Ryan Saptarshi Ray, Prof. Utpal Kumar Ray a / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 3, Issue 2, March -April 2013, pp.1533-1540

Different Approaches for improving Performance of Software Transactional Memory

***Ryan Saptarshi Ray,**Prof. Utpal Kumar Ray**
*PhD Scholar Jadavpur University Kolkata, West Bengal, India
**Associate Professor Department of Information Technology Jadavpur University Kolkata, West Bengal,India

Abstract
The past few years have marked the start of a historic transition from sequential to parallel computation. Earlier a transition of a processor from one generation to another meant that its speed increased. So programmers knew that a program would run faster if a next-generation processor was bought. 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. Thus the necessity to write parallel programs is increasing as systems are getting more complex while processor speed increases are slowing down. Current parallel programming uses low-level programming constructs like threads and explicit synchronization (for example, locks, semaphores and monitors) to coordinate thread execution. Parallel programs written with these constructs are difficult to design, program and debug. Also locks have many drawbacks which make them a suboptimal solution.

Software Transactional Memory (STM) 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, tasks of the program which are executing at the same time. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, parts of the program which are executing simultaneously.

Despite its advantages in programmability, currently the performance of code using STM is worse than that of code using locks. The primary reason for the low performance is due to the extra overheads associated with maintaining the modification logs and aborting/committing the transactions. Consequently, various designs have been proposed to improve the execution speed of STMs.

This paper gives a brief overview of STM and shows different techniques which are currently being used to improve the performance of STM.

1. Introduction

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 TM have to enclose segments of code that access shared variables in transactions. Consequently, the TM system guarantees the atomicity, consistency, isolation and durability (the ACID properties) 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. Another big advantage of transactional memory is that it makes synchronization simple to implement which is not the case with locks. Also code using transactions is very readable and understandable. If transactions are successfully executed then the changes they make are permanent, so to say they "commit". 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". A number of hardware and software TM systems have been developed. Currently even hybrid approaches are being proposed. [**4**]

Software Transactional Memory solves all the problems which occur while using code with locks. 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. 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. [**4**]

Different types of STMs which can be implemented by different programming languages have been proposed.

2. Software Transactional Memory

2.1 Software Transactional Memory Overview

Software Transactional Memory (STM) is a promising new approach to programming shared-memory parallel processors. 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.

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. In STM also any critical section of code that one wants made atomic must be enclosed within a transaction. The STM system also guarantees the atomicity, consistency, isolation and durability (the ACID properties) of executing critical regions. 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 is emerging as a highly attractive programming model due to its ability to mask concurrency management issues to the overlying applications. [**4**], [**10**]

2.2 Pros and Cons of Software Transactional Memory

Pros

STM gives all the benefits which are given by Transactional Memory.

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 itself also provides some additional advantages which are discussed below.

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. [**10**], [**14**]

Cons

STM faces a number of challenges which are discussed here.

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, till now it has been seen that the performance of code using STM is either equal to or worse than that of code using locks and threads.

Despite its advantages in programmability, in practice STM suffers a performance hit by as much as 50% relative to fine-grained lock-based code. The primary reason
for the low performance of STM is due to the extra overheads associated with maintaining the modification logs and aborting/committing the transactions. Consequently, various designs have been proposed to improve the execution speed of STMs. STM systems either impose substantial overhead (performance degradation) in order to guarantee that user code always executes in a "consistent"state, or allows inconsistent states to be observed, which can lead to arbitrary behaviour. The performance of an STM depends highly on the target application, the workload, the underlying architecture and the number of threads used for parallelizing
the code. Performance of STM may decrease after a certain point even if the number of threads goes on increasing. This occurs because the contention becomes too high after a certain point.

So to ensure that STM becomes still more widely used different approaches must be tried so that the performance of codes using STM becomes better than that of codes using locks. Some of these approaches are discussed next. [**4**], [**10**]

3. Different Approaches for improving performance of Software Transactional Memory

Different types of approaches to improve performance of STM are discussed below.

3.1 Approach based on Early Abort Mechanism

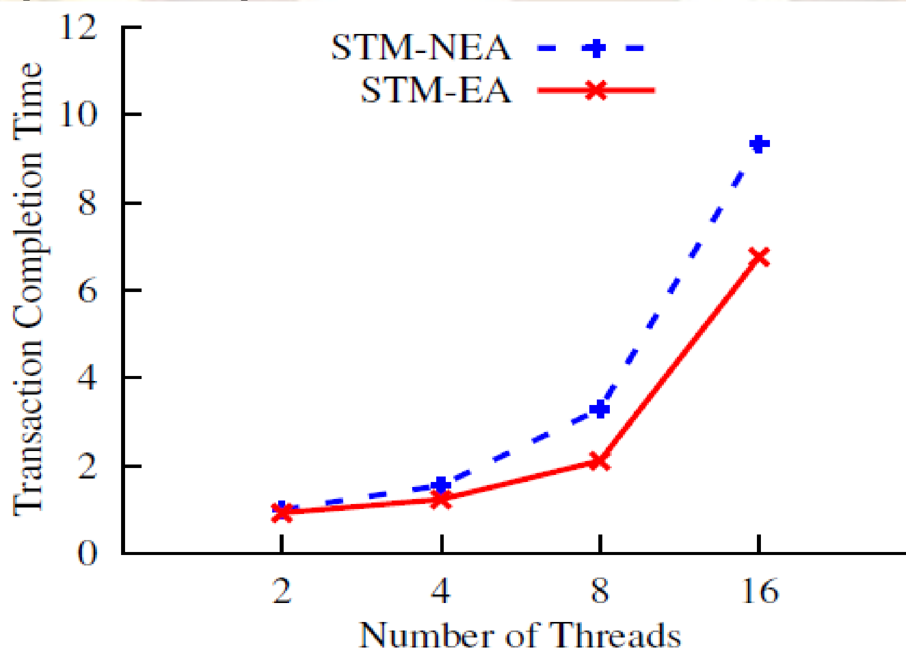
Early abort is one of the important techniques to improve the execution speed of STMs. Early abort helps improve the performance of STMs especially when the contention level is low. Early abort means eager conflict detection and aborting a transaction to resolve the conflict. It saves the computational resources which might be wasted if an uncommitable transaction continues to execute.

Early abort can be formally characterized as follows: if a STM design allows a transaction to be aborted before it reaches the commit point (where the changes made by a transaction are to be validated and made permanent), the design belongs to
the early abort category and is denoted as STM-EA; if a STM design only allows a transaction to be aborted at the commit point, the design belongs to the none early abort
category and is denoted as STM-NEA.

Early abort outperforms STM-NEA when the contention level is low. When the system contention level is low and performance is the most significant consideration then STM-EA is a good choice. STM-EA detects conflicts earlier than STM-NEA because STM-EA checks more frequently and is able to detect the conflicts right after the failures of lock acquiring or version number checking.

Performance of Early Abort Mechanisms

The graph below shows the performances of STM-EA and STM-NEA.



Graph 1: Graph showing performance of STM-EA and STM-NEA

From the above graph we can see that STM-EA performs better than STM-NEA as its transaction completion time is less. [**18**]

3.2 Approach based on Variable Granularity Access

Typical STM implementations employ a conflict detection scheme, which works with uniform access granularity, tracking shared data accesses either at word/cache line or at object level. It is well known that a single fixed access tracking granularity cannot meet the conflicting goals of reducing false conflicts without impacting concurrency adversely. A fine grained granularity while improving concurrency can have an adverse impact on performance due to lock aliasing, lock validation overheads, and additional cache pressure. On the other hand, a coarse grained granularity can impact performance due to reduced concurrency. Thus, in general, a fixed or uniform granularity access tracking (UGAT) scheme is application unaware and rarely matches the access patterns of individual application or parts of an application, leading to sub-optimal performance for different parts of the application. Variable Granularity Access Tracking (VGAT) overcomes these problems. It is a compiler based approach where the compiler uses inter-procedural whole program static analysis to select the access tracking granularity for different shared data structures of the application based on the application's data access pattern. VGAT scheme can improve the performance of STM by 21%.

Access tracking granularity has a significant impact on the performance of STM. These impacts can be classified under 3 categories namely

(a) Impact on false conflicts and hence on concurrency
(b) Impact on cache
(c) Impact on read set validation costs and lock acquire/release costs

A false conflict occurs in an STM when two different addresses are mapped to the same lock and this results in two transactions which are accessing truly disjoint data, getting falsely diagnosed as conflicting when they are not. Such false conflicts result in increased number of aborts/rollbacks and hence can impact the execution time. Based on how false conflicts occur, they are classified into two types, namely intra-conflicts and inter-conflicts. Intra-conflicts occur when two different data items getting accessed in independent transactions are mapped to the same lock due to both of them falling within the same unit granularity region which gets mapped to a single lock. Intra-conflicts occur when concurrent transactions access data which have high inter-transaction data locality.

Inter-conflicts occur when two different data items getting accessed in independent transactions are mapped to the same lock, even when they are not in the same unit granularity region. This happens due to the limited size of the lock table. Since the number of locks is often fewer than the numbers of shared data (grouped at a fixed access tracking granularity), multiple uncorrelated data items can get mapped to the same lock. This is known as lockaliasing. When such uncorrelated data items which have been mapped to the same lock are accessed in concurrently executing transactions, they cause false conflicts between the transactions, which are called inter-conflicts. The terms intra and inter refer to the fact that the data accesses are within the same memory region (of size equal to the access tracking granularity) or across two different memory regions.

Access tracking granularity also has a significant impact on the cache performance of STM applications. Use of fine grained access tracking granularity also leads to more number of lock accesses in a transaction. This can impact the performance adversely for transactions which touch a large volume of data. Every unit granularity memory region of shared data accessed in a transaction results in a corresponding programmer invisible lock access corresponding to that shared data.

Smaller the access tracking granularity, higher is the number of lock accesses for the volume of shared data accessed in a transaction. While on one hand, fine grained locking granularity improves concurrency, but it also increases the pressure on the cache.

The granularity at which accesses are tracked and conflicts are detected also has an impact on the read-set validation costs and write-set lock acquire costs. Read-set validation needs to validate each shared data read by tracking their consistency at the level of unit granularity. Similarly a transaction needs to acquire locks for each data item in the write-set at the level of unit granularity. Smaller the access tracking granularity, higher is the number of locks associated with a given volume of shared data accessed/updated in a given transaction, hence greater is the number of validation operations and lock acquire/release operations. This in turn increases the total lock operation costs and validation costs of the STM.

The three factors discussed above have conflicting requirements with respect to the access tracking granularity. Reducing the access tracking granularity reduces the intra

false conflicts, while it can end up increasing the number of lock accesses. This in turn can end up increasing the inter false conflicts, the cache pressure due to lock accesses, readset validation cost and the write-set locking cost. On the other hand,increasing/coarsening the access tracking granularity can reduce the number of lock accesses in given transaction, and can reduce the inter-

conflicts, cache pressure due to lock accesses, and read-validation and writelocking costs, but it can increase the intra-conflicts leading to a reduction in concurrency, and hence STM performance.
Quantification of the impact of these factors on STM performance would depend on the shared data access patterns of the STM application. Hence the selection of access tracking granularity that results in higher performance needs to factor in these complex and conflicting requirements, while taking into account the data access patterns of a given STM application. However most of the current STM implementations use a fixed size uniform access tracking granularity. Variable Granularity Access Tracking (VGAT) scheme overcomes all the problems associated with uniform granularity access tracking (UGAT). In VGAT the compiler selects the access tracking granularity for different shared data structures of the application based on the application's data access patterns. In VGAT the performance improvement increases with increasing number of threads. VGAT scheme also helps improve the memory performance of the application by reducing the number of lock accesses (due to the variable access tracking granularity). [**19**]

3.3 Approach based on Performance Prediction of STM

In real world scenarios all parameters affecting the performance of an STM system change for different set-ups. So it is critical to know how an STM will perform under a particular setup to ensure the fact that shifting from lock based approach to STM is really useful. So, knowing about the performance of an STM in advance is much desired. This will help us to know exactly in which situations STM should be used and can also help to improve the performance of STM.

Predicting performance of STM based on some key parameters

The factors which affect the performance of STM are:
 Mean number of restarts for a transaction (ER).
 Mean number of steps of a transaction (ES).
 Mean number of locks held by a transaction (EQ).
The two parameters ER and ES represent the quality of conflict management scheme implemented for the concerned STM. It is clear that having smaller mean number of restarts for a transaction (ER) indicates the better performance for an STM. Similarly, a smaller value of mean number of steps of a transaction ES represents that that lesser number of steps were performed every time a transaction was restarted. In other words, a smaller value of ES means that having a conflict between two or more transactions, a better contention manager would restart transaction which costs lowest. Mean number of locks held by a transaction EQ represents the total cost measurement for maintaining lock related information in the system during lifetime of the transaction. Based on these parameters the performance of STM can be predicted. The most important task is to find out how STM behaves as the number of processors goes on increasing.

Machine-learning approaches

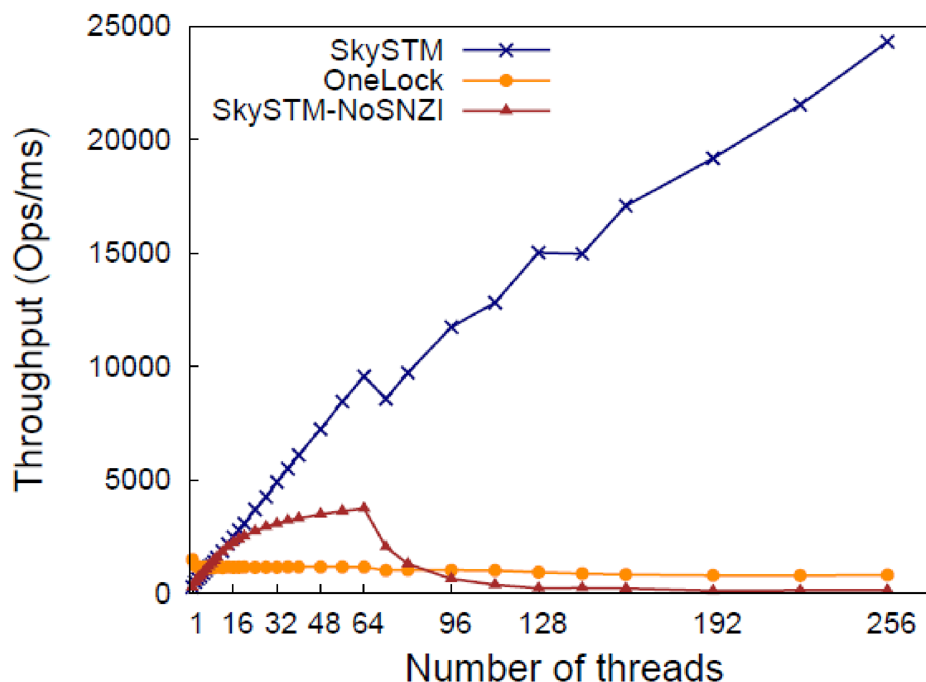
In lock based systems, a user has to take care of locking resources whereas transactional memory systems manage locks internally. Due to this fact, performance prediction approach used for a non-STM system might also be applied to an STM system. Artificial neural networks (ANNs) can be used for predicting the function performance i.e. the runtime information of functions and tasks for a certain amount of input parameters are collected. This information is used to train the network. ANN predicts the optimal task scheduling based on the training and a part of the collected data is used to compare the predicted results and the actual results for higher number of cores. Another approach named as loop cost prediction through test driving decomposes the loops, checks their dependency and systematically assigns and schedules the loops to available processors. Another approach is described for task partitioning and scheduling using MCP (modied critical path) algorithm for directed acyclic graphs (DAGs). These three approaches can be combined to evaluate the performance of STM. [**20**]

3.4 SkySTM Approach

Existing software transactional memory (STM) implementations often exhibit poor scalability, usually because of nonscalable mechanisms for read sharing, transactional consistency, and privatization; some STMs also have nonscalable centralized commit mechanisms. SkySTM eliminates all these bottlenecks. SkySTM supports privatization and scales on modern multicore multiprocessors with hundreds of hardware threads on multiple chips. It eliminates frequent updates to centralized metadata, especially for multi-chip systems, in which the cost of accessing centralized metadata increases dramatically.This scalable privatization mechanism imposes only about 4%
overhead in low-contention experiments; when contention is higher, the overhead still reaches only 35% with over 250 threads. In contrast, prior approaches have been reported as imposing over 100% overhead in some cases, even with only 8 threads.

Performance of SkySTM

The graph below shows the performance of SkySTM and some other types of STM.



Graph 2: Graph showing performance of SkYSTM and some other types of STM

So we can see form the above graph that the throughput of SkySTM is higher than that of both OneLock and SkySTM-NoSNZI. [**21**]

3.5 Approach based on Open Nesting

Nesting in STM can be of two types which are:

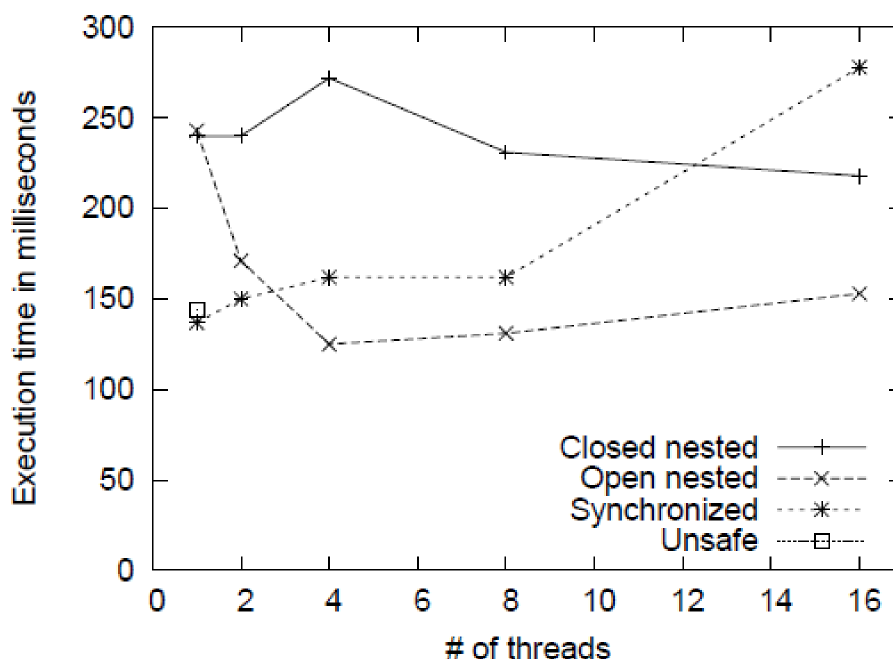
1) Open-nesting- In this case when the inner transaction commits, then the changes it has made are visible to all other transactions.
2) Closed-nesting- In this case when the inner transaction commits, then the changes it has made are visible to only the outer transaction which encloses it.

STM implementations operate by tracking loads and stores to memory and by detecting concurrent conflicting accesses by different transactions. By automating this process, they greatly reduce the programmer's burden, but they also are forced to be conservative. In certain cases, conflicting memory accesses may not actually violate the higher-level semantics of a program, and a programmer may wish to allow seemingly conflicting transactions to execute concurrently. Open nested transactions enable expert programmers to differentiate between physical conflicts, at the level of memory, and logical conflicts that actually violate application semantics. A TM system with open nesting can permit physical conflicts that are not logical conflicts, and thus increase concurrency among application threads. Open nested transactions enable programmers to obtain the precision of locks while retaining the benefits of transactional memory including serializability, composability and deadlock-freedom. Some of this power comes at the cost of increased complexity since a system cannot automatically reason at the level of application semantics. But for expert programmers, the scalability benefits of open nesting outweighs the costs. Open nesting enables expert library programmers to build software components that average programmers can compose together in a scalable way using transactions. Average programmers can use these software components without being aware of open nesting, so they still benefit from the productivity advantage of transactions but also get good performance. Open nesting scales much better than closed nesting.

So using open nesting in transactions is much better than using closed nesting.

Performance of Open Nesting

The graph below shows the performances of Open Nesting and Closed Nesting in STM.



Graph 3: Graph showing performance of Open nesting and Closed nesting in STM

We can see from the above graph that Open Nesting performs better than Closed Nesting in STM as it takes less execution time. [22]

3.6 Approach based on Commit-time Invalidation

To improve the performance of transactional memory (TM), researchers have found many eager and lazy optimizations for conflict detection, the process of determining if transactions can commit. Despite these optimizations, nearly all TMs perform one aspect of lazy conflict detection in the same manner to preserve serializability. That is, they perform commit-time validation, where a transaction is checked for conflicts with previously committed transactions during its commit phase. Commit-time validation is a strategy where a single transaction's read elements, and sometimes its write elements, are checked for consistency at commit time. While commit-time validation is efficient for workloads that exhibit limited contention, it can limit transaction throughput for contending workloads. Commit time invalidation is a strategy where transactions resolve their conflicts with in-flight (uncommitted) transactions before they commit. This is because it does not determine how many in-flight transactions will be aborted due to a transaction's commit.

Commit-time invalidation is a conflict-detection strategy in which transactional conflicts are found by comparing the memory of a committing transaction against the memory of in-flight transactions. Commit-time invalidation differs from commit-time validation in that all a committing transaction's conflicts with in-flight transactions are found and resolved before the transaction commits. Conflicts are sent to the contention manager (CM), to decide which transactions will make forward progress . The CM resolves con-
flicts by either

(1) aborting all conflicting in-flight transactions
(2) aborting the committing transaction or
(3) stalling the committing transaction until the conflicting in-flight transactions have commited or aborted.

Through this mechanism, commit-time invalidation can notably increase transaction throughput when compared to commit-time validation for contending workloads.

Commit-time invalidation supplies the contention manager (CM) with data that is unavailable through commit-time validation, allowing the CM to make decisions that increase transaction throughput. Commit-time invalidation also requires notably fewer operations than commit-time validation for memory-intensive transactions. Commit-time invalidation enables CM to make highly efficient

and informed decisions. In case of high contention commit-time invalidation is up to 3 times faster than commit-time validation.

So we can say that commit-time invalidation is much better to use than commit-time validation in STM especially in cases of high contention. [23]

3.7 Approach based on Single processor for Contention Management

From the available processors if at least one processor is used exclusively for contention management and the remaining processors are used for parallel programming purposes then it is seen that the performance of STM is much better than the case in which all processors are used for parallel programming.

4. 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.

However currently the performance of code using STM is worse than that of code using locks. This paper gives a brief overview of STM and a survey of different techniques which can improve the performance of STM.

References

[1] "Transactional Memory: Architectural Support for Lock-Free Data Structures" by Maurice Herlihy, J. Eliot B. Moss

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

[3] "Beautiful concurrency" by Simon Peyton Jones

[4] "A Survey Paper on Transactional Memory" by Elan Dubrofsky

[5] "Towards Transactional Memory Support for GCC" by Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini

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

[7] "Serializability of Transactions in Software Transactional Memory" by Utku Aydonat, Tarek S. Abdelrahman, Edward S. Rogers Sr.

[8] "Time-Based Software Transactional Memory" by Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel

[9] "Performance Evaluation of Adaptivity in Software Transactional Memory" Mathias Payer, Thomas R. Gross

[10] "Transactional Memory" by James Larus and Christos Kozyrakis

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

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

[13] "The Art of Multiprocessor Programming" by Maurice Herlihy, Nir Shavit

[14] "Transactional Memory" by Tim Harris, James Larus, Ravi Rajwar

[15] "Transactional Locking II" by Dave Dice , Ori Shalev , Nir Shavit

[16] http://tmware.org

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

[18] "Impact of Early Abort Mechanisms on Lock-Based Software Transactional Memory" by Zhengyu He, Bo Hong

[19] "Variable Granularity Access Tracking Scheme for Improving the Performance of Software Transactional Memory" by Sandhya S.Mannarswamy, Ramaswamy Govindarajan

[20] "Analytic Performance Modelling for Software Transactional Memory" by Waheed Aslam Ghumman

[21] "Anatomy of a Scalable Software Transactional Memory" by Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, Marek Olszewski

[22] "Open Nesting in Software Transactional Memory" by Yang Ni, Vijay Menon, Richard L. Hudson, Ali-Reza Adl-Tabatabai, J. Eliot, B. Moss, Bratin Saha, Antony L. Hosking, Tatiana Shpeisman

[23] "An Efficient Software Transactional Memory Using Commit-Time Invalidation" by Justin E. Gottschlich, Manish Vachharajani, Jeremy G. Siek