International Journal of Computer and Communication Technology

Volume 5 | Issue 3 Article 10

July 2014

STM: Lock-Free Synchronization

Ryan Saptarshi Ray
Department of Information Technology Jadavpur University, Kolkata, India, ryan.ray@rediffmail.com

STM: Lock-Free Synchronization

Ryan Saptarshi Ray
Department of Information Technology
Jadavpur University, Kolkata, India
E-mail : ryan.ray@rediffmail.com

Abstract— *Current parallel programming uses low-level programming constructs like threads and explicit synchronization (for example, locks, semaphores and monitors) to coordinate thread execution which makes these programs difficult to design, program and debug.*

In this paper we present Software Transactional Memory (STM) which is a promising new approach for programming in parallel processors having shared memory. It is a concurrency control mechanism that is widely considered to be easier to use by programmers than other mechanisms such as locking. It allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, concurrently executing parts of the program.

Keywords- *Parallel Programming; Locks; Transactional Memory; Software Transactional Memory*

I. Introduction

Generally one has the idea that a program will run faster if one buys a next-generation processor. But currently that is not the case. While the next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If one wants programs to run faster, one must learn to write parallel programs as currently multi-core processors are becoming more and more popular. Parallel Programming means using multiple computing resources like processors for programming so that the time required to perform computations is reduced. [**1**]

The hardest problem that must be overcome when writing parallel programs is that of synchronization. Multiple threads may need to access the same locations in memory and if careful measures are not taken the result can be disastrous. If two threads try to modify the same variable at the same time, the data can become corrupt. Currently locks are used to solve this problem.

Using locks, when a thread tries to enter a critical section, it must first acquire that section's lock. If another thread is already holding the lock, the former thread must wait until the lock-holding thread releases the lock, which it does when it leaves the critical section. Some of the drawbacks of locks are priority inversion, convoying and deadlocks. [**2**]

Transactional memory (TM) is an alternative paradigm to lock-based concurrent programming. Derived from transactional databases, TM uses transactional semantics for critical code regions that require synchronization. Programmers utilizing Transactional Memory(TM) have to enclose segments of code that access shared variables in transactions. Consequently, the TM system guarantees the atomicity, consistency, isolation and durability of executing critical regions. Atomicity means that a critical section will execute completely or not at all. No other threads will be able to see a state of memory where a critical section is only partially complete. Consistency means that data will never get corrupted. Isolation means that the execution of a critical section of a thread will never be affected by the actions of other threads. Durability means that any committed memory modifications are reliable. Code using transactions is very readable and understandable. If the transaction is successfully executed, then the "commit" of the transaction is performed. If conflict occurs, a contention manager is consulted in order to resolve the conflict. After conflict resolution, a single conflicting transaction will continue execution, while the remaining conflicting ones will be "aborted". [**2**]
There are two types of Transactional Memory; Hardware Transactional Memory and Software Transactional Memory.

Hardware Transactional Memory can be implemented by modifying standard processors, multiprocessor cache-coherence protocols and bus protocols to support transactions. But there are some drawbacks to hardware transactional memory that are considered major issues. So we consider Software Transactional Memory in this paper. [**3**]

Software Transactional Memory (STM) supports flexible transactional programming of synchronization operations in software. STMs also support lightweight transactions in concurrent applications. STM has advantages in terms of applicability to today's machines, portability and resiliency in the face of timing anomalies and processor failures. Software Transactional Memory (STM) is a concurrency control mechanism that is widely considered to be easier to use by programmers than other mechanisms such as locking. It allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. [**2**]

II. TRANSACTIONAL MEMORY

Transactional memory (TM) is an alternative paradigm to lock-based concurrent programming which does not suffer from the drawbacks of lock-based concurrent programming. The TM system guarantees the atomicity, consistency, isolation and durability (the ACID properties) of executing critical regions. A "transaction" is a form of program execution borrowed from the database community. Database systems have successfully been exploiting concurrency for decades using transactions. Transactional Memory was originally proposed as a mechanism for lock-free data synchronization and since then has become more and more popular. [**2**]**,** [**4**]

A. Transactional Memory General Implementation

Any critical section of code that is to made atomic must be enclosed within a transaction surrounded with, for example, xbegin and xend tags. When inside a transaction, any attempts to read or write to memory are buffered to some kind of log file. When a transaction ends, if there is no conflict detected, the transaction commits all of its memory modifications from the log file and exits, else the transaction wipes the log file clean and reverts back to the beginning of the transaction. An implementation of transactional memory such as this is called optimistic execution. [**2**]

It is considered to be optimistic because when an xbegin tag is reached, the system enters the transaction with the hope that it will be able to commit all of its changes at the end. Thus a transaction does not worry about obtaining any locks. It simply executes right away and records any memory reads or writes to the log. The verification step at the end checks that the log is valid before the changes are committed. To check that the log is valid, the system must go through every variable that was read or written to ensure that their values are consistent with what they were when the transaction began thus ensuring isolation. The verification step is done atomically.
Overall this is a very clean solution to parallel programming, as concurrency is dealt with simply by surrounding all critical sections with xbegin and xend tags.

B. Pros and Cons of Transactional Memory

1) Pros

Parallel programming poses many difficulties. The most serious challenge is coordinating access to data shared by several threads. Data races, deadlocks, and poor scalability are consequences of trying to ensure mutual exclusion with too little or too much synchronization. Currently locks are used for synchronization in parallel programs. But locks suffer from many drawbacks. TM is another approach for performing synchronization in parallel programs. TM overcomes all the problems which occur while performing synchronization using locking. TM offers a simpler alternative to mutual exclusion by shifting the burden of correct synchronization from a programmer to the TM system. The programmer only needs to identify a sequence of operations on shared data that should appear to execute atomically to other, concurrent threads. After that, through different mechanisms, TM ensures that synchronization is performed. Transactions also make composition possible. TM also overcomes the problems of priority inversion, deadlocks and convoying.

2) Cons

Firstly, transactional memory leads to too much overhead with respect to performance. Secondly, there is the problem of transactional code interacting with non-transactional code. There will always be systems with legacy code and thus this issue needs to be considered. It is unclear how to deal with shared data outside of a transaction (i.e. how to tolerate weak atomicity) and how to deal with locks being used inside transactions. Thirdly, there is the issue of dealing with exceptions. There needs to be an elegant mechanism to handle exceptions and propagate exception information from within a transactional context. Finally, there are some code which cannot be transactionalized such as when I/O is required. In optimistic TM, a transaction that executed an I/O operation may roll back at a conflict. I/O in this case consists of any interaction with the world outside of the control of the TM system. If a transaction aborts, its I/O operations should roll back as well, which may be difficult or impossible to accomplish in general. Buffering the data read or written by a transaction permits some rollbacks, but buffering fails in simple situations, such as when a transaction writes a prompt and then waits for user input.

Because of all these issues transactional memory has not yet matured to the point where it will be widely adopted. Better implementation techniques that are likely to improve performance of transactional memory are an area of active research. [**2**], [**4**]

III. SOFTWARE TRANSACTIONAL MEMORY

Implementation of Transactional Memory entirely in software is called Software Transactional Memory (STM). In STM it is possible to implement lock-free, atomic, multi-location operations entirely in software. STM is a novel design that supports flexible transactional programming of synchronization operations in software.

STM is a promising technique for controlling concurrency in modern multi-processor architectures. [**2**], [**4**]

A. Software Transactional Memory Implementation Details

STM also follows optimistic execution. The important issues which should be kept in mind while implementing STM are contention management and strategies for reads, writes, locking and nesting.

Contention managers in different types of STM follow different strategies some of which are backoff, priority, greedy, delay, suicide and aggressive.

Reads in STM can be either visible reads or invisible reads. In visible reads the transaction has to establish itself as the owner of an object before reading the object. In invisible reads the transaction does not have to establish itself as the owner of the object before reading it; but for every read, consistency till that read is checked.

Writes in STM can be either write-back or write-through. In write-back the new written values are written to the write log initially. After the transaction commits the necessary updates are made by checking the values from the write log. If a transaction does not commit but aborts due to some reason, then this approach is advantageous. In write-through the new values are immediately written to memory. An undo log is also maintained so that the changes can be undone in case the transaction aborts. In some cases the conflicting transaction itself may abort due to some reason. Then there is no need for the current transaction to abort. In these situations the write-through approach is advantageous as it prevents the unnecessary abort of the current transaction.

Locking in STM can be done either at commit time or at encounter time. In commit-time locking locks are acquired when the transaction commits. The advantage of this approach is that a conflicting transaction itself may abort due to some reason. In that case unnecessary lock contention is avoided. In encounter-time locking whenever a transaction has to write an object it has to acquire a lock on it. The advantage of this approach is that it avoids unnecessary work. If locks are acquired at the time of commit of the transaction, then in most cases at least one of the conflicting transactions has to abort.

Locks can be either fine-grained or coarse-grained. In coarse-grained locking a single lock covers the entire memory. But this type of lock is not scalable. In fine-grained locks there are multiple locks each of which covers a different part of memory.

There are two types of nesting in STM which are open-nesting and closed-nesting. In open-nesting when the inner transaction commits, then the changes it has made are visible to all other transactions. In closed-nesting when the inner transaction commits, then the changes it has made are visible to only the outer transaction which encloses it. [**5**], [**6**]

B. Pros and Cons of Software Transactional Memory

1) Pros

STM overcomes all the problems which occur while performing synchronization using locking. STM is easier to use than locks. STM offers a simpler alternative to mutual exclusion by shifting the burden of correct synchronization from a programmer to the STM system. The programmer only needs to identify a sequence of operations on shared data that should appear to execute atomically to other, concurrent threads. After that through different mechanisms the STM system ensures that synchronization is performed. STM allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, concurrently executing parts of the program.

STM also ensures composition in synchronization. A programming abstraction is said to support composition if it can be correctly combined with other abstractions without needing to understand how the abstractions operate. Through different other mechanisms the STM system also overcomes the problems of priority inversion, deadlocks and convoying which occur while performing synchronization using locks.

STM is more scalable than explicit coarse-grained locking and easier to use than fine-grained locking. STMs also support lightweight transactions in concurrent applications. STM has advantages in terms of applicability to today's machines, portability and resiliency in the face of timing anomalies and processor failures. STM also provides all the advantages which are provided by Transactional Memory. [**4**], [**6**]

2) Cons

Firstly, there is the problem of transactional code interacting with non-transactional code. There will always be systems with legacy code and thus this issue needs to be considered. It is unclear how to deal with shared data outside of a transaction (i.e. how to tolerate weak atomicity) and how to deal with locks being used inside transactions. Secondly, some code cannot be transactionalized, such as when I/O is required. In optimistic STM, a transaction that executed an I/O operation may roll back at a conflict. I/O in this case consists of any interaction with the world outside of the control of the TM system. If a transaction aborts, its I/O operations should roll back as well, which may be difficult or impossible to accomplish in general. Buffering the data read or written by a transaction permits some rollbacks, but buffering fails in simple situations, such as when a transaction writes a prompt and then waits for user input. Thirdly, the overhead involved in case a transaction has to roll back due to a conflict is also huge. Fourthly, the performance of code using STM is either equal to or worse than that of code using locks and threads.

Because of all these disadvantages STM is still an area of active research.

IV. TYPES OF SOFTWARE TRANSACTIONAL MEMORY

Different types of STM are Static STM, Dynamic STM, Object-based STM and Word-based STM.
In static STM the required data set and the inputs to the transactions are known in advance. From this information the outputs can also be calculated in advance.
In dynamic STM the data sets and the inputs to the transactions can be changed dynamically. Thus dynamic STM is more flexible, advantageous and more popular. In object-based STM more than one transaction cannot access the same object simultaneously.
In word-based STM more than one transaction cannot access the same field of the same object simultaneously. Word-based STMs can be easily integrated into different programming languages which is not the case with object-based STM.

V. DIFFERENT IMPLEMENTATIONS OF SOFTWARE TRANSACTIONAL MEMORY

Some types of STM implementations are adapt STM, Time-based STM, Log-based STM, TL2 and TinySTM.

A. adaptSTM

adaptSTM is a flexible STM library with a non-adaptive baseline common to current fast STM libraries to evaluate different performance options. The baseline is extended by an online evaluation system that enables the measurement of key runtime parameters like read-and write locations, or commit- and abort-rate. The performance data is used by a thread-local adaptation system to tune the STM configuration. The system adapts different important parameters like write-set hash-size, hash-function, and write strategy based on runtime statistics on a per-thread basis.
AdaptSTM uses local adaptivity. This means that the different parameters are measured on a per-thread basis. The advantage of local adaptivity over global adaptivity is that every thread has its local settings, e.g., a reader-thread optimizes the transactional parameters for best read performance and a writer-thread optimizes for write throughput.
Global adaptivity is a bottleneck for scalability as it requires global synchronization and barriers for all threads that make frequent changes of the adaptive parameters expensive. In local adaptivity each thread can change the local transactional settings without synchronization overhead every time a transaction is started or restarted.
Some different parameters which are adapted according to the prevailing situation in adaptSTM are discussed below:

1) Write-back vs. write-through

adaptSTM samples the abort rate and decides to switch between write-back and write-through, if the abort rate reaches a threshold. The adaptation system uses the average of the last 64 transactions to calculate the abort rate. If the number of aborts is greater, then adaptSTM uses write-back. On the other hand if the number of commits is higher then adaptSTM uses write-through.

2) Hash-table size

The size of the write-set hash-table is crucial for good performance. If the hash-table is too large, then the overhead of resetting the table every time a transaction starts is high.
On the other hand, if the table is too small, then the lookup will be slow due to many hash collisions. In the current implementation of adaptSTM, hash collisions are queued in a linked list in the same hash-table slot. The adaptation system samples the moving average of unique write locations per transaction. If the load of the hash-table is more than 33% then the size of the table is doubled. On the other hand, if the load is below 10% then the size of the table is halved.

3) Adaptive Contention Management

An extension of the basic contention management is to scale the number of yield operations according to the overall contention in the system. The current transaction is yielded an amount of times relative to the number of retries for this transaction.
This adaptive contention strategy implements a backoff strategy that retries immediately if the contention is low, or yields an increasing amount of times in contended situations. [**7**]

B. Time-based STM

STMs either rely on visible read designs, which simplify conflict detection while pessimistically ensuring a consistent view of shared data to the application, or optimistic invisible read designs that are significantly more efficient but require incremental validation to preserve consistency, at a cost that increases quadratically with the number of objects read in a transaction. Time-based STM benefits from the advantage of invisible reads without incurring the quadratic overhead of incremental validation. The first time-based STM algorithm was the Lazy Snapshot Algorithm (LSA). Its performance is highly competitive, both for obstruction-free and lock-based STM designs. [**5**]

C. Log-based STM

Transactional memory (TM) simplifies parallel programming by guaranteeing that transactions appear to execute atomically and in isolation. Implementing these properties includes providing data version management for the simultaneous storage of both new (visible if the transaction commits) and old (retained if the transaction aborts) values.
Most TM systems leave old values “in place” (the target memory address) and buffer new values elsewhere until commit. This makes aborts fast, but penalizes the much more frequent commits.
Log-based Transactional Memory (LogTM), makes commits fast by storing old values to a per-thread log in cacheable virtual memory and storing new values in place. [**8**]

D. TL2

Transactional locking II (TL2) algorithm is a word-based software transactional memory (STM) algorithm based on a combination of commit-time locking and a novel global version-clock based validation technique. TL2 improves on state-of-the-art STMs in the following ways:
(1) Unlike all other STMs it fits seamlessly with any systems memory life-cycle, including those using malloc/free.
(2) Unlike all other lock-based STMs it efficiently avoids periods of unsafe execution. This means that using its novel version-clock validation, user code is guaranteed to operate only on consistent memory states.
(3) In a sequence of high performance benchmarks, while providing these new properties, it delivered overall performance comparable to and in many cases better than that of all former STM algorithms, both lock-based and non-blocking. On various benchmarks, TL2 delivers performance that is competitive with the best hand-crafted fine-grained concurrent structures. It is ten-fold faster than a single lock.

These characteristics make TL2 a viable candidate for deployment of transactional memory today. [**9**]

E. TinySTM

TinySTM is a word-based and time-based software transactional memory implementation that uses locks to protect shared memory locations. Its name stems from the simplicity and performance of its design. It is lightweight and highly efficient. It performs better in many situations than TL2 which is currently one of the fastest word-based software transactional memories.
TinySTM uses a single-version, word-based variant of the LSA algorithm. It shares many properties with other word-based STMs like TL2. But it also follows different design strategies on some key aspects. Automatic tuning can be performed in TinySTM and it is also adaptive. [**10**]

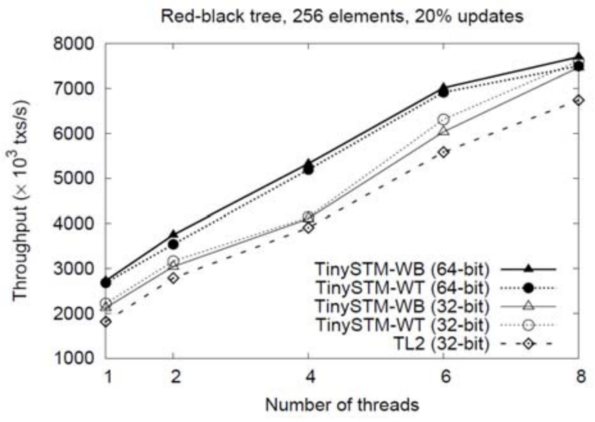

Figure 1: Performance of TinySTM versus TL2

In the above figure, the lowest line shows the throughput of TL2 and the other lines show the performance of different types of TinySTM in a red-black tree application. As can be clearly seen the performance of all types of TinySTM are better than that of TL2. [**11**]

VI. CONCLUSION

STM has been shown in many ways to be a good alternative to using locks for writing parallel programs. While locks are messy and complicated, STM primitives are elegant and allow code synchronization sections to be easily implemented and understood by developers. STM by itself is unlikely to make multicore computers readily programmable. Many other improvements to programming languages, tools, runtime systems, and computer architecture are also necessary. STM, however, does provide a timetested model for isolating concurrent computations from each other. This model raises the level of abstraction for reasoning about concurrent tasks and helps avoid many parallel programming errors.

This paper has discussed different types and implementations of STM and also different issues to be considered while implementing STM. It has also discussed the benefits and drawbacks of STM.
Many aspects of the semantics and implementation of STM are still the subject of active research. While it may still take some time to overcome the various drawbacks, the necessity for better parallel programming solutions will drive the eventual adoption of STM. Once the adoption of STM begins it will have the potential to pick up momentum and make a very large impact on software development in the long run. In the near future STM will become a central pillar of parallel programming.

APPENDIX

```
1   template<class T> void CoarseQ<T>: : enq (T t )
2   {
```

```
3        pthread_mutex_lock (&m) ;
4        node _tmp = new node ( t ) ;
5        if ( back == NULL)
6          { front = back = tmp ;}
7        else
8          { back−>next = tmp ;
9            back = tmp ;
10         }
11     pthread_mutex_unlock (&m) ;
12     }
```

Code Snippet 1: Enqueue method for a coarse-locked linked list implementing a concurrent queue

Code Snippet 1 shows the enqueue method for a concurrent queue implemented as a linked list with a single coarse lock. Lock is used (line 3) and a new node is allocated (line 4). If the back pointer doesn't point to anything, i.e. the queue is empty (line 5), the queue will now have one element, and both the front and back pointers will point to it (line 6). Otherwise the new node is swung onto the end of the back pointer (line 8) and the back pointer points to the new node (line 9). Finally the lock is unlocked (line 11).

```
1    template<class T> void TxQ<T>: : enq (T t )
2    {
3      START( 0 , RW) ;
4       node _tmp = ( node _) stm_malloc ( sizeof (
                       node ) ) ;
5     tmp−>t = t ;
6     node _myback = ( node _) LOAD_PTR(&back
                       ) ;
7    i f ( myback == NULL)
8      {STORE_PTR(&front , tmp ) ;
9      STORE_PTR(&back , tmp ) ;
10    }
11    e l s e
12     { STORE_PTR(&back−>next , tmp ) ;
13       STORE_PTR(&back , tmp ) ;
14    }
15   COMMIT;
16   }
```

Code Snippet 2:Enqueue method for a linked list implementing a concurrent queue with TinySTM

```
1 # define RO 1
2 # define RW 0
3 # define START( id , ro )
{ \
4 sigjmp_buf  * _ e = stm_getenv ( ) ; \
5 stm_tx_attr_t  _a = { id , ro } ; \
6 sigset_jmp (*_ e , 0 ) ; \
7 stm_start ( _e , & a )
8 # define COMMIT stm_commit ( ) ; }
```

Code Snippet 3: Convenience macros for TinySTM transactions

Code Snippet 2 shows the enqueue method for a concurrent queue implemented as a linked list using TinySTM transactions. A few macros have been defined to make TinySTM a little easier to use. The main idea is that a transaction is started (line 3), a new node is allocated (line 4), and it is checked if the queue is empty (lines 6–7). If it is, both front and back pointers point to the new node (lines 8–9). Otherwise the new node is swung onto the end of the back pointer (line 12) and the back pointer points to the new node (line 13). Finally the transaction (line 15) is commited. Thus, the logic of this method is identical to that of the coarse-locked queue. The main difference is syntactic, because, one has to explicitly tell TinySTM which loads and stores inside a transaction to monitor. The wrappers.h header provides the functions stm_load_ptr and stm_store_ptr , and the LOAD_PTR and STORE_PTR macros are wrappers around them, just to avoid having to do the type casting each time:

```
# define LOAD_PTR( addrofptr ) \
stm_load_ptr ( ( volatile void __) addrofptr )

# define STORE_PTR( addrofptr , value ) \
Stm_store_ptr ( ( volatile void __) addrofptr , \
( void _) v a l u e )
```

One also has to be explicit about allocations inside a transaction, since TinySTM has to know how to undo them in the event that a transaction aborts. Thus in line 4, stm_malloc (defined in the mod mem.h header) is used instead of malloc or operator new.
Let us see what the START and COMMIT macros are doing (Figure 4). The way that TinySTM implements flow of control around a transaction is on top of the old setjmp and longjmp standard library functions. The START macro just sets up a jump buffer pointing at the beginning of the transaction (lines 4–6), so that if the transaction aborts, it will automatically retry. If one does not want this behavior, TinySTM allows one to pass a null jump buffer to stm_start. In this case one has to manually check the return from stm_commit to see whether the transaction succeeded. The id parameter of START is just an identifier for the present transaction that might be helpful for debugging, and the ro parameter is a hint about whether the transaction is read-only or not.
There are a couple of other things one has to do to set up TinySTM. One has to initialize the library at the outset with stm_init, and one has to initialize each thread that will perform transactions (including the "main" thread that called stm_init, if need be) with stm_init_thread. There are also corresponding thread and global shutdown functions which are stm_exit_thread and stm_exit respectively.

ACKNOWLEDGEMENT

I would like to thank my advisor Prof. Utpal Kumar Ray, Associate Professor, Department of Information Technology, Jadavpur University whose suggestions, guidance and encouragement have helped me immensely in understanding the subject and without whose help this paper would not have been completed.

REFERENCES

[1] Simon Peyton Jones, “Beautiful concurrency”.

[2] Elan Dubrofsky, “A Survey Paper on Transactional Memory”.

[3] http://en.wikipedia.org/wiki/Transactional_memory

[4] James Larus and Christos Kozyrakis. “Transactional Memory”

[5] Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel, “Time-Based Software Transactional Memory”

[6] Tim Harris, James Larus, Ravi Rajwar, “Transactional Memory”

[7] Mathias Payer, Thomas R. Gross, “Performance Evaluation of Adaptivity in Software Transactional Memory”

[8] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, David A. Wood., “LogTM: Log-based Transactional Memory”

[9] Dave Dice , Ori Shalev , Nir Shavit., “Transactional Locking II”

[10] http://tmware.org

[11] Pascal Felber, Christof Fetzer, Torvald Riegel, “Dynamic Performance Tuning of Word-Based Software Transactional Memory”.

[12] Maurice Herlihy, J. Eliot B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”.

[13] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, Luca Benini, “Towards Transactional Memory Support for GCC”.

[14] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, Michael L. Scott, “Lowering the Overhead of Nonblocking Software Transactional Memory”.

[15] Utku Aydonat, Tarek S. Abdelrahman, Edward S. Rogers Sr., “Serializability of Transactions in Software Transactional Memory”.

[16] Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”.

[17] Brendan Linn, Chanseok Oh, “G22.2631 project report: software transactional memory”.