Implementing Software Transactional Memory Using STM Haskell

Ammlan Ghosh and Rituparna Chaki

Abstract Software transaction memory (STM) is a promising programming abstract for shared variable concurrency. This paper presents a brief description of one of the recently proposed STM and addresses the need of STM implementation. The paper also describes the implementation technique of STM in STM Haskell. In the STM implementation process, three different approaches have been presented which employ different execution policies. In the evaluation process, transactions with varying execution length are being considered which are executed in multi-threaded environment. The experimental results show an interesting outcome which focuses on the future direction of research for STM implementation.

Keywords Software transactional memory (STM) · Haskell · Concurrency

1 Introduction

Software Transactional memory (STM) [1] is a promising approach for concurrency control in multi-core processors environment. A transaction in STM executes a series of reads and writes to shared memory, which is grouped into an atomic action. STM guarantees that every action will appear to be executed atomically to the rest of the system.

There are several STM approaches those have worked on basic concurrency implementation for avoiding deadlock. These approaches use blocking [2–4] or non-blocking [5–8] process synchronization technique. In non-blocking process synchronization, the major challenge is reducing abort of concurrently executing transactions. A limited of works have been done in this area [7, 8]. In [7], aborting of

A. Ghosh (✉) · R. Chaki
University of Calcutta, Kolkata, India
e-mail: ammlan.ghosh@gmail.com

R. Chaki
e-mail: rchaki@ieee.org

© Springer India 2016

R. Chaki et al. (eds.), *Advanced Computing and Systems for Security*,
Advances in Intelligent Systems and Computing 396,
DOI 10.1007/978-81-322-2653-6_16

transaction has been identified as a major limitation for STM solutions. The work in [7] is on abort- free execution for a cascade of transactions. Although, theoretical estimation shows a good performance improvement; however, the actual STM implementation was not being done to explore the actual performance improvement.

Some STM solutions explore software engineering aspects either by using realistic concurrent data [9, 10, 11] or by a theoretical study [12]. One of the major breakthroughs is the implementation of composable software transactional memory [10, 11] in Haskell.

STM Haskell [10] provides composable memory transaction, i.e., transactional actions that are defined can be combined to generate a new transaction. STM Haskell takes an action as its argument and performs it atomically by maintaining two guarantees: Atomicity and Isolation. Atomicity ensures that execution of a transaction is visible to other threads all at once. Isolation property guarantees that execution of a transaction cannot be affected by other transactions. Since its introduction, several extensions to the basic primitives have been proposed in STM Haskell. This makes STM Haskell more flexible and easy customizable implementation.

This paper describes an implementation of software transactional memory using STM Haskell, using three different concurrency control mechanisms and compares their performance.

The paper is organized as follows: Section 2 presents a brief description of one of the recently proposed STM solution [7] that has claimed to improve throughput in all possible scenarios. This section follows a critical observation on the said work [7] and its analysis to establish the importance of implementing STM solutions on a suitable platform towards appropriate performance analysis. Section 3 describes the implementation technique of software transactional memory using STM Haskell. Section 4 explores the performance and presents the result set. We have presented a set of observations on the advantages of STM Haskell towards implementing STM solutions. The paper concludes in Sect. 5 with a note on future direction of research for STM implementation.

2 Retrospection of an OFTM Solution Towards Abort Freedom

In [7], an interesting obstruction free implementation of STM was proposed that allows contentious transactions to execute without causing any abort to other transactions. The basic idea of this algorithm is that, a transaction, say T_k, may be in active state even after the completion of update process of a transactional variable. Now if another transaction, say T_m, wants to access the same transactional variable, it faces contention with T_k. Thus, in conventional method, either T_m will be blocked or T_m will abort T_k to get access of that transactional variable. In contrast, this algorithm [7] allows T_m to access the transactional variable optimistically, with an

expectation that T_k will not update that transactional variable further, thus T_m will find a consistent data value at commit time. At commit time T_m will check the data consistency, i.e., transactional value at the start time is same as at the time of its commit. If data is consistent and T_k is committed, then T_m commits; otherwise, T_m re-executes its operation after reading the last updated value of the transactional variable.

The paper elaborately explains how to execute read and write operations for two transactions in presence of contention. This procedure is also extensible for a cascade of transactions. The efficiency and performance improvement is compared with DSTM (STM for Dynamic-sized Data Structures) [5] in terms of the average execution time (AET) of the transactions. Three different cases are being considered: Where AET of two transactions are equivalent; AET of first transaction less than the second transaction, and lastly, AET of first transaction is greater than the second transaction. The result set shows the throughput of the algorithm is better or at least equivalent to the DSTM [5]. In spite of having several potentials, the algorithm in [7] suffers from some serious drawbacks.

- the solution [7] does not ensure isolation property as transactions communicate between themselves and share the non-committed transactional data;
- the paper [7] proposes abort-free execution, which is tailored only for two concurrent transactions. It has given only an idea on how cascade of transaction may run without any abort;
- the algorithm [7] claims to execute in abort-free manner. However, in some specific cases, transaction either aborts its enemy transaction or backs-off for some arbitrary time;
- authors of [7] claimed that the approach yields higher throughput in comparison to DSTM [5]. However, the actual STM implementation is not done.

These drawbacks can be actually verified and analyzed by implementation or at least by some proper simulation of STM. The GHC STM Haskell could be one of the suitable platforms for STM implementation due to the following reasons:

- GHC Haskell implements some major extensions to support both concurrent and parallel programming, which is highly desirable in multi-core processor environment;
- No new language construct has been introduced in concurrent Haskell, rather it appears as libraries. The functions are exported by these libraries;
- In Haskell, the STM library includes various features like Atomic blocks, Transactional Variables and more importantly the composability of transactions.

All these features make STM Haskell a promising language construct for STM implementation. The algorithm presented in [7] has encouraged the authors of the current paper to discuss on implementing STM in Haskell. In next section, the implementation technique of STM using STM Haskell has been described with different concurrency control mechanism available in GHC Haskell.

3 Implementing STM Using STM Haskell

3.1 Important System Variables

The STM Haskell uses a monad to encapsulate all access operation to shared transactional variables (TVars). The operations in TVars are as follows:

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

Both `readTvar` and `writeTVar` operations return STM actions, which can be composed by **`do {}`** syntax. STM actions are executed by a function atomically, with type `atomically :: STM a -> IO a.`

This function takes memory transaction and delivers I/O action. It runs the transaction atomically with respect to all other transactions. An STM expression can also retry to deal with blocking, when a transaction has to wait for some conditions to be true.

```
retry :: STM a
```

The semantics of retry is to abort the current transaction and run it again. But, instead of blindly rerunning the transaction again and again, transaction reruns only when the TVar that has been read by the current transaction has changed.

Finally, the orElse function allows two transactions to be combined, where only one transaction is performed but not both.

```
orElse :: STM a -> STM a -> STM a
```

The operation `orElse` T_1 T_2 has the following behavior:

- First T_1 is executed, if it returns result then `orElse` function returns.
- If T_1 retry instead then T_1 is discarded and T_2 is executed.

3.2 Implementation

To implement STM in Haskell, we have chosen three different implementation approaches to execute a specific task. The task is to read a sharable data object, calculate the Fibonacci value, and finally write that Fibonacci value to the sharable data object. The whole job will be executed with the protection of atomically.

The first approach uses `TVars` of STM Haskell. The atomically function of STM Haskell maintains a per-thread log that records the tentative access made to `TVars`. Whenever atomically is invoked, it checks whether log is valid, i.e., no concurrent

transactions has committed conflicting updates. If the log is valid then transaction commits; otherwise, transaction re-executes with a fresh log.

The next two approaches use `TMVars`. In Haskell; `MVars`, Mutable Variables, can be either empty or full. When a value is tired to remove from an empty `MVar` or to put a value into a full `MVar`, the operation is being blocked. `TVar` is modeled with `MVar` that contains `Maybe a`, i.e. `newtype TMVar a = TMVar (TVar (Maybe a))`. '`Maybe a`' is very common data type used in Haskell, where a function may or may not succeed. This data type is as follows:

```
Data Maybe a = Nothing
           | Just a
```

The `TMVar` implementation is included in the `Control.Concurrent.STM.TMVar` module of STM package in Haskell.

The second approach uses `TMVar` to execute as per the shortest job first process implementation. The third approach also uses `TMVar` to execute transactions sequentially in a first-in-first-out basis.

The first approach uses non-blocking synchronization, where as other two use blocking methodology of STM. All these three implementations guarantee atomicity and isolation properties of STM.

Finding Fibonacci Value

Haskell's Control.Parallel module provides a mechanism to allow users to control the granularity of parallelism. The interface is shown below:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

The function `par` evaluates the first argument in parallel with the second argument by returning its result to the second argument. The function `pseq` specifies which work of the main thread is to be executed first. The expression `a pseq b` evaluates `a` and then returns `b`. An elaborated explanation on Haskell parallelism is discussed in [13, 14].

While calculating Fibonacci value, the `par` and `pseq` monad is used to gain parallelism. The code is as follows:

```
nFib :: Int -> Int
nFib n | n <= 2      = 1
       | otherwise   = par n1 (pseq n2 (n1 + n2 ))
                 where n1 = nFib (n-1)
                       n2 = nFib (n-2)
```

Achieving Concurrency in Haskell

Haskell provides explicit concurrency features via a collection of library functions. The module `Control.Concurrent` provides an abstract type `ThreadId` to identify the Haskell thread. A new thread is created in the `IO` monad by calling the `forkIO` function, which returns `IO` unit.

```
forkIO :: IO () -> IO ThreadId
```

At the time of execution, while using TVars, the main thread in Haskell does not wait for its children threads to complete. The `mapConcurrently` function has been used to overcome this problem. This function is provided by Haskell's `Control.Concurrent.Async` module. The function `mapConcurrently` ensures that main thread does not quit till all its children threads complete their operations. The detailed explanation about this module is available in [14].

STM implementation using TVar

Function transTest is created to define the task of the transaction. The block of code is as follows:

```
transTest :: TInt -> Int -> IO ()
transTest n t = do
  atomically $ do
      let x = nFib t
      writeTVar n x
```

`TInt` is an integer type Transactional Variable. The type is defined as

```
type TInt = TVar Int
```

The function transTest has two parameters, a TVar and an integer. It calculates Fibonacci value of the given integer and writes that value to the TVar. Calculation of Fibonacci value determines the execution time of the transaction. As nFib 40 takes much more time than nFib 20, thus execution time in the prior case will be higher.

The code for function main (), is as follows:

```
main :: IO ()
main = do
  n <- newTVarIO 0
  _ <- mapConcurrently (transTest n) [40, 20]
```

This code executes two transactions concurrently, where first one will write Fibonacci value of 40 to the TVar *n* and the second one will write Fibonacci value of 20. Now question is that how the Haskell STM will execute these two transactions. As the execution time of the second transaction is less, it completes its execution earlier than first one and finds a valid log value, thus commits. As a result, first transaction will get invalid value in its per-thread log and thus it will re-execute its operation with a fresh log value.

Now suppose we want to track the commit pattern of the transactions. To do so, a list of MVar data type is to be created, where the `threadIds` will be stored when transactions successfully commit. The modified code is as follows:

```
type TInt = TVar Int

transTest::MVar[(ThreadId, Int)] -> TInt -> Int -> IO ()
transTest mvar n t = do
  tid <- myThreadId
  atomically $ do
      let x = nFib t
      writeTVar n x
  list <- takeMVar mvar
  t2 <- atomically $ readTVar n
    putMVar mvar $ list ++ [(tid, t2)]
main :: IO () -- Asynchronous Thread
main = do
  n <- newTVarIO 0
  ms <- newEmptyMVar
  putMVar ms []

  _ <- mapConcurrently (transTest ms n) [40, 20]
  mms <- takeMVar ms
  putStrLn (show mms)
```

Steps for program compilation

The command to compile the program [13] in multi-threaded environment is as follows:

```
$ ghc –o testTVar --make testTVar.hs –threaded –rtsopts
```

To execute the program, we need to specify how many real threads are available to execute the logical threads in the Haskell program. The command to execute the program with two real threads is:

```
$ ./testTVar +RTS –N2 –s
```

The flag *–s*, if included, shows the actual evaluation thread executions. The portion of the actual output, while executing with two threads, is as follows:

```
[(ThreadId 6,6765),(ThreadId 4,102334155)]
  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    7.25s  (  3.63s elapsed)
  GC      time    1.00s  (  0.50s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    8.25s  (  4.13s elapsed)
Alloc rate    3,611,846,709 bytes per MUT second
Productivity  87.9% of total user, 175.6% of total elapsed
```

The output shows that commit pattern of the transactions. The execution time is 4.13 s against actual 8.25 s. It also shows the 175.6 % productivity.

STM Implementation using TMVar (*Shortest Job First*)

In our attempt to implement this, we have used TMVar and threads together. We have created an empty TMVar and forked the job to run in the background. The main thread has been blocked until each results return. While calling Fibonacci, `BangPatterns` [15] is used to evaluate the Fibonacci value, so that at the time of execution, first thread to finish will have its result first.

We have taken the advantage of TMVar's empty/full semantics to block the main thread for each of the children threads. The program code is given below. The function nFib is same as above. In this implementation also, transactions run atomically and obey the basic principles of STM.

```
{-# LANGUAGE BangPatterns #-}
main :: IO ()
main = do
      result <- newEmptyTMVarIO
      forkIO $ do
        atomically $ do
          let !x = nFib 40
          putTMVar result x
      forkIO $ do
        atomically $ do
          let !x = nFib 20
          putTMVar result x

      t <- atomically $ takeTMVar result
      putStrLn ("Fastest job is: " ++ (show t))

      t <- atomically $ takeTMVar result
      putStrLn ( "Slowest job is: " ++ (show t))
```

STM Implementation using TMVar (*First-In-First-Out*)

The implementation is same as the previous one, but in this case transactions execute in first-in-first-out basis. Here, the second transaction waits till the first one completes its execution. In this approach, transaction variables are also accessed atomically.

```
main :: IO ()
main = do
          result <- newEmptyTMVarIO
    forkIO $ do
      atomically $ do
        let x = nFib 40
        putTMVar result x

    t <- atomically $ takeTMVar result
          putStrLn ("First value: " ++ (show t))

    forkIO $ do
      atomically $ do
        let x = nFib 20
        putTMVar result x

    t <- atomically $ takeTMVar result
          putStrLn ( "Second value: " ++ (show t))
```

In the next section, these three approaches are being implemented in multi-threaded environment to analyze result set.

4 Simulation Results

In this experiment, parallelism and concurrency both are taken care of while implementing Software Transactional Memory in STM Haskell. This case study considers that transactions perform ‘some task’, which can be executed in parallel and update the transactional variables. The whole task is to be executed atomically, i.e., either all at once or none. The execution length of transaction depends on the execution time of the task. Thus, throughput of the concurrent execution of transaction also depends on the efficiency of the parallel and concurrent execution of the task.

In this case study three different approaches, as stated in Sect. 3.2, are being considered. The first one is STM Haskell by using TVar, second one (SJF) uses TMVar while execution shortest job first, and third one (FIFO) also uses TMVar but execution pattern is in first-in-first-out basis. The performance of these implementations varies due to these execution policies although all of them ensures STM properties.

The experimental results are summarized by varying execution length of the transaction. To do so, a set of transactions with different execution length are being considered while they are executing concurrently and sharing a common resource. Each set of transaction is formed up with five write transactions. Depending on the AET, transactions are segregated into three groups. In the first group, the AET of

transactions is comparatively lower. In the second group, AET is comparatively medium and in third group the AET of transactions is comparatively higher.

In order to investigate scalability, the said three approaches are being executed on these three different groups of transactions. While executing the program, the number of threads is varied from 1 to 5 to observe the efficiency of each method in terms of parallel and concurrent execution.

This implementation is performed on Intel Core i7, 64 bit processor with 8 GB memory, and 2 MB L2 cache, running on Linux and GHC 7.8.3.

4.1 Case-I: Lower Average Execution Time

When transactions have lower execution time, SJF performs best up to three threads. Although, with a higher number of threads, STM Haskell has the slightly better throughput. Table 1 and Fig. 1 show these scenarios.

4.2 Case-II: Medium Average Execution Time

In the case of medium length transactions, performance varies with number of threads, same way as stated in previous case, i.e., with higher number of threads

Table 1 Performance of the said approaches with lower average execution time

	#Thread 1	#Thread 2	#Thread 3	#Thread 4	#Thread 5
STM Haskell	3.92	2.14	1.52	1.31	1.16
TMVar SJF	3.78	2.05	1.45	1.34	1.14
TMVar FIFO	4.01	2.15	1.60	1.42	1.33

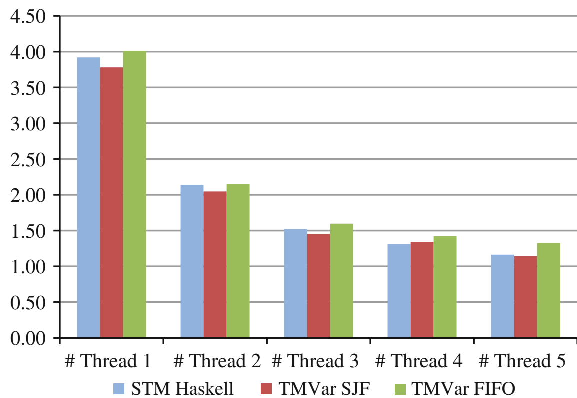

Fig. 1 Performance graph of the said approaches with lower average execution time

Table 2 Performance of the said approaches with medium average execution time

	#Thread 1	#Thread 2	#Thread 3	#Thread 4	#Thread 5
STM Haskell	8.57	4.66	3.30	2.96	2.62
TMVar SJF	8.36	4.55	3.23	3.19	2.72
TMVar FIFO	9.13	5.00	3.57	3.27	3.12

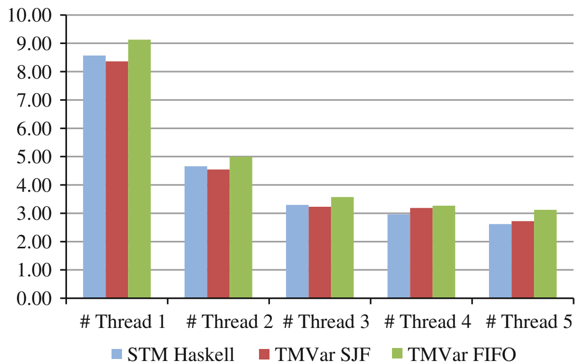

Fig. 2 Performance graph of the said approaches with medium average execution time

STM Haskell performs better. Table 2 shows the result and Fig. 2 depicts the performance graph.

4.3 Case-III: Higher Average Execution Time

When transactions are too lengthy, STM Haskell outperforms others, except in single-threaded execution. The result set and corresponding graph are shown in Table 3 and Fig. 3 respectively.

4.4 Productivity Improvement with Parallel Execution

Figure 4 shows the average productivity improvement in elapsed time while executing transaction with a varying number of threads. Higher number of threads shows higher productivity.

Table 3 Performance of the said approaches with higher average execution time

	#Thread 1	#Thread 2	#Thread 3	#Thread 4	#Thread 5
STM Haskell	29.46	9.30	11.06	9.69	8.70
TMVar SJF	29.04	15.88	11.13	10.94	9.03
TMVar FIFO	31.35	17.64	13.05	11.19	9.67

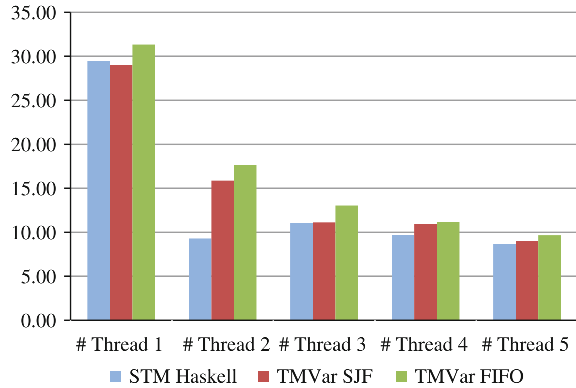

Fig. 3 Performance graph of the said approaches with higher average execution time

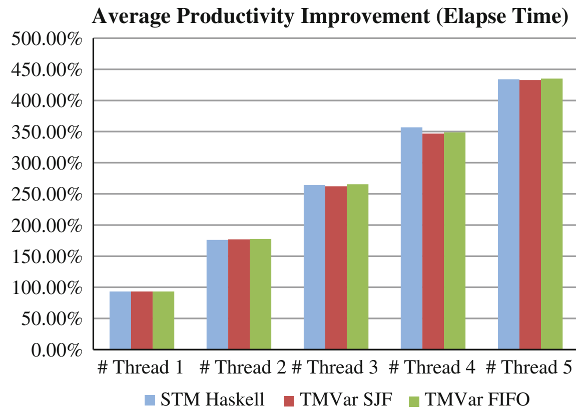

Fig. 4 Productivity improvement on elapse time with increasing number of threads

4.5 Summary of Results

The shortest job execution policy has minimum waiting time, which implies low turnaround time for processes. For this reason, in single-threaded environment, our implementation performs better with shortest job first execution policy. In multi-threaded environment, the parallel activities, i.e., scheduling the job for multi-cores, switching between threads etc. are managed by Haskell compiler and OS. Under this scenario, our STM implementation with TVar performs better than other two approaches. When transactions' execution length is higher, this approach performs best while executing in multi-threaded environment. In our third implementation, where transactions execute in first-in-first-out basis, transactions' average waiting time becomes higher, which results in high turnaround time and a lower throughput.

5 Conclusions

In this paper, we have critically described one of the recently proposed STM solutions to establish the importance of STM implementation for appropriate performance analysis. We have also implemented STM in Haskell using three different approaches. The first implementation uses TVars (STM Haskell) to access transactional variables concurrently. Each transaction maintains a log and depending on its validity transaction re-executes with a fresh log. In second implementation, we have combined TMVar and bang-pattern for strict evaluation, which enables transactions to execute any job in the background. Using this technique, we implemented shortest job first execution policy. The third implementation executes transactions as per their initiation order in first-in-first-out basis.

In all these implementations, we have executed task of the transactions in parallel and observed the performance impact of different execution policies. The experimental results show variations in performance depending on number of threads and transactions' execution length. Transactions with smaller execution length perform better in shortest job first implementation when number of threads is less. When number of threads is increased, the STM Haskell performs better. On the other hand, when transaction execution length is high, STM Haskell performs better, irrespective of number of threads available.

References

1. Shavit, N., Touitou, D.: Software transactional memory. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 204–213. ACM (1995)
2. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 388–402. ACM Press (2003)
3. Gidenstam, A., Papatriantafilou, M.: LFthreads: a lock-free thread library. In: Principles of Distributed System, pp. 217–231. Springer, Berlin (2007)
4. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: ACM SIGPLAN Notices, vol. 46, no. 8, pp. 179–188. ACM (2011)
5. Herlihy, M.P., Luchangco, V., Moir, M., Scherer, W.M.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (2003)
6. Tabba, F., Wang, C., Goodman, J.R., Moir, M.: NZTM: non-blocking zero-indirection transactional memory. In: Proceedings of the 21st ACM Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 204–213. ACS (2009)
7. Ghosh, A., Chaki, N.: The new OFTM algorithm toward abort-free execution. In: Proceedings of the 9th International Conference on Distributed Computing and Information Technology, pp. 255–266. Springer, Berlin (2013)
8. Dolev, S., Fatourou, P., Kosmas, E.: Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions, Preprint, TRANSACT'13 (2013)
9. Discolo, A., Harris, T., Marlow, S., Jones, S.P., Singh, S.: Lock free data structures using STM in Haskell. In: Functional and Logic Programming, pp. 65–80. Springer, Berlin (2006)

10. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005. ACM Press, New York (2005)
11. Du Bois, A.R.: An implementation of composable memory transactions in Haskell. In: Software Composition, pp. 34–50. Springer, Berlin (2011)
12. Borgström, J., Bhargavan, K., Gordon, A.D.: A compositional theory for STM Haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, pp. 69–80. ACM (2009)
13. Jones, S.P., Singh, S.: A tutorial on parallel and concurrent programming in Haskell. In: Advanced Functional Programming, pp. 267–305. Springer, Berlin (2009)
14. Marlow, S.: Parallel and Concurrent Programming in Haskell, 1st, edn. O'Reilly Media, Inc. (2013)
15. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell, 1st edn. O'Reilly Media, Inc. (2008)