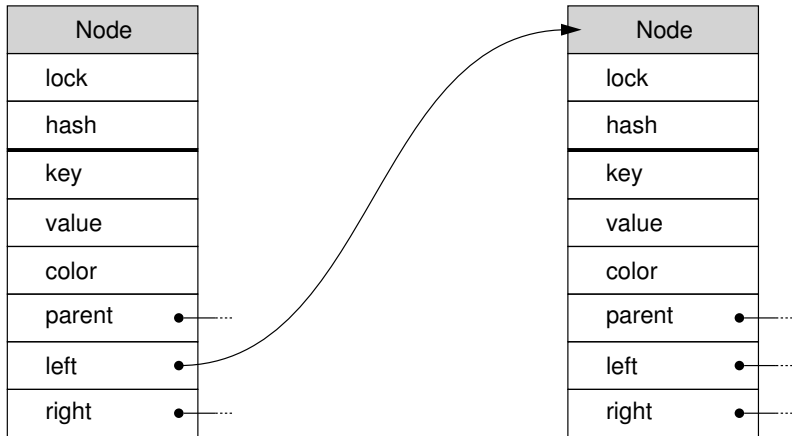
Haskell HTM Metadata Structure



Haskell Before TStruct



Haskell with TStruct



Haskell STM TQueue Implementation

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

```
dequeue :: TQueue a -> a -> STM ()
```

```
dequeue (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
```

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

Commit

- Coarse grain: perform writes while holding the global lock.
- Fine grain:
 - Acquire locks for writes while validating.
 - Check that read-only variables are still valid while holding the write locks.
 - Perform writes and release locks.



Broken code that we are not allowed to write!

```
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 launchMissles
    else return ()
```

Broken code that we are not allowed to write!

```
thread :: IO ()  
thread = do  
    transfer a b 200  
    transfer a c 300
```

C ABI vs Cmm ABI

- 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.



References

- [Dalessandro et al., 2011] Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L., and Spear, M. F. (2011).
Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory.
In Proc. of the 16th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 39–52, Newport Beach, CA.
- [Dalessandro et al., 2010] Dalessandro, L., Spear, M. F., and Scott, M. L. (2010).
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.
- [Fraser, 2004] Fraser, K. (2004).
Practical lock-freedom.
PhD thesis, University of Cambridge Computer Laboratory.
- [Matveev and Shavit, 2013] Matveev, A. and Shavit, N. (2013).
Reduced hardware NOrec.
In 5th Workshop on the Theory of Transactional Memory (WTTM), Jerusalem, Israel.
- [Riegel et al., 2011] Riegel, T., Marlier, P., Nowack, M., Felber, P., and Fetzer, C. (2011).
In Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 53–64, San Jose, CA.

