Recent Trends in STM Haskell

Priyanka Tiwari[1], Smt.Meenu[2]

[1]*Department of Computer Science & Engg.*
Madan Mohan Malaviya University of Technology Gorakhpur, India

[2] *Asst.Professor, Department of Computer Science & Engineering,*
Madan Mohan Malaviya University of Technology Gorakhpur, India

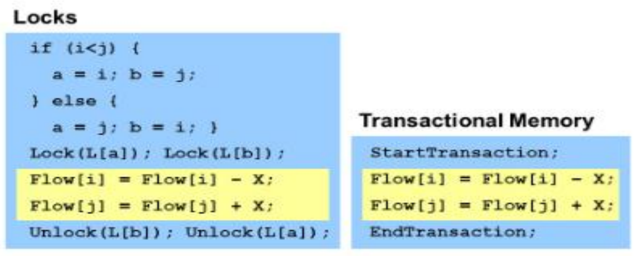
---***---

Abstract - *Software Transactional Memory (STM) is an encouraging programming concept in case of the shared variable.STM expansion to Haskell supplied a simpler way of applying the lock-free method in concurrent programming, employing atomically composed block functioning on the transactional variable. As Haskell consist of an abundant collection of synchronization primitives for developing shared state concurrency concepts, starting from greater level (STM) to lower level (mutual variable using atomic read-alter-write). In this review, we discussed transactional memory basics and all its three approach namely HTM, STM & HyTM.Along with this main focus is made on the software approach of TM and its various implementation through Haskell till now. This paper also state methods and software used by several authors for STM implemenatation. Issues and challenges of STM is also added in the end.*

Key Words: **STM Haskell, TVar, STM, HTM, HyTM, Locks**

1. INTRODUCTION

For STM, various modern systems have been developed in recent years. Attention toward this system is more as hardware sellers have mostly deserted the search for high speed uniprocessor as writing good lock-based code is difficult. Also, expanding of coarse grain locking is not possible plus fine grain locks algorithm based algorithm is notably hard to make. In contrast to locks, transaction averts several essential problems such as priority inversion, deadlock, and sensitivity to thread non-success also the execution difficulty of lock convoying plus preemption and page fault sensitivity. Perchance the greatest essential thing is that they release programmers from making a sad choice amidst concurrency and theoretical transparency: transaction integrates the directness of individual coarse grain lock with extreme contention execution of fine-grain locks. In figure.1 [37] shows the difference between locks and Transactional Memory in terms of work.



```
if (i<j) {
  a = i; b = j;
} else {
  a = j; b = i; }
Lock(L[a]); Lock(L[b]);
Flow[i] = Flow[i] - X;
Flow[j] = Flow[j] + X;
Unlock(L[b]); Unlock(L[a]);
```

```
StartTransaction;
Flow[i] = Flow[i] - X;
Flow[j] = Flow[j] + X;
EndTransaction;
```

Fig -1: Comparison between Lock and Transactional memory (source [37])

Initially, Herlihy and Moss proffered the hardware mechanism [1], transactional memory(TM) takes the concept of atomicity, consistency, and isolation from database transaction. With TM [2], shared location can be concurrently accessed by multiple threads in an atomic method, therefore every access made by a individual thread either succeed or none, inside an atomic transaction. In the case of two mutually conflicting transactions, one will terminate and restart automatically. The capability to terminate the transaction removes the complication present in fine grain locking. The capability to accomplishing (non conflicting) transaction concurrently results in feasibly high performance. Current TM systems can be developed in hardware, in software, or applying a mixture of both hardware and software. Hardware transactional memory (HTM) in 1993 first designed by Herlihy and Moss which was based on a modification of standard cache coherence protocol. In 1995 Shavit and Touitou [3] gave STM to address the intrinsic limitation of HTM, like the absence of commodity hardware along with suggested feature and an insufficient number of locations which a transaction can approach. Apart from above two approaches, vigorous research on Hybrid transactional memory (Hytm) [4] is going on that uses Hardware transactional and switch to STM as the hardware resource requirement exceed. All the three approaches of Transactional memory and their description given above are shown in figure 2.

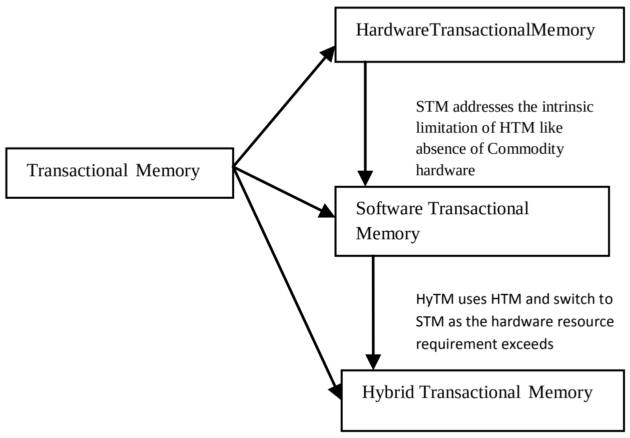

HardwareTransactionalMemory
STM addresses the intrinsic limitation of HTM like absence of Commodity hardware
Transactional Memory
Software Transactional Memory
HyTM uses HTM and switch to STM as the hardware resource requirement exceeds
Hybrid Transactional Memory

Fig -2: Three different implementation approach of Transactional memory

We mainly concentrated here on software. As STM is relevant for today's machines movability, and resiliency in the face of timing irregularity and processor collapse. STM has very eminent in Haskell. Haskell's package archive presently contains 500 such packages which can be used by STM without any intermediary. There is a widespread real-world application on STM Haskell. In this survey paper, we addressed some of the STM Haskell based work followed by deep analysis and comparison among various work focused here. In section 2, we introduced the general idea of STM and its variants that have been proposed till now. In section 3, we discussed the STM Haskell design and its semantics. In section 4 we overviewed various transactional memory implementations using STM Haskell. Finally, we conclude with a statement on upcoming direction work on STM Haskell.

1.1 STM Haskell Background

The GHC inherently consist of STM functions in concurrent Haskell library [16], supporting abstraction for communication among explicitly fork threads. Harris et al. assert in his paper [17], that STM can be conveyed exquisitely within a declarative language. Also, Haskell type system (especially monadic operation) compels threads to access approach shared variable solely within the transaction. Even though the crux of the language is dissimilar to languages such as C++ or C#, the real STM operation is applied in a uncomplicated command form as well as STM application applies the same technique applied in the primary languages.

STM Haskell includes following benefits (i) runtime system is tiny, which makes it easier to make experimental changes. (ii) Number of example application has been developed using STM support and transactions: certainly leading to the application which has been written by "common" programmers instead of those who built the STM-Haskell. STM gives secure way of accessing shared variable amidst simultaneously running threads by the application of monads [18]. Containing I/O action within the I/O monad as well as STM action within the STM monad. Only STM action and pure computation can be conducted inside a memory transaction using different STM & I/O action, though outside the transaction only I/O action plus pure computation can be performed. This ensures that outside the protection of atomically TVars cannot be altered. This type of protection is called "strong atomicity"[19]. Furthermore, computation with side effects and computation that are impact free are totally detached due to monads and Haskell background. Making use of declarative language in TM helps it to provide specific read/writes from/to changeable/variable (mutable) cell: Needlessly STM doesn't try to track memory operation that is implemented through functional computation. because they are never required to roll back[17].In STM Haskell threads interact by reading and writing transactional variable or TVars.STM monad is used by all STM operation.STM monads provide a series of transactions operation, counting, allotting, writing and reading transactional variables, especially functions that are shown below in figure 3[25].

STM OPERATION	**Atomically :: STM a-> IO a** **retry :: STM a** **or Else :: STM a-> STM a-> STM a**
TRANSACTIONAL VARIABLE	**data TVar a** **newTVar:: a-> STM(TVar a)** **readTVar::TVar a-> STM a** **writeTVar::TVar a-> a-> STM()**

Fig -3: STM Operations and Transactional Variable

In Haskell transactions are initiated inside the IO monad with the help of atomically construct. When a transaction completes, it is approved with the help of runtime system that the transaction was carried out upon a consistent system state plus no further completed transaction may have altered the appropriate portion of the system state at the same time[12].In such situation, the alteration of the transaction are committed or else they are aborted. Record of acquired transactional variables in every transaction is maintained by Haskell STM runtime. The written variable in the record is known as "writeset" and every variable that was read is known as "readset" in transaction. It is important to note that above two sets can overlap. Atomically take the temporary update as well as employ the updates to TVars used in the operation which makes these effects seeable to other transaction. This mechanism for each thread keep transaction log which store the temporary acquiring done on TVars.When atomically is initiated, the STM runtime examines that these accesses are authentic as well as no other conflicting updates have been committed by a concurrent transaction. If the validation becomes successful, then the alteration is committed collectively to the heap.

2. STM AND ITS VARIANTS

The thorough analysis of software transactional memory (STM) started in 1995 when Shavit & Touitou used the word "STM" first. Shared memory words were used as concurrent objects in Shavit & Touitou model. Only one transaction at a given time will be permitted to hold the shared memory word. The shared memory word ownership information is kept in another equivalent word known as ownership record. Every memory word is having different connected ownership record. The ownership record stores either a null value (indicating that equivalent shared memory word is not owned by any other transaction currently) or a instance of its owner 'transaction record. A transaction record is defined as a data structure which keeps information regarding equivalent transaction's STM accesses. A transaction holds one transaction record at a given time. Each and every transaction is given shared access to every present transaction record. Yet transaction record is held by just one transaction.

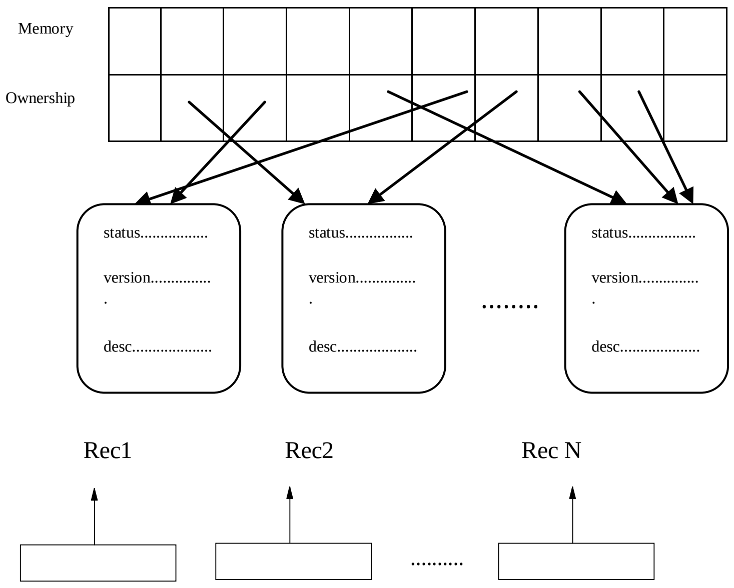


Fig -4: STM Implementation: shared data structure (source [3])

Figure 4 gives the clear-cut idea of above explaination. If a transaction miss to achieve an ownership (since the memory location is held by any other transaction at the same time), the transaction let go previously acquired ownership after aborting. If the transaction gains all the acquired ownership, it makes its state atomically as Committed do the updates and discharge the achieved ownership.
In [1] they focused on the fundamental case such transaction that is non-nesting and that do updates on shared memory inside single multithreaded operation, concentrating on the primary issues that STM must deal. A greater level difference between STM application is the way it arranges data inside memory. One method isolates transactional data as well as regular data, presenting a well-defined memory scheme for transactional objects. Another method permits data to hold its regular form inside memory, and STM applies a distinct data structure for managing its own metadata. The examples of very first kind of STM designs are DSTM [5],Adaptive STM(ASTM)[6],Object-based STM(OSTM)[7]. In the group of developed STM, DSTM [5] WSTM [8], ASTM [6] and unessential SXM[9] are obstruction]free.DSTM[11] given by Herlihy is a feasible obstruction free STM. For isolating the problems of progress and correctness for given data structure DSTM depends on contention manager unit. ASTM [6] like DSTM applies the same contention management junction. WSTM [8] and unessential McRT-STM [10] are termed as "word-based" or more commonly may be termed as "block based": they find conflicts and imposes consistency on a stable chunk of memory free of data semantics of high level. Tiny STM [11] is another word based STM approach which applies locks to prevent shared memory area. Tiny STM applies word-based version of LSA algorithm [12] that like TL2 algorithm [13]. Tiny STM shares various feature with the alternative word-based STM.
Tiny STM applies encounter time locking. Like TL2 and LSA, Tiny STM is a time based approach that assures that transaction will read the consistent state of memory. Memory access is permitted by word-based STM at word granularity and may be applied to the not controlled environment. McRT-STM uses two-phase strict locking [14] for its implementation. McRT-STM consists of blocking commit and abort sequences which makes effective implementation, and further permits McRT-STM to develop various design alternatives. RSTM [15] for accessing data objects applies a single level of indirection. RSTM stay away from dynamic allocation or group of each object or each transaction metadata. It also averts determine or reference totaling garbage collection entirely. RSTM supports various choices for conflict observation and management of contention. RSTM is based on C++, permitting its API to apply inheritance and template.

3. STM IMPLEMENTATIONS TILL NOW

Shavit and Touitou [3] in 1997 introduced STM (Software Transactional Memory) as a model that provides peliable transactional programming for concurrent operations in software.Prior to [1] many related construct[20,21,22,23,24] were suggested. Maurice Herlihy and Victor Lunchango [5] in 2003 given the very first dynamic STM that permits transaction and its object to be built in dynamically. In DSTM, transaction can discover the order of object to approach on the basis of value examined in objects approached previously in the same transaction. Earlier STM model needed predetermined space & fixed transaction. In 2005 Tim Harris & Simon Marlow [17] re-expressed the concept of transactional memory in the context of concurrent Haskell which provides significantly powerful assurance than the traditional system. Two new functions were also presented known as retry and or Else. The retry function is capable of occurring anywhere inside the transaction, preventing it till we get another execution path. Apart from this orElse permits transactions to be composed as alternatives which help transaction module to run the second part if first retries. This capability permits threads to wait for various things at same time. They rely on the concept of using the retry and orElse that can absolutely be applied to other languages. This paper also showed an interesting difference between languages like C# or Java and Haskell in the context of "Atomic Block". In Haskell, codes inside an atomic block contain STM type thus it can only run by atomic execution. But in other conventional languages, atomicity is taken care by callee instead of caller thus it can be supported defensively at different stages in the call chain. After this effort, various other work conducted using STM Haskell. One such work was performed by Christian Perfumo [25] in 2007 where he used series of transactional Haskell application such as Block world, GCD, Prime, LL etc and drawn the outcomes from the data obtained by examining these applications. For this, he altered the Haskell Runtime System(RTS) with the help adding monitoring metrics which helped to gather transactional data like commit and abort rate along with their runtime overheads on mentioned application in the paper. On the basis of gather transactional data, new parameters like wasted time & useful work were obtained. Also, application suit used in the paper can serve as a benchmark for several research communities.

Nehir Sonmen et al [26] in 2007 proposed an extended Haskell STM for performance. His work was encouraged by the fact that although STM with Haskell supports lock free idea for concurrent programming by the use atomic operation on transactional variable still when it comes to linked structure atomicity might need more care than what is required. This may lead to a decrease in the whole performance. To stop above situation a completely new mechanism termed "unreadTVar" was coined for increasing the performance of specific application like a linked list. The use of unreadTVar provides two major benefits in the execution of transactional linked list. One is that they provide smaller dataset to transactions for work that considerably lessen the chances of having rollbacks. Second, the commit happens fastly as the number of TVars which have to be examined for consistency prior to commit is small. Besides the benefits there is two most essential demerit of employing unreadTvar, one is it needs more supervision by the programmer. As it is applied for performance increment. Another demerit is can't be applied to all kinds of operation which may be performed with linked structure.STM application programs feature called rollback rate will be helpful in deciding the transactional flexibility of the programs. Commit phase overhead is also one of the metrics that the authors proposed to assess the performance of STM.In brief, however supporting the programs with atomic security and extra cores seems to be beneficial but the overhead attached with transactional management also becomes high. It was stated in the paper that running the Altix 4700 ccNUMA machine on their benchmark for many of the application and Haskell STM system caused several scalability issues. The barrier can be discovered by doing a thorough examination of each application transactional performance i.e., abort rate, commit phase overhead and analysis about access made by transaction and hardware behavioral counters arranged in the paper. The authors also suggested some future work such as for application that does not execute for much time within transactions, their commit overhead seems to be high. For this, they advised to do further analysis of course grain and fine grain STM. Furthermore, more work required to deal with problems whether the recent system along with STM obey to the needs of transactional management or not.

Christian Perumo, Oswal Unsal Nehir Sonmenz in [27] together presented work on dissecting STM Haskell on the many-core environment. In their work they have shown Haskell STM application suit to be employed as a benchmark by various research centers as done in [25] but Haskell runtime system here is arranged with PAPI library as well as physically placed counters to gather data on the transaction part of every application like read/writesets, commit rate, abort rate, runtime overheads, wasted time, useful work as well as cache accesses for each transaction access made are determined. For finding the constraints of STM the transactional performance was noticed and examined on till 120 core implementation upon the ccNUMA machine. Along with this hardware behavior of STM was also examined for viewing the barrier present in STM.Mainly cache efficiency as well as halts were examined for stating essential scalability issues. This fact will be useful to researchers for analyzing the efficiency of their STM schemes.

Mertin Sulzamann[28] in 2009 introduced overall six algorithms extremely concurrent single linked list such as compare & swap applying either IORef or STM, STM, dissected STM, hand over hand applying either MVar or STM.They carried out comprehensive experiments to assess the comparative tradeoffs in every execution and additionally extract a comparison with linear execution. They also showed insertion of single primitive which enhances the execution of one STM algorithm with a factor of seven. The authors concluded that compare & swap algorithm with IORef is better than any other algorithm in terms order of magnitude. If we neglect the other interest, then an extremely concurrent data structure can use this mechanism. Nevertheless, it is also said that if readTVarIO is inserted then it is feasible to obtain similar performance of IORef with the use of STM as in some situation it maintains the capability for the use transaction in difficult modification of the data structure: in situation where use of compare & swap is going to be extremely difficult. In the case where composability is needed then pure STM algorithm is the only option. Here, one clear cut outcome is that, hand over hand locking is definitely not a good option as it scales badly. Along with that, it can't be composed and execution they showed here does not add the exception safety which will be required if they wanted to give as a library.

In 2012 Andre Rubre Dubois in [29] performed work on STM Haskell. His paper focuses on developing a new STM totally

written down in Haskell. The difference between the implementation before and this one is it applies early write/write conflict detection whereas in earlier implementations conflict detection happens at commit time only (which is used by almost all STM Haskell).The implementation used Swiss TM[30] algorithm after extending it, so that it provides retry and orElse transactional function support. The work contrast the swissTM of STM Haskell with remaining two implements (STM with TL2) with the use of Haskell STM benchmark suit[27,31].For many benchmarks the developed Swiss TM gives much superior performance then earlier developed STM on TL2.TL2 provides fine results up to 3 threads in the SI benchmark.(SI benchmark is a synthetic benchmark that pushes clash between access made to a single integer).Here for a benchmark like BT(Binary Tree), the Swiss TM in the absence of unreadTVar[26] provide optimization speedup to 1.18 with 8 threads. Although with just 6 threads unoptimized Swiss TM gives fine results. In the case of LL (Linked List) benchmark, STM Haskell has similar results as the optimized version of Swiss TM whose speed belongs in the range of 1.27 to 3.7 in contrast to the unoptimized version of SwissTM.SwissTM with the unreadTVar gives 39% slower performance than STM Haskell. In the case of HT (Hyper-Threading) benchmark, Swiss TM with 8 threads performs 4.5 times faster than STM Haskell with 6 threads.
David Sabel et al. [32] in 2014 also presented an STM Haskell implementation with conflict detection. In contrast to the implementation present in GHC, his approach can detect conflict as soon as possible. Pointer equality is not used to investigate conflict and is written down in Haskell, so it is not associated with the runtime system. They used the CSHF model for STM implementation in Haskell. Also, the semantics of CSHF was explained and its implementation as a library for STM Haskell was presented. The most important dissimilarity CSHF and STM Haskell in GHC are that in CSHF method the conflict detection might happens before in the GHC.CSHF does not depend on TVar content comparison. Furthermore, with respect to STM Haskell semantics GHC implements required to examine the transaction log on short term basis opposed to the state of global storage but in CSHF you need to do so. A demerit of CSHF is that in case of Haskell CSHE need to eliminate all the entries in the notify list made by thread whereas in GHC you just to have to eliminate the transaction log. A rough conclusion was given in the context of efficiency that STM implementation in GHC gives better results. Also, it was stated that STM implementation should be careful in terms of ceasing non-terminating transaction if there is chance of conflict. This is not the situation for every implementation of STM Haskell.
In 2016 Mathew Le [33] in his paper "Revisiting STM in Haskell" showed the redesigned STM in GHC which is similar to TL2 implementation that provides retry and orElse function even the absence of nested transactions. The authors also tested the orElse and retry performance. They used orElse and retry function to develop the work stealing scheduler based on STM.For using this scheduler they altered the Par monad which help them to prove that stealing scheduler is simpler and performs fine as the present schedular. Using the TL2 algorithm and using Haskell's retry and orElse blocking characteristics gives an easier implementation which helped to neglect trusted transaction plus log constructs.
Ammalan Ghosh and Rituparna Ghosh in [34] presented implementation of STM applying STM Haskell with the help of three distinct approaches namely TVars of STM Haskell and two TMVars approach: the two TMVars approaches used two different execution policy i.e., SJF(Shortest Job first) and FIFO(First In Out First).Transaction of varying length was considered. Also, they share common resource and executed concurrently. Each set of the transaction was made up by five write transaction. The implementation here provided a better result with SJF policy in the single threaded environment as we know SJF has minimum waiting time which in turn gives a lower turnaround time for processes whereas in the multithreaded environment all the parallel activities are managed by Haskell compiler. Here STM implementation with TVar is more efficient than other said approaches. This approach performs best in multithreaded environment where execution length is greater. In the third case, where transaction executes on FIFO pattern, average waiting time was higher which led to high Turn around Time (TAT) and lower throughput.
In 2018 Rodrigo Medeiros Duarte[35] showed the comparison of various implementation of concurrent hash table with the following algorithm mentioned in bracket i.e, Block MVar(using Lock Striping technique), Fine-grain applying STM(fine-grain lock implementation applying STM) Fine-grain applying MVar(fine-grain lock algorithm), CAS using STM(also lock-free but substituting IORefs by TVArs), CAS(lock-free hash applying IORef), Sequential(a linear variant of hash tables).For evaluating the various algorithm performance mentioned along with hast table techniques two different settings were arranged namely a Uniform Memory Access(UMA) architecture with Core i7 Intel processor and 4 physical cores plus Hyper threading(8 logical cores) and 8GB RAM. The second setting is Non-Uniform Memory Access (NUMA) accompanying 4 physical cores. Hyper threading (16 logical cores), 12GB RAM as well as two Intel Xeon. Here, BlockMVar, Fine-grain using STM, Fine-grain using MVar, CAS gave the finest result in case of UMA up to 8 threads in GHC 7.6.3.BlockMVar execution recompenses the performance barrier of a small number of locks preserving tables by its lesser complication for table duplication. Furthermore, fine-grain using MVar execution with a larger number of locks go through the effect of duplicating tables, and thus outcomes were identical to BlockMVar.CAS execution gives larger synchronization cost with performance poor than BlockMVar. NUMA machine performance was very akin to the UMA machine in case of fine-grain applying STM. Memory Contention, in general, is a critical problem as the amount of threads rises in NUMA machine.

An experiment conducted here also showed that Haskell Runtime System, adding its garbage collector was implemented primarily for UMA machine, besides no optimization in case of NUMA machine. In the evaluation environment mentioned in this work, fine grain using STM gives the finest performance till 8 threads and it is simpler to implement than other possible choices mentioned in the work. Among all implementation, CAS implementation has given best scalability.

4. ISSUES OF STM

4.1 Atomicity & code communication:

In Hardware Transactional Memory (HTM) model, usual memory accesses communicate fairly with transactions. Just committed state is seen by non-transactional reads and conflicts are determined across non-transactional modification and concurrent transactions accessing common data. Although, in the case of STM it is false as concurrency control process must be clearly presented. [2]

4.2 Burdens in STM:

In contrast to HTM or other shared memory programming STM gives greater linear overheads because of the software extension of loads and store on shared mutable areas within transactions to many of extra instructions that form the STM implementation. Based on the transactional feature of a workload, the burdens can become a greater obstacle for STM to obtain performance.[36]

4.3 Meanings:

To avoid incurring greater STM burdens, non-transactional accesses (like loads and stores happening outside transactions) are generally not extended. This led to the weakening and thus, confusing the descriptions of transactions, which may need the programmer to be extra cautious than in the case where strong transactional descriptions are supported. There are few of the weakened guarantees that are typically attached with such STMs:

- Conflict across transaction and not transactional accesses cannot be determined generally by STM runtime libraries.
- Few STM models prevent the absolute privatization of memory locations. For a few STM models, once a location is approached transactionally, it must uncease to be approached transactionally.
- Few STM models do not allow recapture memory locations approached transactionally for random reuse, like applying malloc and free [36].

4.4 Bequest Binaries:

STM requires examining every memory actions of the transactional area to confirm atomicity plus isolation. STMs which obtain above examination by code in code instrumentation usually cannot provide transaction asking legacy code which is not instrumented restricting concurrency like serializing transactions [36].

5. CONCLUSIONS

In this review paper, we primarily discussed various implementations using STM Haskell including the various elements involved in the mentioned implementations. The information about these STM Haskell implementation is necessary to work on new development through STM Haskell and draw some interesting characteristics of STM for future development

REFERENCES

[1] Marathe, Virendra J., Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisen-stat, William N. Scherer III, and Michael L. Scott. "Lowering the overhead of nonblocking software transactional memory." In Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT). 2006.
[2] Harris, Tim, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. "Transactional memory: An overview." *IEEE micro* 27, no. 3 (2007): 8-29.
[3] Shavit, Nir, and Dan Touitou. "Software transactional memory." *Distributed Compu-ting* 10, no. 2 (1997): 99-116.
[4] Damron, Peter, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. "Hybrid transactional memory." In ACM Sigplan Notices, vol. 41, no. 11, pp. 336-346. ACM, 2006.
[5] M. Herlihy, "Software Transactional Memory for Dynamic-Sized Data Structures*", Proc. 22nd Ann. Symp. Principles of Distributed Computing (PODC 03), pp. 92-101, 2003.*
[6] V.J. Marathe, W.N. Scherer, M.L. Scott, "Adaptive Software Transactional Memory*", Proc. 19th Int'l Symp. Distributed Computing (DISC 05) LNCS 3724, pp. 354-368, 2005.*
[7] K. Fraser, *Practical Lock-Freedom doctoral dissertation UCAMCL-TR579, 2004*.
[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In OOPSLA 2003 Conference Proceedings, Anaheim, CA, October 2003.
[9] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. InProceedings of the Nineteenth International Symposium on Distributed Computing, Cra-cow, Poland, September 2005.

[10] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High-Performance Software Transactional Memory System for a Multi-Core Runtime. In Proceedings of the Eleventh ACM Symposium on Principles and Practice of Parallel Programming, New York, NY, March 2006.

[11] Felber, Pascal, Christof Fetzer, and Torvald Riegel. "Dynamic performance tuning of word-based software transactional memory." In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 237-246. ACM, 2008.

[12] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy SnapshotAlgorithm with Eager Validation. In Proceedings of the 20thInternational Symposium on Distributed Compu-ting (DISC), pages 284–298, September 2006.

[13] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), pages 194–208, September 2006.

[14] Gray, J. and Reuter A. Transaction processing: concepts and techniques.

[15] Marathe, Virendra J., Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. "Lowering the overhead of nonblock-ing software transactional memory." In Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT). 2006.

[16] S. Peyton-Jones, A. Gordon, and S. Finne, "Concurrent Haskell", ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL), 1996.

[17] 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, Chicago, IL, USA, June 15-17, 2005.

[18] M. Herlihy and E. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", in 20th Annual International Symposium on Computer Architecture, May 1993.

[19] Blundell, Colin and Lewis, E Christopher and Martin, Milo M. K., "Subtleties of Transactional Memory Atomicity Semantics", Computer Architecture Letters, Vol 5, Number 2, November 2006.

[20] Afek, Yehuda, Dalia Dauber, and Dan Touitou. "Wait-free made fast." In *STOC*, vol. 95, pp. 538-547. 1995.

[21] Barnes, Greg. "A method for implementing lock-free shared data structures." (1994)

[22] Israeli, Amos, and Lihu Rappoport. "Disjoint-access-parallel implementations of strong shared memory primitives." In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pp. 151-160. ACM, 1994.

[23] Moir, Mark. "Transparent support for wait-free transactions." In *International Workshop on Distributed Algorithms*, pp. 305-319. Springer, Berlin, Heidelberg, 1997.

[24] Turek, John, Dennis Shasha, and Sundeep Prakash. "Locking without blocking: making lock based concurrent data structure algorithms nonblocking." In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 212-222. ACM, 1992.

[25] Perfumo, Cristian, Nehir Sonmez, Adrian Cristal,Osman Unsal, Mateo Valero, and Tim Harris. "Dissecting transactional executions in Haskell." In *TRANSACT'07: Second ACM SIGPLAN Workshop on Transactional Computing*. 2007.

[26] Sönmez, Nehir, Cristian Perfumo, Srdjan Stipic, Adrian Cristal, Osman S. Unsal, and Mateo Valero. "unreadTVar: Extending Haskell Software Transactional Memory for Performance." *Trends in Functional Programming* 8 (2007): 89-114.

[27] Perfumo, Cristian, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. "The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment." In *Proceedings of the 5th conference on Computing frontiers*, pp. 67-78. ACM, 2008.

[28] Sulzmann, Martin, Edmund SL Lam, and Simon Marlow. "Comparing the performance of concurrent linked-list implementations in Haskell." *ACM Sigplan Notices* 44, no. 5 (2009): 11-20.

[29] Du Bois, André Rauber, Maurício Lima Pilla, and Rodrigo Medeiros Duarte. "A High-Level implementation of STM Haskell with Write/Write conflict detection." In *2012 Third Workshop on Applications for Multi-Core Architecture*, pp. 24-29. IEEE, 2012.

[30] A. Dragojevi´c, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY,USA, 2009. ACM.

[31] The Haskell STM Benchmark. WWW page, http://www.bscmsrc.eu/software/haskell-stm benchmark,October 2010.

[32] Sabel, David. "A Haskell-Implementation of STM Haskell with Early Conflict Detection." In Software Engineering (Workshops), pp. 171-190. 2014.

[33] Le, Matthew, Ryan Yates, and Matthew Fluet. "Revisiting software transactional memory in Haskell." In *ACM SIGPLAN Notices*, vol. 51, no. 12, pp. 105-113. ACM, 2016.

[34] Ghosh, Ammlan, and Rituparna Chaki. "Implementing Software Transactional Memory Using STM Haskell." In Advanced Computing and Systems for Security, pp. 235-248. Springer, New Delhi, 2016.

[35] Duarte, Rodrigo Medeiros, André Rauber Du Bois, Maurício Lima Pilla, Gerson Geraldo H. Cavalheiro, and Renata Hax Sander Reiser. "Comparing the performance of concurrent hash tables implemented in Haskell." Science of Computer Programming (2018).
[36] Cascaval, Calin, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. "Software transactional memory: Why is it only a research toy?." *Queue* 6, no. 5 (2008): 40
[37] Website-1: https://slideplayer.com/slide/6257180 (as accessed on 15 May 2019)