# A Model for Concurrency in Multithreaded Nested Software Transactional Memory in Distributed Real time Database System

**Meenu**
**Associate Professor**
**Computer Science & Engineering Department**
**Madan Mohan Malaviya University of Technology**
**Gorakhpur**
**myself_meenu@yahoo.co.in**

## ABSTRACT

Software transactional memory (STM) is a programming abstract for shared variable concurrency. There are very few updates in Haskell's STM implementation since its introduction in 2005 by Tim Harris et al. In this paper we present a concurrency model, based on Software transactional memory, that offers nesting. In the STM implementation process, two different approaches have been presented. The first one is STM Haskell by using TVar without nesting, second one uses TVar with nesting.

**CCS Concepts:** • Information systems → Database Management; Database transaction processing

**KEYWORDS** Software transactional memory (STM), Non-nested TVar, NestedTVar

## 1 INTRODUCTION

Transactional memory (TM) [5,15] is a new way to simplify parallel programming. It is an optimistic concurrency-control mechanism [16] for controlling accesses to a shared memory region in concurrent programming.
Using Transactional memory (TM) permits grouping memory operations into transactions that execute atomically: no transaction views the intermediate states of other transactions executing in other threads, and all work of a transaction either happens (the transaction commits) or not (the transaction aborts). It is more abstract than locking, and avoids many of the problems encountered with locks, such as deadlock, priority inversion, convoying, pre-emption, and reduced concurrency. When two transactions access the same memory unit and at least one of the accesses is a write then there is a conflict: one of the transactions must

abort (discarding its pending writes) and restart. Transactional memory can be implemented in hardware, in software, or in a combination of the two.The original idea of a transactional memory with hardware support [15] was proposed by Herlihy et. al. Hardware transactional memory (HTM) is dependent on hardware structures such as caches and store buffers. That is why HTM has not been adopted widely by the computer industry to date. Shavit and Touitou further extended idea of a transactional memory with hardware support to a software only TM [13]. Software transactional memory (STM) provides software engineering benefits as it is independent of hardware structures such as caches and store buffers. There are several STM approaches [1-3,7,8,9,12]. These TM works triggered the development of several versions and extensions of hardware, software and hybrid TM implementations. Damron et al. propose a hardware/software hybrid approach [9] to transactional memory. Software and hardware Transactional memory schemes can combine into a hybrid system, which allows use of low-cost HTM when it works, but reverts to STM when it doesn't. Nested transactions [17] originated in the database community by Moss. Moss and Hosking introduced Nesting to Transactional Memory [10]. Nested transactional memory (TM) facilitates software composition by letting one module invoke another without either knowing whether the other uses transactions. Large transactions limit concurrency. Compared with locking, using transactions over the same objects improves concurrency in two ways. One is that transactions can proceed concurrently without conflict if they access different data, e.g., different fields of the same object or different objects. The other is that transactions distinguish between reading and writing, and concurrent reads do not conflict. Locks requires mutual exclusive access even for reads.

Haskell provides an excellent context for implementing transactional memory [14]. Haskell's type system differentiates static code that perform IO and code executing in a transaction. The STM Haskell [11] 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 T1 T2 has the following behavior:

• First T1 is executed, if it returns result then orElse function returns.

• If T1 retry instead then T1 is discarded and T2 is executed

## 2 IMPLEMENTATION

To implement STM in Haskell, we have chosen two different approaches to execute a specific task. The task is to read a sharable data object, calculate the account balance value, and finally write that account balance value to the sharable data object. The first approach uses non -nested TVars of STM Haskell. The atomically function of STMHaskell 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 approach uses nestedTVars [10].

### 2.1 STM Performance Metrics

For analyzing, the transactional behaviors of TM applications, following metrics [4,5] are commonly used:

**2.1.1 Execution Time**
**2.1.2 Aborts per Commit**
**2.1.3 Transaction Retry Rate**

#### 2.1.1 Execution Time

The execution time displays the transactional effectiveness of application scale with respect to the increasing number of threads.

### 2.1.2 Aborts per Commit

This is ratio of aborted transactions to committed transactions. This metric indicates the efficiency with which computing resources have been utilized as amount of work committed is assessed by the workload inputs

### 2.1.3 Transaction Retry Rate

This metric is used to exploit the inherent concurrency of the underlying STM implementation. A transaction self-aborts explicitly and in the process, it also reschedules itself after detecting its precondition for the operation, which may not hold any longer.

Performance comparison of two STM implementations is conducted on Execution Time parameter.

### 2.2 STM implementation using non-nestedTVar

Function deposit and withdraw are created to define the deposit and withdrawal task of a transaction using non-nestedTVar .The block of code is as follows:

```
deposit :: Account-> Int -> STM ()
deposit acc amt = do
bal<- readTVaracc
writeTVaracc (bal+amt)
withdraw :: Account -> Int -> STM ()
withdraw acc amt = do
bal<- readTVaracc
writeTVaracc (bal-amt)
```

Account is an integer type Transactional Variable. The type is defined as

```
type Account= TVar Int
```

The function deposit and withdraw have two parameters, a TVar and an integer. It calculates balance amount after deposit and withdrawal task of a transaction and writes that value to the TVar. Calculation of balance amount determines the execution time of the transaction.

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

```
main = do
acc<- atomically (newTVar 100)
   atomically (deposit acc 10)
  atomically (withdraw acc 10)
```

This code executes two transactions concurrently, where first one will write balance amount after deposit to the TVar and the second one will write balance amount after withdrawal to the TVar.

### 2.2.1 Program compilation

The command to compile the program is as follows:
ghc accnew.hs -rtsopts -prof
The command to execute the program is:
accnew +RTS –s
The flag –s, if included, shows the actual executions. The portion of the actual output is as follows:

```
Balance in acc: $100
Depositing $10 into acc-
Balance after deposit in acc: $110
Withdrawing $10 from acc-
Balance after withdraw from acc: $100
86,928 bytes allocated in the heap
2,752 bytes copied during GC
        43,936 bytes maximum residency (1 sample(s))
        25,696 bytes maximum slop
            2 MB total memory in use (0 MB lost due to fragmentation)
Tot time(elapsed) Avg pause Max pause
  Gen 0         0 colls, 0 par    0.000s 0.000s0.0000s    0.0000s
  Gen 1         1 colls, 0 par    0.000s   0.001s     0.0008s   0.0008s
INIT    time    0.000s (0.001s elapsed)
  MUT   time    0.000s (0.001s elapsed)
  GC      time    0.000s (0.001s elapsed)
  RP        time     0.000s (0.000s elapsed)
  PROF   time     0.000s (0.000s elapsed)
  EXIT   time   0.000s (0.000s elapsed)
  Total time    0.000s (0.003s elapsed)
%GC     time       0.0% (32.1% elapsed)
Alloc rate    0 bytes per MUT second
Productivity 100.0% of total user, 35.3% of total elapsed
```

The output shows the commit pattern of the transactions. The execution time is0.003s against actual 0.00 s. It also shows the 35.3 % productivity.

### 2.3 STM implementation using nested TVar

Function deposit and withdraw are created to define the deposit and withdrawal task of a nested transaction using nested TVar. The block of code is as follows:

```
deposit :: Account-> Int -> STM ()
deposit acc amt = do
```

```
bal<- readTVaracc
   if amt < 0
   then retry
   else writeTVaracc (bal+amt)
withdraw :: Account -> Int -> STM ()
withdraw acc amt = do
bal<- readTVaracc
   if amt > 0 && amt >bal
   then retry
   else writeTVaracc (bal-amt)
```

Account is an integer type Transactional Variable. The type is defined as

```
type Account= TVar Int
```

The function deposit and withdraw have two parameters, a TVar and an integer. It calculates balance amount after deposit and withdrawal task of a transaction and writes that value to the TVar. Calculation of balance amount determines the execution time of the transaction.

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

```
main = do
acc<- atomically (newTVar 100)
   atomically (deposit acc 10)
  atomically (withdraw acc 10)
```

This code executes two transactions concurrently, where first one will write balance amount after deposit to the TVar and the second one will write balance amount after withdrawal to the TVar.

### 2.3.1 Program compilation

The command to compile the program is as follows:

```
ghcacc.hs  -rtsopts -prof
```

The command to execute the program is:

```
acc +RTS –s
```

The flag –s, if included, shows the actual executions. The portion of the actual output is as follows:

```
Balance in acc: $100
Depositing $10 into acc-
Balance after deposit in acc: $110
Withdrawing $10 from acc-
Balance after withdraw from acc: $100
        87,096 bytes allocated in the heap
         2,752 bytes copied during GC
        43,936 bytes maximum residency (1 sample(s))
```

```
        25,696 bytes maximum slop
        2 MB total memory in use (0 MB lost due to fragmentation)
Tot time (elapsed) Avg pause  Max pause
  Gen  0         0 colls, 0 par    0.000s   0.000s     0.0000s    0.0000s
  Gen  1         1 colls, 0 par    0.000s   0.006s     0.0058s    0.0058s
INIT    time    0.000s  (0.000s elapsed)
  MUT   time    0.000s  (0.001s elapsed)
  GC    time    0.000s  (0.006s elapsed)
  RP    time    0.000s  (0.000s elapsed)
  PROF  time    0.000s  (0.000s elapsed)
  EXIT  time    0.000s  (0.000s elapsed)
  Total time    0.000s  (0.007s elapsed)
  %GC   time    0.0%  (80.0% elapsed)
Alloc rate    0 bytes per MUT second
Productivity 100.0% of total user, 16.5% of total elapsed
```

The output shows the commit pattern of the transactions. The execution time is 0.007sagainst actual 0.000 s. It also shows the 16.5 % productivity.

## 3. RESEARCH CHALLENGES

The design of a TM system that supports nested parallel transactions [6] is challenging.

**3.1 Transformation of transactional code**
**3.2 Conflict detection scheme**
**3.3 Memory overhead**
**3.4 Single level of parallelism**

### 3.1 Transformation of transactional code

Transformation of transactional code is also a challenge in STM. In databases non transactional code runs inherently as a transaction. In STM this is done by either separating transactional and non transactional code or dynamically categorizing their access to shared objects.

### 3.2 Conflict detection scheme

The conflict detection scheme must be able to correctly track dependencies in a hierarchical manner instead of a flat way. Nested parallel transactions may conflict and restart without necessarily aborting their parent transaction.

### 3.3 Memory overhead

Memory overhead necessary for tracking the state of nested transactions should be small.

### 3.4 Single level of parallelism

Some applications may not use nested parallelism, we must ensure that its overhead is reasonable when only a single level of parallelism is used**.**

## 4 SIMULATION RESULTS

In this experiment, while implementing Software Transactional Memory in STM Haskell, parallelism and concurrency both are taken care of. This case study considers that transactions perform task which can be executed in parallel and update the transactional variables.

In this case study two different approaches, are being considered. The first one is STM Haskell by using TVar without nesting, second one uses TVar with nesting.

We have used execution time as performance metric.The execution time displays the transactional effectiveness of application scale with respect to the increasing number of threads .The performance of two implementations varies due to two execution policies.

**Table 1. Performance of non-nested and nested TVar with execution time**

| | Execution Time |
|---|---|
| Non-nested TVar | 0.003 s |
| Nested TVar | 0.007 s |

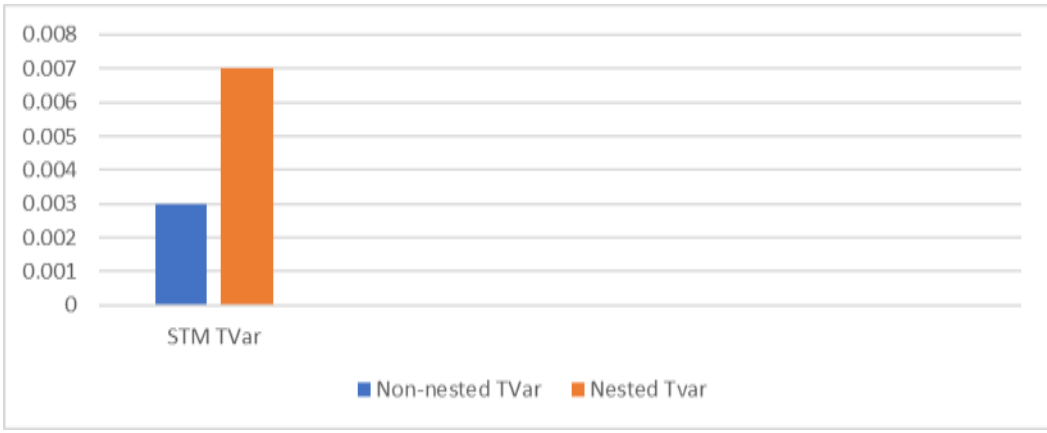


**Fig. 1. Performance graph of non-nested and nested TVar with execution time**

### 4.1 STM Implementation Using Non-Nested TVar

#### 4.1.1 Time and Allocation Profiling

```
accnew +RTS -p -i0.0001 -hy -RTS
total time =        0.00 secs (0 ticks @ 100 us, 1 processor)
total alloc =      68,888 bytes (excludes profiling overheads)
COST CENTRE MODULE          SRC             %time %alloc
MAIN       MAIN<built-in>0.0   48.9
CAF        GHC.IO.Handle.FD<entire-module>    0.0   50.5
individual      inherited
COST CENTRE MODULE                  SRC             no.     entries %time %alloc
%time %alloc
MAIN       MAIN<built-in> 41        0    0.0   48.9     0.0 100.0
CAF          GHC.IO.Encoding.CodePage<entire-module>62          0      0.0      0.3
0.0    0.3
CAF        GHC.IO.Encoding<entire-module> 55        0    0.0    0.1      0.0    0.1
CAF        GHC.IO.Handle.FD<entire-module>51        0    0.0   50.5      0.0   50.5
CAF        Main                    <entire-module> 48        0    0.0    0.2      0.0    0.2
```

#### 4.1.2 Heap Profiling

```
JOB "accnew +RTS -p -i0.0001 -hy"
DATE "Mon Aug 13 16:13 2018"
SAMPLE_UNIT "seconds"
VALUE_UNIT "bytes"
BEGIN_SAMPLE 0.000000
END_SAMPLE 0.000000
BEGIN_SAMPLE 0.015625
IO       40
IO       24
String   24
TextEncoding 32
IO       16
->IO     16
MVAR             32
Handle 24
Word32           16
ThreadId         16
-> (#, #)        40
->>IO   16
```

CodeBuffer 32
BufferState 24
ForeignPtrContents 48
TVAR 32
BufferCodec 48
(,) 48
Buffer 112
TVar 16
Int 16
->IO 16
Newline 24
Maybe 32
IO 24
IO 32
String 16
->> (#, #) 48
Handle__ 136
Newline 24
MUT_VAR_CLEAN 128
->IO 16
->IO 16
WEAK96
ARR_WORDS 36816
END_SAMPLE 0.015625
BEGIN_SAMPLE 0.015625
END_SAMPLE 0.015625

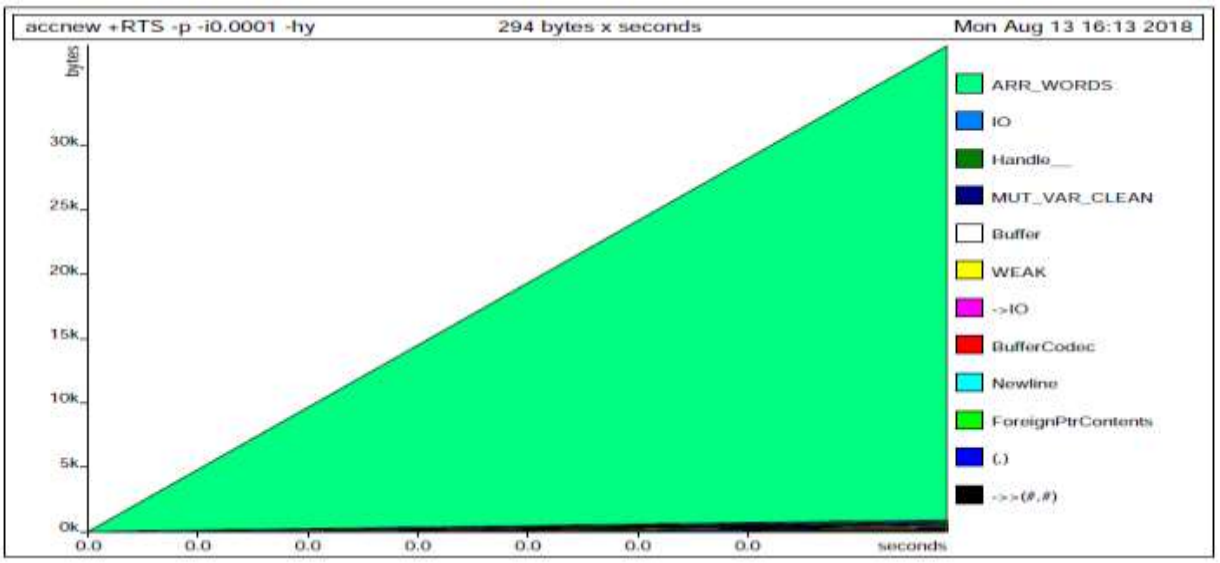

**Fig. 2. Performance graph of STM using non-nested TVar**

### 4.2 STM Implementation Using Nested TVar

#### 4.2.1 Time and Allocation Profiling

```
acc +RTS -p -i0.001 -hy -RTS
total time =        0.00 secs (0 ticks @ 1000 us, 1 processor)
total alloc =      68,944 bytes (excludes profiling overheads)
COST CENTRE MODULE          SRC             %time %alloc
MAIN      MAIN<built-in> 0.0   49.0
CAF       GHC.IO.Handle.FD<entire-module> 0.0   50.4
individual    inherited
COST CENTRE MODULE                    SRC no.    entries %time %alloc %time
%alloc
MAIN      MAIN<built-in> 41         0    0.0   49.0     0.0 100.0
CAF         GHC.IO.Encoding.CodePage<entire-module>62         0     0.0     0.3
0.0    0.3
CAF       GHC.IO.Encoding<entire-module>55        0    0.0    0.1     0.0    0.1
CAF       GHC.IO.Handle.FD<entire-module>51         0    0.0   50.4     0.0   50.4
CAF       Main                                <entire-module>48        0    0.0    0.2
0.0   0.2
```

#### 4.2.2 Heap Profiling

```
JOB "acc +RTS -p -i0.001 -hy"
DATE "Fri Aug 10 17:09 2018"
SAMPLE_UNIT "seconds"
VALUE_UNIT "bytes"
BEGIN_SAMPLE 0.000000
END_SAMPLE 0.000000
BEGIN_SAMPLE 0.015625
IO        40
IO        24
String    24
TextEncoding 32
IO        16
->IO      16
MVAR          32
Handle 24
Word32        16
ThreadId      16
```

-> (#, #) 40
->>IO 16
CodeBuffer 32
BufferState 24
ForeignPtrContents 48
TVAR 32
BufferCodec 48
(,) 48
Buffer 112
TVar 16
Int 16
->IO 16
Newline 24
Maybe 32
IO 24
IO 32
String 16
->> (#, #) 48
Handle__ 136
Newline 24
MUT_VAR_CLEAN 128
->IO 16
->IO 16
WEAK96
ARR_WORDS 36816
END_SAMPLE 0.015625
BEGIN_SAMPLE 0.015625
END_SAMPLE 0.015625

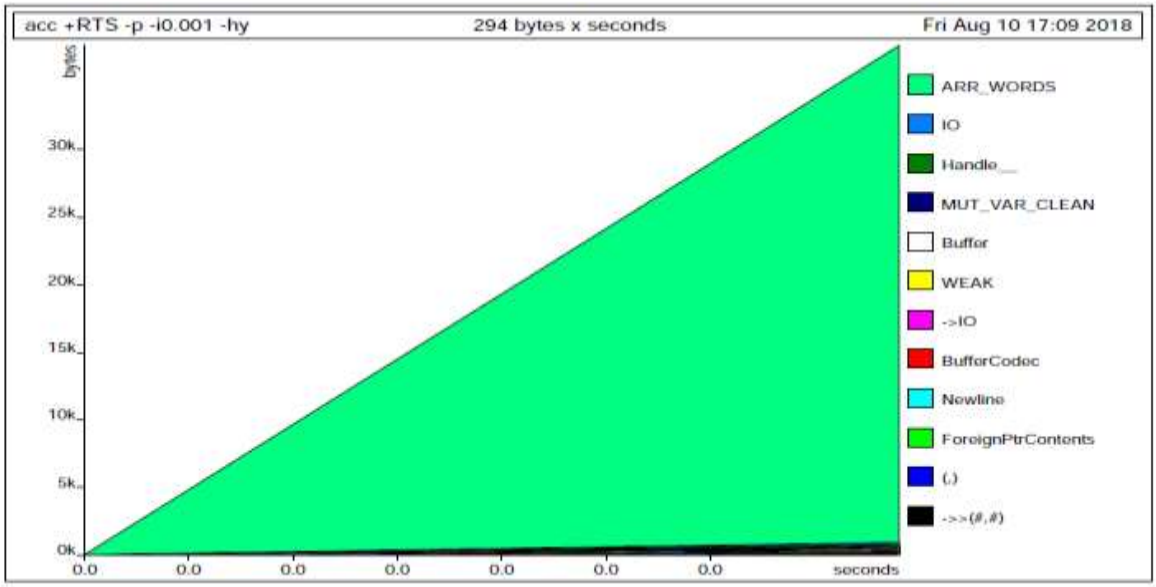


**Fig. 3. Performance graph of STM using nested TVar**

### 4.3 Summary

The nested software transactional memory has minimum waiting time comparable to non-nested software transactional memory, which implies low turnaround time for processes. That is why in single-threaded environment, nested software transactional memory implementation performs better than non-nested software transactional memory as nested transactions have several advantages. First, they provide higher level of concurrency among transactions. Second, it is possible to recover independently from failures of each subtransaction. Finally, it is possible to create new transactions from existing ones.

## 5 CONCLUSIONS

Our new STM implementation of nested TVar shows significant performance improvements over the existing implementation of non-nested TVar.Using nested STM improves concurrency level among transactions.

## REFERENCES

[1] Ghosh A., Chaki R. (2016) Implementing Software Transactional Memory Using STM Haskell. Advanced Computing and Systems for Security. Advances in Intelligent Systems and Computing, vol 396. Springer, New Delhi.

[2] Matthew Le, Ryan Yates, and Matthew Fluet. 2016.Revisiting software transactional memory in Haskell in Proceedings of the 9th International Symposium on Haskell .ACM, New York, NY, USA,

[3] Du Bois A.R. (2011) An Implementation of Composable Memory Transactions in Haskell. In: Apel S., Jackson E.(eds) Software Composition. SC 2011. Lecture Notes in Computer Science, vol 6708. Springer, Berlin, Heidelberg.

[4] Gulfam Abbas, Naveed Asif, "Performance Tradeoffs in Software Transactional Memory", Master Thesis Computer Science, School of Computing Blekinge Institute of Technology Sweden, No:MCS-2010-28, May 2010

[5] T. Harris, J. R. Larus, and R. Rajwar. Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[6] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '10, pages 253–262, Thira, Santorini, Greece, 2010. ACM

[7] Discolo A., Harris T., Marlow S., Jones S.P., Singh S. Lock Free Data Structures Using STM in Haskell. In: Hagiya M., Wadler P. (eds) Functional and Logic Programming. FLOPS 2006. Lecture Notes in Computer Science, vol 3945. Springer, Berlin, Heidelberg

[8] M. Herlihy, V. Luchangco and M. Moir, "A flexible framework for implementing software transactional memory, "Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications, pp. 253-262, 2006.

[9] P. Damron et al., ''Hybrid Transactional Memory,'' Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 06), ACM Press, 2006, pp. 336-346.

[10] J.E.B. Moss, A.L. Hosking, Nested transactional memory: Model and preliminary architecture sketches, in: the OOPSLA 2005 Workshop on

Synchronization and Concurrency in Object-Oriented Languages, SCOOL (no formal proceedings), October 2005.

[11] 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, PPoPP ’05, pages 48– 60, Chicago, IL, USA, 2005. ACM.

[12] M. Herlihy, V. Luchangco, M. Moir and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 92-101, 2003

[13] Nir Shavit and Dan Touitou. Software transactional memory. Distributed Computing. Volume 10, Number 2. February 1997.

[14] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In 23rd ACM Symposium on Principles of Programming Languages (POPL’96), pp. 295–308.

[15] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93). Volume 21, Issue 2, May 1993.

[16] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, Vol. 13(2), June 1981, pp. 186 – 221.

[17] J.E.B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, Apr. 1981; also published as MIT Laboratory for Computer Science Technical Report 260.