Master Thesis
Software Engineering
Thesis no: MSE-2009-18
September 2009

Software Transactional Memory Techniques

Principles, Design, and Implementation Trade-offs

Muhammad Nasir

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Muhammad Nasir
Address: G-Infartsvagen 3B, LGH 683, 371 41 Karlskrona, Sweden
E-mail: muhammad_nasir_atd@yahoo.com

University advisor(s):
Professor Dr. Håkan Grahn
School of Computing,
Blekinge Institute of Technology, Sweden

Internet : www.bth.se
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ACKNOWLEDGEMENTS

All praise be to Allah the most merciful and beneficent, who gave us mind to think and courage to explore the shores of knowledge. Peace be upon his prophet Muhammad and his true followers who would be a source of guidance for the people of knowledge.

I am grateful to Dr. Håkan Grahn, my supervisor, whose consistent efforts, guidance and support have made it possible for me to complete this research work.

My heart-felt love for my parents whose prayers and support has always been source of motivation for me. Moreover, I would like to thanks my elder brothers Amjad Ali and Azhar Mehmood and my younger sister, who have always been very supportive through-out my life and have been source of happiness.

I would also like to gratitude my friends Nadeem Ahmed, Mohsin Irshad, Muhammad Naveed Ahmed, Muhammad Iftikhar Nasir and Ahmed Raza Mir for encouraging and motivating me through-out my thesis work and for useful discussions. Moreover, I would like to pay special thanks to Rizwan Iqbal for his entire support and guidance.

ABSTRACT

The advent of multicore processors has put the performance of traditional parallel programming techniques in question. The traditional lock-based parallel programming techniques are error prone and suffer from various problems such as deadlocks, live-locks, priority inversion etc. In the last one and half decade, a considerable amount of the research has been carried out to achieve the synchronization among the parallel applications without using locking. One of most promising technique which has come out as a result of this research work is Transactional Memory (TM). Transactional Memory system commits the data in atomic code sequences called the transaction. Research has shown that Transactional Memory has the potential to out perform traditional locking mechanisms. In order to understand the design and implementation trade-offs of different implementations of the Software Transactional Memory, a comprehensive comparative study is required. Although some comparative studies have been carried out in the past, they were very focused in their scope and covered only few STM implementations. In this master thesis, a qualitative literature survey is conducted and the state of the art in Software Transactional Memory is presented, covering prominent approaches to date while discussing their design and implementation trade offs.

Keywords: Multiprocessor, Concurrency, Synchronization, Transactional Memory

CONTENTS

ACKNOWLEDGEMENT 3

Abstract 4

Contents 5

List of Figures 7

List of Tables 8

1. INTRODUCTION 9

1.1. AIMS And OBJECTIVES 10

1.2. RESEARCH QUESTIONS 10

1.3. OUTCOMES 10

1.4. RESEARCH METHODOLOGY 10

2. BACKGROUND 10

2.1. ACHIEVING SYNCHRONIZATION IN PARALLEL APPLICATIONS 11

2.1.1. Priority Inversion 11

2.1.2. Convoying 11

2.1.3. Deadlock and Livelock 11

2.1.4. Wait-Freedom 12

2.1.5. Lock-Freedom 12

2.1.6. Obstruction-Freedom 12

2.2. SPECULATIVE LOCKING AND SYNCHRONIZATION 12

3. SOFTWARE TRANSACTIONAL MEMORY CONCEPTS 14

3.1. TRANSACTIONS IN DATABASE PROGRAMMING 14

3.1.1. Database Transaction Taxonomy 14

3.2. MEMORY TRANSACTIONS VS. DATABASE TRANSACTIONS 15

3.2.1. Linearizability 15

3.2.2. Failure Atomicity 15

3.2.3. Isolation 15

3.2.4. Software Transactional Memory Semantics and Constructs 16

3.2.5. Atomic Block 16

3.2.6. Retry Statement 16

3.2.7. OrElse Statement 17

3.3. SOFTWARE TRANSACTIONAL MEMORY DESIGN ISSUES 17

3.3.1. Nested Transactions 17

3.3.2. Exception Handling in Transactions 18

3.3.3. Transaction Granularity 18

3.3.4. Data Update 19

3.3.5. Concurrency Control 19

3.3.6. Conflict Detection Schemes 19

3.3.7. Contention Management 20

4. SOFTWARE TRANSACTIONAL MEMORY IMPLEMENTATIONS 22

4.1. SOFTWARE TRANSACTIONAL MEMORY (STM) 23

4.1.1. Design Limitations 24

4.2. WORD BASED SOFTWARE TRANSACTIONAL MEMORY (WSTM) 25

4.2.1. Design Details 27

4.2.2. Design Limitations 28

4.3. DYNAMIC SOFTWARE TRANSACTIONAL MEMORY (DSTM) 29

4.3.1. Basic Design Features 30

4.3.2. Overview with Example 30

4.3.3. Design Details 31

4.3.4. Design Limitations 34

4.4. TRANSACTIONAL LOCKING II (TL2) 34

4.4.1. Design Details 35

4.4.2. Design Limitations....36
4.5. DYNAMIC SOFTWARE TRANSACTIONAL MEMORYII (DSTM2)....37
4.5.1. The DSTM2 Library Features 37
4.5.2. Transactional Factories....38
4.5.3. Design Limitations 40
4.6. OBJECT BASED SOFTWARE TRANSACTIONAL MEMORY (OSTM)....40
4.6.1. Design Details 41
4.6.2. Design Limitations....42
4.7. ROCHESTER SOFTWARE TRANSACTIONAL MEMORY....43
4.7.1. Design Details 43
4.7.2. Design Limitations....44
4.8. TIME-BASED TRANSACTIONAL MEMORY....45
4.8.1. Design Details 45
4.8.2. Design Limitations....46
4.9. HYBRID TRANSACTIONAL MEMORY....46
4.9.1. Design Details 46
4.9.2. Design Limitations....47
4.10. HYBRID TRANSACTIONAL MEMORY (HYTM)....48
4.10.1. Design Details 48
4.10.2. Design Limitations....49
4.11. NON-BLOCKING ZERO-INDIRECTION TRANSACTIONAL MEMORY(NZTM)....49
4.11.1. Design Details 49
4.11.2. Design Limitations....50
4.12. PHASED TRANSACTIONAL MEMORY (PHTM) 51
4.12.1. Design Details 51
4.12.2. Design Limitations....52
4.13. SIGNATURE ACCELERATED TRANSACTIONAL MEMORY (SIGTM) 52
4.13.1. Design Details 52
4.13.2. Design Limitations....53
4.14. DRACOSTM....54
4.14.1. Design Details 54
4.14.2. Design Limitations....55
4.15. MCRT-STM....55
4.15.1. Design Details 55
4.15.2. Design Limitations....56
5. SUMMARY AND DISCUSSION....57
5.1. Synchronization Approaches....57
5.2. Concurrency Control....57
5.3. Transaction Granularity....58
5.4. Update Strategy....58
5.5. Contention Management Strategies....59
5.6. Isolation and Nested Transactions....59
5.7. Hybrid Transactional Memory Systems....60
5.8. Validity of the Study....61
5.8.1. Reliability and Trustworthiness of the research Work....61
5.8.2. Validity of the research work 61
5.8.3. Rigorousness and Quality of research work....61
6. FUTURE CHALLENGES....62
6.1. Achieving Strong Isolation....62
6.2. Nested Transactions....62
6.3. Integration Overheads....62
6.4. Performance Comparison Studies....62
6.5. Transactional Memory Benchmarks....63
6.6. Transactional Memory Debugging and Testing Tools....63
7. CONCLUSION....64
8. REFERENCES....67

LIST OF FIGURES

Figure 2.1 A shared variable with Lock…….....11
Figure 2.2 Example of Speculative Lock Elicitation…….....13
Figure 3.3.1 Atomic Block……...16
Figure 3.3.2 Retry Statement……....17
Figure 3.3.3 The OrElse Statement…….....17
Figure 4.1.1 A Shavit and Touitou's STM Shared Memory Model…….....24
Figure 4.2.1 The Conditional Critical Region ……25
Figure 4.2.2 The Structure of Atomic Block……26
Figure 4.2.3 The WSTM Data Structure…….....27
Figure 4.2.4 The Hash table update operations performance……...29
Figure 4.3.1 Encapsulating the Object inside TMObject…..……..31
Figure 4.3.2 Opening a transaction in Write mode…....31
Figure 4.3.3 The Transactional Object Structure……..32
Figure 4.5.1 The Transactional Object Structure…….....39
Figure 4.5.2 The Backup operation in shadowfactory…….……....39
Figure 4.5.3 The Recovery operation in shadowfactory…….……40
Figure 4.6.1 The OSTM data structure…….……41
Figure 4.7.1 The RSTM data structure…….……44
Figure 4.9.1 The Hybrid TMObject …..…….....47
Figure 4.11.1 The NZTM data structure…….……50

LIST OF TABLES

Table 4.1 The Brief History of STM……22
Table 4.1.1 The Basic Design features of STM……..23
Table 4.2.1 The Basic Design features of WSTM……..25
Table 4.3.1 The Basic Design features of DSTM……...29
Table 4.3.1 The Basic Design features of TL2……...34
Table 4.5.1 The Basic Design features of DSTM2…….37
Table 4.6.1 The Basic Design features of OSTM……..40
Table 4.7.1 The Basic Design features of RSTM……...43
Table 4.8.1 The Basic Design features of Time-based STM…….45
Table 4.9.1 The Basic Design features of Hybrid TM……...46
Table 4.10.1 The Basic Design features of HyTM……...48
Table 4.11.1 The Basic Design features of NZTM……..49
Table 4.13.1 The Basic Design features of SigTM……...52
Table 4.13.2 The Signature in SigTM……..53
Table 4.14.1 The Basic Design features of DracoSTM……...54
Table 4.15.1 The Basic Design features of McRT-STM…….55
Table 7.1 The Comparison of Design Features of TM Systems…….....65

1. INTRODUCTION

Now-a-days, we are living in an era of multi-core processor systems where we need robust and scalable, parallel applications to utilize the full power of multi-core architecture systems. Parallel applications share data, and therefore need synchronization among multiple processes. The locking mechanism and mutual exclusion have been used to achieve this synchronization. However, it may create performance bottle-necks, and it is more time consuming and vulnerable to the errors. In the last two decades, a considerable amount of research has been carried out to achieve synchronization in the parallel applications without using locks. One of most promising technique that has come out as a result of this research work is Transactional Memory (TM) [3][34][59].

Transactional Memory allows data sharing without using locking mechanisms. The Transactional Memory can be implemented in software as well as in hardware. It commits the data in atomic code sequences called transactions [26]. Before committing a transaction it checks whether the data which it read was not outdated or changed by another transaction. When there is a read/ write conflict, it aborts the transaction and rolls back and process the transaction again until it has no conflict. So in this procedure, Transactional Memory maintains a log for each transaction so that it would go back to its previous state. Software Transactional Memory has a flexible framework for executing parallel operations with contention manager for resolving the conflicts and load balancing. Transactions have been used a lot in databases since long ago and it do not suffer from resource starvation and deadlocks [27].

Research has shown that the Transactional Memory has the potential to out perform traditional locking mechanism. The idea of Transactional memory was first introduced by Herilhy et al. [34] based on the hardware approach in 1993. Later on in 1995, Shavit and Touitou [59] came up with the idea of Software Transactional Memory. Now-a-days, several Hybrid Transactional Memory (HTM) implementations have also been released [44]. However, this research report is based on study of Software Transactional Memory as well as some of the Hybrid Transactional Memory implementations. There are many implementations of Software Transactional Memory which have been developed over the period of time. Some of the Transactional Memory techniques developed are the Dynamic Software Transactional Memory (DSTM) [36], the Fast Software Transactional Memory (FSTM) [23][24], the Lock Based Software Transactional Memory (TL2) implemented by a research group at Sun Microsystems Laboratories [16], and the Light Weight Library (LibLTx) a C language implementation of the Software Transactional Memory by the Robert Ennals [20].

In order to understand the design and implementation trade-offs of different implementations of the Software Transactional Memory, a comprehensive comparative study is required. Such studies will help practitioners and researchers to develop better Software and Hybrid Transactional Memory systems. Although some comparison studies have been carried out in the past but those were very focused in their scope and covered only few STM implementations. A comparative study has been done by the Marathe and the Scott [46] in which they compared the FSTM, the DSTM and a hash table based STM system design. Similarly, a book has been written by Larus and Rajwar [41] in which overview of all the current STM implementations has been discussed latest by mid 2006. However, this master thesis discusses total fifteen Transactional Memory systems out of which eight TM systems are new and have not been discussed by Larus and Rajwar [41]. In other words, this is a state-of-art report describing the current research front in the area of software transactional memory.

1.1 Aims and Objectives

The main objective of this thesis project is to identify and discuss design issues of the Software Transactional Memory (STM) systems so that it may help in understanding and developing better STM systems. Following sub-objectives are achieved in this master thesis to support the main objective.

1. Identifying and discussing the problems in traditional locking mechanism in parallel applications.
2. Analyzing and discussing the need for STM systems.
3. Discussing the early STM systems and problems associated with them.
4. Identifying and analyzing the modern STM systems.

1.2 Research Questions

Following research questions are addressed in this thesis project.

1. Which approaches exist to support software-based transactional memory?
2. What are the designs and implementations trade-offs of various approaches?

1.3 Outcomes

The outcomes of the thesis project are as below;

1. A state-of-art report describing the current research front in the area of software transactional memory systems.

1.4 Research Methodology

The qualitative research methodology is adopted in this master thesis. The qualitative research methodology is chosen because in the qualitative study a large pool of software and hybrid transactional memory implementations can be analyzed and discussed. On the other hand, in an empirical study not many systems can be tested, compared and discussed due to the increased complexity of the work and limitation of the time. A comprehensive qualitative survey is conducted to identify and discuss the current approaches which support Software and Hybrid Transactional Memory systems and their design and implementation trade offs.

2. BACKGROUND

In this chapter we will look at different generic approaches to achieve synchronization in parallel applications and would briefly discuss the problems associated with them.

2.1 Achieving Synchronization in Parallel Applications

Parallel applications share data and the traditional mechanism to achieve synchronization has been a locking mechanism. Locking uses mutexes, semaphores etc. to ensure mutual exclusion in resource sharing. Figure 2.1, shows a code in which the variable counter is accessed exclusively using a locking mechanism.

```
Lock ()
{
        // shared variable counter
        counter++;
}
Unlock ()
```

Fig. 2.1. A Shared variable with Lock.

Locking ensures mutual access to the shared data but it creates a bottleneck for other threads or parallel processes. Other processes have to wait until the thread which is holding the lock completes its execution. Blocking a process can lead to the following problems [34].

2.1.1 Priority Inversion: Priority inversion takes place when a lower priority process is holding a resource which is required by a higher priority process, which makes the higher priority process wait until resource is released.

2.1.2 Convoying: Convoying takes place when a process holding a lock is re-scheduled due to the different reasons, such as, if the process has consumed its processing quantum of time and yet not completed its execution, may be due to the page fault or due to some other interference. Meanwhile other threads waiting in queue to acquire the lock will not be able to progress ahead until this thread release the lock. Even if the lock is released, it will take some time to re-set the queue, which as a result will slow down the processing.

2.1.3 Deadlock and Livelock: A deadlock is a situation where one or more processes are waiting for each other to release a resource and this situation lead to a circular chain of wait with no progress taking place on part of each process. A good example of a deadlock can be explained in a client-server database application.

Let suppose a client application has acquired lock over a database table and it requests for an exclusive access to another table which is locked by another client application and that client application is waiting for the first client application to release the table, it is holding. In this way both clients are waiting for each other to release locks over the tables they are holding while none is doing so, which leads to deadlock situation.

On the other hand, livelock does not wait for anything but keeps on processing based on the erroneous input. A good example of the livelock can be endless loop. It is analogous to the deadlock that no real progress is made ahead yet differs in a sense that no process is blocked or

waiting for any resource. A daily life example of the livelock can be two people meeting in the corridor. Both change their position to give way to the other person, but both are unable to make any real progress because both move the same side at the same time.

The error prone nature of the locking mechanism also creates performance bottleneck. Therefore researchers focused their efforts towards non-blocking algorithms, techniques and data structures. Non-blocking approaches allow sharing data without acquiring a lock over the data.

A system is non-blocking if suspension of one process may not stop other processes to progress ahead [25]. In other words, in locking mechanism, if one process is having a lock over a shared resource and it is stuck, then the other process will have to wait until this process release the lock, hence locking mechanism blocks other process to progress ahead. Non-blocking synchronization approaches try to avoid mutual exclusion. Non-blocking design property of a parallel system leads to the high throughput and better performance by avoiding deadlocks, live-locks and priority inversion and this is the reason of shifting from locking to lock-free and non-blocking approaches for synchronization in parallel applications. Basically there are three generic properties of non-blocking systems, based on which these are classified, Wait Freedom, Lock freedom, obstruction freedom [25].

2.1.4 Wait-Freedom:

[25] This property of the non-blocking system allows each process to progress without taking the contention into the context. Wait-freedom infact ensures that there would not be any starvation. However practically its not possible to develop efficient wait-free algorithms in parallel applications as the memory cost increases linearly with the number of processes. Therefore not much attention has been paid in this regard.

2.1.5 Lock-Freedom:

[46] The Lock-freedom ensures that multiple processes run at the same time but only one process goes ahead and completes its execution within finite number of execution time. The rest of the processes have to wait. The Lock-freedom ensures deadlock prevention but suffers from starvation. In lock-freedom, every process try to complete its execution but when it identify that original values have been changed by another process then it rolls-back and starts its processing again based on new values.

2.1.6 Obstruction-Freedom:

[36] An algorithm is obstruction-free if it allows completing a process only if it is not obstructed by another process. This is a very weak property of a non-blocking algorithm as it is hardly possible that another process will not contend the currently executing process. Furthermore, Obstruction-free algorithm introduces the problem of livelock [36]. Therefore to avoid livelock and deadlock, roll-back is used. Moreover, a contention manager can be used to decide which processes have higher priority and based on the priority level higher priority process is allowed to execute while lower priority processes are obstructed.

Wait-freedom seems to be a more efficient design property of Non-blocking synchronization algorithms, but practically obstruction-freedom leads to more simplicity and design tradeoffs [46]. So in practice obstruction-freedom is more efficient than lock-freedom and wait-freedom.

2.2 Speculative Locking and Synchronization

Another approach to achieve synchronization in Parallel applications is speculative locking of shared data. This technique was introduced by Rajwar and Goodman [52] with name of SLE (Speculative Lock Elision). SLE is hardware based synchronization technique for multiprocessing. The hardware dynamically monitors the synchronization operations and detects

if the synchronization is found un-necessary then it is eliminated. The detection of un-necessary synchronization is achieved by cache coherence mechanism. If hardware found that synchronization is required then it performs a recovery operation and explicitly lock is acquired. See the example code below in figure 2.2.

Process No. 1

```
lock(employee_struct.mutex)
Struct Struct_Employee;
If (Struct_Employee.Salary >1000)
{
Struct_Employee.Bonus=10/100*1000;
}
unlock(employee_struct.mutex)
```

Process No. 2

```
lock(employee_struct.mutex)
Struct Struct_Employee;
If (Struct_Employee.Salary >1000)
{
Struct_Employee.Rank=”Manager”;
}
unlock(employee_struct.mutex)
```

Figure 2.2. Example of Speculative Lock Elicitation.

In this example two threads are working on the shared data Struct_Employee but as they are modifying different properties of shared data Struct_Employee hence lock is not required to synchronize the operation. Furthermore, if there is a conflict in synchronizing the parallel operation then operation is rolled back and a proper lock is acquired to complete the process. The conflict detection and lock acquisition is done with the help of the Cache Coherence mechanism. SLE has many similarities to Software Transactional Memory which would be discussed in chapter 3. The problem arises with SLE when data conflicts take place and locks are acquired which as result creates bottle neck and other similar problems which are associated with typical locking mechanism.

3. SOFTWARE TRANSACTIONAL MEMORY CONCEPTS

In this chapter we will discuss the basic artifacts and constructs of Software Transactional Memory. A memory transaction is basically a finite sequence of instructions, preserving the serializability and atomicity properties [34]. These properties will be discussed later in this chapter, in detail.

3.1 Transactions in Database Programming

Transactions have long been a part of database programming while their importance is now realized in parallel programming [41]. Database Systems allow multiple queries to run in parallel and it maintains concurrency with consistency. In other words, if a concurrent transaction is left in an illegal state, then it is aborted and rolled back. In the last few decades it is realized that a programming model which has been successfully used by databases can also be utilized in parallel programming.

The main building block in database execution environment is Transaction. A transaction execution procedure defines the flow of the database computation. A transaction is a set of instructions which executes atomically and it may interact with other transactions but its results are separate from other transactions. Multiple transactions run over a database and produce accurate and consistent results.

The transactions have a great deal of abstraction in itself which makes them simpler to understand and develop as atomic. Abstraction can be achieved in two ways i.e. data abstraction and process abstraction. Abstraction implies, hiding the implementation details of a compound data object or a process object from its usage details [7]. The data used by a transaction is not visible to other transactions and it executes independently but it may interact with other transactions if required through a proper interface.

Transactions in parallel programming are similar to database system transactions [41]. Since today is an era of multi-core processors, parallel programming can unitize the old concepts of database transactions which have long been playing a vital role in the concurrent programming environment of database systems.

3.1.1 Database Transaction Taxonomy

A transaction is a set of instructions which are basically one unit. It has proper start and end with consistent results. A particular database transaction has four basic properties: Atomicity, Consistency, Isolation and Durability also known as ACID, to ensure that transactions take place with correctness.

1. Atomicity: Atomicity means that a transactions either successfully completes or it fails and roll back. On successful completion it commits its results and on failure it aborts.

2. Consistency: Consistency means that every transaction has the same view of the data. When a transaction starts the data which it reads remains on-changed until it completes its operation and it commits its results. The same updated data is visible to all the transactions.

3. Isolation: Isolation is a property which makes sure that each transaction can execute in parallel independently and its internal execution and data should be isolated and hidden from other transactions and failure of one transaction may not affect the result of other transactions.

4. Durability: Durability implies that when a transaction completes, its results should be committed and should remain permanent and the same results should be visible to all other transactions.

3.2 Memory Transactions vs. Database Transactions

In this section we will look at the differences and similarities between database transactions and a typical memory transaction.

Database transactions are usually very long and may take quite long time to complete and may consist of numerous instructions [12]. A typical memory transaction usually contains low number of instructions and would execute in faster.

In a database transaction, data resides on hard disk and it makes the data retrieval slow and hence total execution time of the transaction is longer. On the other hand, memory transactions reside in the main memory and the cache, hence executed quickly.

Moreover database transactions are often nested while memory transactions are usually not nested [12]. Database transaction are based on the properties discussed in section i.e. Atomicity, Consistency, Isolation, Durability while memory transactions support *linearizability* and atomicity [34]. However if *linearizability* and atomicity is achieved then consistency and Isolation is automatically attained. *Linearizability* is more explained in next section.

3.2.1 Linearizability

The concept of *linearizability* in parallel computation was first introduced by Herlihy and M. Wing [35]. They said that a parallel computation is *linearizable* if its results are equal to its sequential version. In other words if a parallel or concurrent computation is run sequentially and it comes up with same results then it has the property *linearizability.*

Linearizability is basically a correctness criterion for concurrent transactions which makes sure that every running processes, and runs instantaneously and has some valid pre and post conditions which give meaning to existence. In other words every process is invoked by some valid events or by other processes and it runs in a single point in time, and it executes and ends with some valid results and response. Linearizability creates an illusion that each concurrent process runs instantaneously [35]. Linearizability is also a non-blocking property and it supports the parallel computation and concurrency.

3.2.2 Failure Atomicity

Failure atomicity ensures that a transaction either successfully completes its execution and commits its results or entirely aborts. Failure atomicity play very important role in order to make the system execution consistent. In other words, if a transaction aborts all the changes made by a transaction are reverted back.

3.2.3 Isolation

Isolation, as discussed in section 3.1.1, implies that operation of one transaction may not effect other transactions. Isolation is more categorized in to strong isolation and weak isolation [9].

Inside an STM-based system there can be two types of operations, i.e., transactional and non-transactional. So, non-transactional operations may have a negative effect over transactional operations which may lead to data races and inconsistencies.

Strong Isolation implies that data access is always restricted to a transaction only. However, this assumption is not much practical as transactions sometime require data which is not available inside a transaction. The idea behind strong isolation is this that data confliction should be minimum.

The concept of *Weak Isolation* believes that data can be accessed inside a transaction as well as outside a transaction. But as data inside a transaction is atomic and consistent but not necessarily outside transaction. So, in this case data outside a transaction can be formed in to a transaction and thus a transaction can communicate with other transaction through an interface hence reducing the possibility of data race and helps in conflict management.

3.3 Software Transactional Memory Semantics and Constructs

In this section we will discuss the basic building blocks of Software Transactional Memory and their usage details.

3.3.1 Atomic Block

A memory transaction is in fact contained inside a block, called an Atomic Block. This Atomic Block defines the boundaries of a memory transaction [33]. Figure 3.3.1 shows an example of memory transaction.

```
Atomic
{
        int x, y, radius;
        CalculatePieVal (int radius, int x, int y);
}
```

Figure 3.3.1. Atomic Block.

A transaction executes with atomicity and serializability as discussed in section 3.2. The memory transaction only exposes their execution outcome to the system and performs its processing in isolation, hence hiding the implementation details from rest of the system. In other words, atomicity leads to abstraction in memory transaction. The memory transaction has states like *running*, and if it successfully completes then it *commits* its results to main memory after identifying that no data conflicts are detected.

3.3.2 Retry Statement

In a software transactional memory system, multiple transactions execute concurrently and there is no specific order in which the transactions should update their results. Moreover if one transaction updates the result, other transactions may have to restart and execute again based on updated values. So in this context there should be some mechanism which could re-execute a transaction. Harris et al. [32] suggested a retry statement to synchronize the transactions in time. A Retry statement aborts a transaction and then re-executes it. Figure 3.2.2 shows an example of use of the retry statement.

```
Atomic
{
        int x, y, radius;
        CalculatePieVal (int radius, int x, int y);
        if (OriginalVal.updated==True)
        {
                retry;
        }
}
```

Figure 3.3.2. The Retry Statement.

3.3.3 OrElse Statement

OrElse is another conditional control statement introduced by Harris et al. [32]. For example, if there are two processes within a transaction, then the OrElse statement can be very useful in order to coordinate them. Figure 3.3.3 illustrates the use of the OrElse statement.

```
Atomic
{
        //Process A
        {
                obj1.CalculateMatrixMulti();
        }
        orElse
        //Process B
        {
                obj2.CalculateMatrixMulti();
        }
}
```

Figure 3.3.3. The OrElse Statement.

In the figure 3.3.3, the Process A is executed first and if it retries then Process A is left behind and the control goes to the Process B. If Process B also fails and retries as well then whole atomic block is executed again from the beginning [32].

3.4 Software Transactional Memory Design Issues

In this section we will discuss more STM related design issues and their trade-offs on STM systems. Earlier on, in section 3.2.1 and 3.2.2 we discussed basic properties of memory transaction i.e. *Linearizability* and *Isolation*.

3.4.1 Nested Transactions

A transaction is nested when it contains one or more transactions inside it. The behavior of the inner and the outer transaction can be related with each other in many ways. Initially, outer transaction initiates the inner transaction, then, depending on the how control shifts between two transactions, *Nested Transactions* are divided into many categories.

If an inner transaction aborts and it causes the outer transaction to abort, then this type of the nested inner transaction is called a *Flattened Transaction* [41]. In the flatten transactions, changes

are not committed until the outer transaction commits, its changes. Flatten transactions are simple in implementation. As *Flattened Transactions* cause the outer transaction to abort, this may decrease the overall performance of the system [50].

Two other categories that are different from *flatten transactions*, are *Closed* and *Opened Transactions* [41]. A *Closed Transaction* aborts without terminating its outer transaction. If an inner transaction commits or aborts then control is passed to outer transaction. In the case, when *Nested Transaction* commits, its updates are visible to the outer transaction or surrounding transactions. Having said this, a non-surrounding transaction can only see the changes when the outermost transaction also commits its changes successfully.

In *Open Transactions,* changes committed by the inner transaction become visible and remain permanent to all the running transactions in the system; even if the outer parent transaction is still in process or it may fail or abort.

However *flattened nested transactions* actually threaten the isolation property of the transaction. Isolation implies that if one transaction fails or abort it may not affect other transactions. Similarly, in *open transactions* changes made by inner transactions may not be visible by the outer transactions, and vice versa as it is against the essence of isolation which emphasizes that inner data of transaction may not be visible to other transactions unless the transaction commits completely with success.

3.4.2 Exception Handling in Transactions

Exceptions are meant to prevent the whole software system from crashing. Exception handling is also a part of a STM system. Exception handling inside a transaction has two design issues, i.e., while exception occurs, the transaction may save its results and then quit the system or quit without saving results. Moreover the transaction may re-execute or just quit simply. Committing the inconsistent results can be fault prone, so more appropriate approach can be that when exception occurs, the transaction may abort and execute again.

3.4.3 Transaction Granularity

Transaction granularity refers to the storage space on which the STM system detects conflicting access to data [41]. Granularity can be implemented at an *object level*, *word level,* and *block level*. All these levels refer to the shared resource which is infact different amount of data.

An ideal STM system maintains meta-data about each running object in order to track them, and avoid any possible conflicting access and to maintain consistency [41]. There are two ways to maintain Meta-data; first that meta-data should be part of each object which is at the *object level* granularity, secondly there can be a separate data structure where meta-data, about all the running process, and data may be maintained in a separate memory block, this approach is called *block level* granularity.

Block level granularity offers more precise sharing of resources than *object level granularity* but mapping meta-data from memory to another data-structure is another overhead besides keeping a separate data storage. *Object level granularity* is more understandable to the programmer than *block level granularity* as objects are more visible to programmer than memory blocks. Aggregate data structure such as arrays are very helpful in block level fine grained sharing of the data, as aggregate data structure can be portioned logically [41].

3.4.4 Data Update

When a transaction completes successfully it updates the original values with updated values. Based on the update strategy there are two approaches, i.e., *Direct Update* and *Deferred Update* [41].

Direct update implies that a transaction modifies the original value directly. However the system maintains the original value so that meanwhile if the transaction aborts then it is able to roll back the changes made by the transaction. In order to keep record of the original value, an STM system maintains a log of the activity. The cost of aborting is high in case of *direct update* as every transaction which read the updated value will have to roll back.

Another approach to update the values is *Deferred Update.* In this approach a running transaction maintains a separate copy of the updated values and when it successfully completes its execution then it copies all the values from temporary copy to the original memory locations. However, in this approach, maintaining a separate copy of variables is another overhead besides copying from temporary locations to original locations [41]. As the cost of a roll back in direct update is more than a successful commit, so deferred update seems more performance oriented than direct update.

3.4.5 Concurrency Control

Another design issue of the STM system is *concurrency control* which is an integral part of an STM, in order to facilitate the concurrent execution of the processes. Concurrency control is equally important both in *direct update* as well as in *deferred update*.

When more than one transaction try to access a shared memory then a conflict arise. As a result, in order to resolve the conflict one of the transactions either has to wait or it has to abort its processing. The concurrency control is always based on three events, which occur in a sequence. First of all, a conflict occurs and a STM system detects that conflict, and in the third step it resolves the conflict. Basically there are two categories of concurrency control, i.e., *pessimistic concurrency control* and *optimistic concurrency control* [41].

Pessimistic concurrency control is based on the fact that all three events i.e. *conflict occurrence, conflict detection,* and *conflict resolution* take place at the same point in execution. This means that as soon as a conflict arises it is detected by the STM system and resolved. *Pessimistic concurrency control* gives exclusive access of a shared object to a transaction. This exclusive access can be acquired while aborting other transactions.

However, *optimistic concurrency control* assumes that *conflict detection* and *resolution* take place after conflict arises. In this case of concurrency control multiple transactions are allowed to access a shared resource. *Optimistic concurrency control* detects and resolves the conflict when a transaction commits its results. Conflict resolution is implemented by aborting conflicting transaction or by putting them in wait queue. The *Optimistic* approach of concurrency control is suitable when conflict does not arise too frequently. As result more transactions can complete their execution with acquiring an exclusive access of shared resource [41]. Another dimension of concurrency control is blocking and non-blocking synchronization approaches which has already been discussed in chapter 2 in detail.

3.4.6 Conflict Detection Schemes

The Conflict can be detected at different stages of a running transaction and every approach has its relative advantages and disadvantages and potential influence on the performance of a STM.

All the approaches which detect the conflict before commit falls in to the category of *early conflict detection*. While detecting conflict on commit is *late conflict detection*. There are basically three approaches as discussed by James Larus and Rajwar [41] below;

- ***Conflict Detection before Validation and Commit:***

In this approach, a conflict is detected while a transaction tries to access a shared resource or memory.

- ***Conflict Detection on Validation:***

Another way to detect conflicts is to check regularly whether any transaction has accessed a conflicting resource previously. The checking of transactions regularly is a part of the validation strategy.

- ***Conflict Detection on Commit:***

Another point where a conflict can be detected is on commit. While a transaction try to commit its results it may check whether it has read any values which have been modified and no longer are valid.

Early conflict detection reduces the amount of computation which goes in aborting a transaction. On the other hand, there are situations where a transaction could have completed its processing if conflict would have not been detected earlier. Consider the problem where there is a transaction T_1 and T_2 and both have a conflict with Transaction T_3 over two different objects. Meanwhile T_2 aborts as soon as it identify that it has conflict with T_3 .Similarly T_3 aborts because it had conflict with T_1. Now in this situation if conflict would have been identified late then there was chance that T_1 and T_2 could complete their processing successfully without aborting.

However late fault detection is costly, due to that fact that when a transaction, which has developed a conflict, and is suppose to be aborted at the end, keeps on executing and when late fault detection aborts it, it has consumed a lot of processing power of system and resources, which goes in waste. In this case, if conflict would have been detected earlier then the transaction will not have consumed resources of system.

In order to detect that a value has been modified, there should be some kind of mechanism which could identify it. One way of doing this is version number management. A transaction can check the version number of the data object which it read by comparing it with the existing data objects' version number and hence can avoid any conflict. The version number is basically a counter which is incremented on every modification to the data object.

3.4.7 Contention Management

In order to resolve the conflicts between running transactions in STM based system, ideally there should be a contention manager. A contention manager can resolve a conflict either by aborting a transaction or putting one transaction in a wait queue. A dynamic contention manager should incorporate several contention management policies for different situations in order to re-act efficiently. Similarly an ideal contention manager should ensure forward progress. Transactions should execute and system must proceed ahead and it should not be stuck in a state where it can not proceed further.

There are several contention management policies to resolve the conflicts; most of them have been discussed in detail by William Scherer and Michael Scott [58]. Some of the better performing contention managers, based on their work are discussed below;

1. Polite Manager

This contention manager uses exponential back-off for a certain amount of time, called spinning time, for resolving the conflict. The spinning time is calculated by the formula shown in figure 3.4.7.1. Polite manager, counts the number of times, a transaction has been trying to access an object. After a particular number of access attempts, the *Polite* manager, aborts all the conflicting transactions and give access to the competing transaction.

Spinning Time = 2^{n+k}

Where

n = Number of times, a conflicting access took place for an object
K = Architectural tuning constant
m = Maximum number of spinning rounds

Ideal values for m = 22, and k = 4

Figure 3.4. The Exponential Back-off formula.

2. Karma Manager

The karma manager resolves the conflicts among competing transactions by considering the amount of the data processed by a particular transaction. It prefers to abort a competing transaction which has just started its processing compare to that transaction which is in its final stage of execution. However finding the amount of data processed by a transaction is difficult to judge.

The Karma manager uses number of objects opened by a transaction as measure of data processing. And this number is maintained in a counter with every transaction. If transaction commits then this counter is set to zero. However on abort of a transaction this counter is not changed, which gives a higher priority to this transaction to complete, next time.

3. Kindergarten Manager

Kindergarten manager is based on the conflict resolution strategy for Dining Philosophers problem [11]. In this scenario, all the transactions try to access the shared object. Each transaction maintains a hit list of the enemy transactions which are competing for the access of a shared object. Initially hit list is empty. Hit list contains the identity of the transactions which have been aborted in favor of a transaction.

Whenever a transaction has a conflict accessing an object, Kindergarten manager, aborts the transactions which are in the hit list of this transaction. If it is not in the hit list, then it is added to it, and Kindergarten manager, backs off for specific interval of time to give them a chance to complete their execution. If the transaction still can not proceed ahead, then Kindergarten manager aborts its own transaction and kick-off a restart.

4. Timestamp Manager

Timestamp manager records the starting time of each transaction. And it aborts the enemy transactions which has newer time stamp. Otherwise, Timestamp manager sets a flag on enemy transaction, considering it as dead transaction. Timestamp manager checks the flag after some interval of time, if it is there then it kills that transaction. However, active transactions clear their flag. Timestamp manager is considered to be a fair contention manager.

4. SOFTWARE TRANSACTIONAL MEMORY IMPLEMENTATIONS

In this chapter we will discuss different implementations of Software Transactional Memory systems, with their design and implementation trade offs. In the beginning, Transactional memory was supported only in the hardware, but in 1995, Shavit and Touitou introduced Software Transactional Memory. Since then, many STM systems have been introduced. Software Transactional Memory has many advantages over the hardware based Transactional Memory Systems [41]:

- STM is easy to implement, more flexible and diverse.
- STM is more evolvable and easy to modify than a hardware approach.
- STM memory can be made a part of a programming language which increases its usability.
- STM has less internal limitations as compare to a Hardware approach, e.g., limited size of cache

Though till now there exist many implementations of software transactional memory but we would focus on Software and Hybrid Transactional Memory System implementations, show in table 4.1 shows;

Table 4.1. The Brief History of STM.

Year	Software Transactional Memory Systems	Synchronization
1995	STM (Shavit, Touitou)	Lock-free
2003	WSTM (Fraser, Harris)	Lock-free
2003	OSTM (Fraser)	Lock-free
2003	DSTM (Herlihy et al)	Obstruction-free
2006	RSTM (Marathe)	Obstruction-free
2006	Time based STM (Riegel)	Obstruction-free
2006	DSTM 2 (Herlihy OOPSLA)	Obstruction-free
2005	McRT-STM (Saha et al)	Lock-based
2006	TL2 (Dave Dice, Ori Shalev, Nir Shavit)	Lock-based
2007	DRACO STM	Lock-based
2007	NZTM (TABBE)	Hybrid TM
2006	HyTM (Damron)	Hybrid TM
2006	HybridTM (Kumar)	Hybrid TM
2007	PhTM (Lev)	Hybrid TM
2007	SigTM (Chi Cao Minh)	Hybrid TM

4.1 Software Transactional Memory (STM)

The STM system proposed by Shavit and Touitou [59] identifies and tries to get access of all the memory locations which it would need for a particular transaction. The basic unit of memory, on which this system is based on, is word. In other words, the transaction granularity is at word level. The basic design features of Shavit and Touitou's STM are shown in table 4.1.1. When a transaction holds the control of memory word, it becomes the owner of that memory word. Ownership information is stored separate beside the actual data as shown in figure 4.1.

Table 4.1. The Basic Design features of STM.

STM	
Synchronization	**Non-blocking (Lock-freedom)**
Concurrency Control	**Pessimistic**
Conflict Detection level (Granularity)	**Word**
Update Strategy	**Direct Update**
Conflict Detection	**Early**
Conflict Management Strategy	**Helping**
Nested Transaction Type	**N/A**

The ownership record either has a valid address of the owner or it has a Null value which indicates that no transaction owns the data. Although each transaction can access shared data, but one memory block can be owned by only one transaction at a time. This implies that only that transaction can make changes to that memory block. The system ensures acquiring of memory objects in increasing order which avoid the possibility of deadlocks.

If a transaction fails to acquire ownership of a memory object then it aborts and releases the memory locations which it already has acquired. If a transaction manages to take all the desired memory locations then it completes its execution and updates the results without the risk of rollback. This implies that system uses the *Direct Update* approach.

The system ensures non-blocking access with forward progress. No matter some transactions will fail and abort but at least one transaction will manage to complete its execution. Concurrency control is *pessimistic, i.e.,* system assumes that conflict occurrence, conflict detection and resolution events take place in separate course of time. The System has an early conflict detection mechanism and uses a concept called *helping* for conflict resolution, which implies that if a transaction can not proceed further then it should abort and help other transaction in completing their execution.

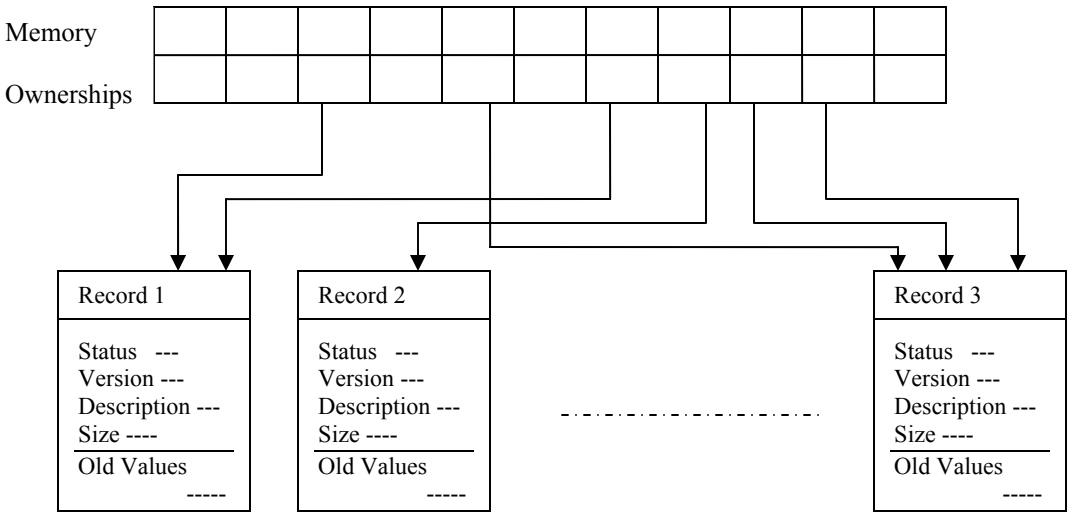

Figure 4.1.1. The Shavit and Touitou's STM Shared Memory Model [59].

4.1.1 Design Limitations

One of the major draw back which Shavit and Touitou's system have is its *helping* mechanism for conflict resolution. This conflict resolution technique is based on the concept that if a transaction can not proceed ahead due to some conflict then it should *help* other transaction in their completion. In this case two threads are executing the same transaction. This means that if transaction X found that it is conflicting with transaction Y then transaction X makes update on behalf of transaction Y.

Shavit and Touitou originally adopted the concept of *helping* from Greg Barnes [5], who gave the concept of *recursive helping.* In recursive helping a transaction being helped may be helping another transaction. Moreover, consistent *helping* can deteriorate the performance by unnecessary conflicts [59]. However, in Shavit's STM, *helping* is restricted to a specific level only; even then it leads to a great level of complexity, as it exposes data to one or more threads [8]. However, later conflict resolution strategies, e.g., *stealing,* by Harris and Fraser [31] overcame the drawbacks of *helping* strategy. More on *stealing* would be discussed in chapter 4.2.

One of the limitations this system, includes the advance declaration of the memory locations that a transaction will acquire. This restricts the transaction from acquiring a memory location dynamically [46]. However, recent versions of the STM can acquire the memory locations dynamically e.g. Hash table based STM developed by Harris and Fraser [31].

Another drawback found in Shavit and Touitou's STM is maintaining separate memory locations for ownership records beside data values. This is not very efficient approach to keep ownership records. Consider the example, when a data object is extended over two or more words of memory with same redundant ownership records. This makes the memory requirement for a data object double and redundant as shown in the figure 4.1.

4.2 Word based Software Transactional Memory (WSTM)

Harris and Fraser [31] in 2003 introduced a first ever STM system that is an integrated part of an object oriented programming language, Java. WSTM was the first STM that detects the conflicts at the word-level hence given the name, Word-based Software Transactional Memory (WSTM). It has weak isolation with a deferred update mechanism. Moreover, it supports *flattened*, nested transactions. Other characteristics include optimistic concurrency control and synchronization achieved through non-blocking mechanism (obstruction freedom).

Table 4.2. The Basic Design features of WSTM.

WSTM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Word**
Isolation	**Weak**
Update Strategy	**Deferred Update**
Conflict Detection	**Late**
Conflict Management Strategy	**Helping**
Nested Transaction Type	**Flattened**

The basic design features of WSTM are shown in table 4.2.1. This system has late conflict detection mechanism. The conflict resolution is achieved through *stealing* [31]. The *stealing* strategy allows a transaction to take the ownership of the memory locations from a conflicting transaction by an atomic, compare-and-swap operation. However, this strategy ensures that the logical state of the memory locations must not change in *stealing* operation. Secondly, after the new transaction has taken the ownership of the memory locations, it is made sure that the new-owner transaction commit before the former owner-transaction.

WSTM does not require that memory locations are declared in advance for transactions like Shavit and Touitou's STM. WSTM is basically inspired by Hoare's conditional critical regions (CCRs) [39] and Lomet's [43] atomic blocks using CCRs. In CCR, a programmer can protect a particular code region under a boolean condition. Figure 4.2.1 shows an example of a CCR. In this figure atomic block is guarded by a condition that index is greater than or equal to zero. The control will not enter the critical condition region unless the index is not greater than or equal to zero.

```
public int get_Item(int index)
{
        atomic (index >= 0)
        {
                index --;
                return buffer[inde];
        }
}
```

Figure 4.2.1. The Conditional Critical Region [31].

Harris and Fraser, built the system by improving Lomet's work and presented a system that prevents deadlock with two phase locking, more-over it does not require defining the conditional variables for control synchronization in advance [31]. In contrast to Lomet, WSTM facilitated the CCRs to function on the basis of program state instead of specific variables. WSTM enhanced the

earlier version of CCRs. WSTM pauses the execution of a transaction until one of the condition variables is updated by another transaction [31]. WSTM can support procedural languages as well although it was implemented as part of an object oriented language, Java. WSTM introduced a novel statement in Java which allows declaring an atomic block of code with proper managed CCR. Fig. 4.2.2 shows the structure of a WSTM atomic block.

```
bool done = false;
while (!done)
{
        STMStart();
        try
        {
                if (<condition>)
                {
                        <statements>;
                        done = STMCommit();
                }
                else
                {
                        STMWait();
                }
        }
        catch (Exception t)
        {
                done = STMCommit();
                if (done)
                {
                        throw t;
                }
        }
}
```

Figure 4.2.2. The Structure of Atomic Block [31].

An important thing to note in the Atomic block shown in Figure 4.2.2, that it commits the transaction when the exception takes place. However, usually exceptions are only used for handling the errors and preventing the system to crash completely. The basic idea behind putting commit statement is to save the processing done, so far, in the transaction. Moreover, in nested transactions if the inner transaction fails, then, in order to save the outer transaction, committing inside the exception can be very useful [31].

Basic building blocks of WSTM library are as below [31];

1. void STMStart()
2. void STMAbort()
3. boolean STMCommit()
4. boolean STMValidate()
5. void STMWait()

WSTM maintains different helping data structures beside the main data structure i.e. the Heap. One of the data structures is Unique Transaction Descriptor, which uniquely identifies the transactions. Transaction status data structure maintains the status of a transaction. A transaction

has four states. It begins with Active state, and then changes its states to ASLEEP, ABORTED or COMMITTED. Another data structure is Transaction Entry which keeps the addresses of the different values like location of old value and version and its version, location of new value and its new version etc as shown in figure 4.2.3. Version numbers are very helpful in conflict management.

Another supporting data structure is the Transaction Descriptor which keeps the records like status of transaction, its nesting depth, and link to transaction entries as shown in figure 4.2.3. Basically Transaction Descriptor holds the transaction status data structure and transaction entry.

WSTM also maintains a data structure OREC (Ownership record), which keeps the records of ownership for every transaction [31]. Moreover it maintains the version number of memory address which has been updated or committed recently by any transaction. A typical OREC contains a pointer to the transaction which owns the memory location or a version number of it [31].

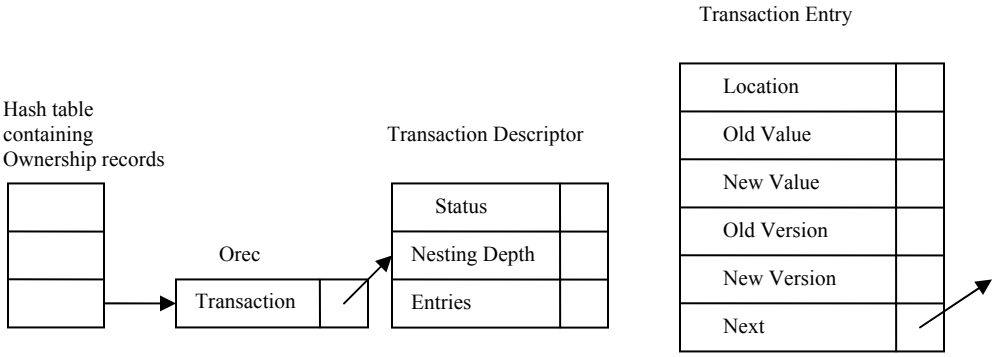

Figure 4.2.3. The WSTM Data Structure [31].

4.2.1 Design Details

The STMStart operation is used to allocate descriptor for a transaction and set its status to Active. STMAbort updates the transaction status to Aborted. In STMRead method, there are two cases, if the descriptor has no entry for a required memory location then it is a new entry otherwise system will find the current state of the location and start a new entry with new and old value while the version of the value would also be recorded as old-version and new-version [31].

MemRead function is used to get the memory location value and its version number. Information about value and version number is kept at different places based on the status of the transaction. If a location has not been accessed by any transaction then current value would reside in the memory location and ownership record would be holding the version number.

If a transaction accessed the memory location and committed its results then the latest value would be found in the new-value field and version in the new-version memory location of the

transaction. Similarly if transaction has not committed but it is in process then the value is kept in the old-value memory location of the transaction and version in the old-version memory location.

STMWrite is another method used by WSTM library. STMWrite, first make sure that entry exist for the memory location that is accessed. This is achieved through read operation. New value is written over old value and old version is incremented and set as new version and stored in a new location, separate from old version [31].

STMCommit operation takes the ownership records of all the memory locations which are accessed by the transaction. When transaction completes successfully it updates the transaction state to Committed and updates the memory locations with new values and releases the ownership records [31].

Transaction status updation, releasing the ownership records and releasing the values all step take place in an atomic and concurrent fashion. WSTM detects the conflicts while committing the transactions however it does not do validation continuously as in DSTM. So some time transaction may go into an inconsistent state and in this case transaction is aborted and executed again. WSTM However uses STMValidate to make sure that no loops inside a transaction run to infinity [31].

Nested transactions are bit more complicated when it comes to committing a transaction as outer transaction can expose the updated values. Moreover, inner transaction can cause the outer transaction to abort. Transaction descriptor tracks the number of nested transactions inside a transaction and keeps the commit operation pending until the outer-most transaction commits [31]. If inner transaction aborted then transaction can not commit at all.

STMValidate doesn't write or update anything; instead it is a read-only operation which verifies the ownership records for each memory location with its relevant version number [31]. WSTM library has STMWait that is used in CCRs for holding a transaction until another transaction modifies the memory location accessed by this transaction. In order to do so it first acquires the ownership of the transaction entry record and updates the status of the transaction to ASLEEP and stop the thread executing the transaction. Mean while the another transaction updates the memory locations, however it will conflict with the thread halted but conflict manager should permit the active thread to proceed ahead and then later on let the suspended transaction continue. As result, suspended thread will have to re-execute.

4.2.2 Design Limitations

Haris and Fraser [31] have identified several optimization shortcomings in their WSTM design;

1. One ownership record can be accessed by only transaction.
2. Read and Write operation requires searching the transaction descriptor for transaction entries in a specific orec.
3. Processing a read only access involves updating the orec twice, which is an overhead.
4. Retry operation prevents the system from being non-blocking.

In fact, WSTM is well suited for applications where concurrent operations are possibly conflict free [31]. Figure 4.2.3 shows the performance of WSTM in comparison with single lock and fine grained locking in hash table operations.

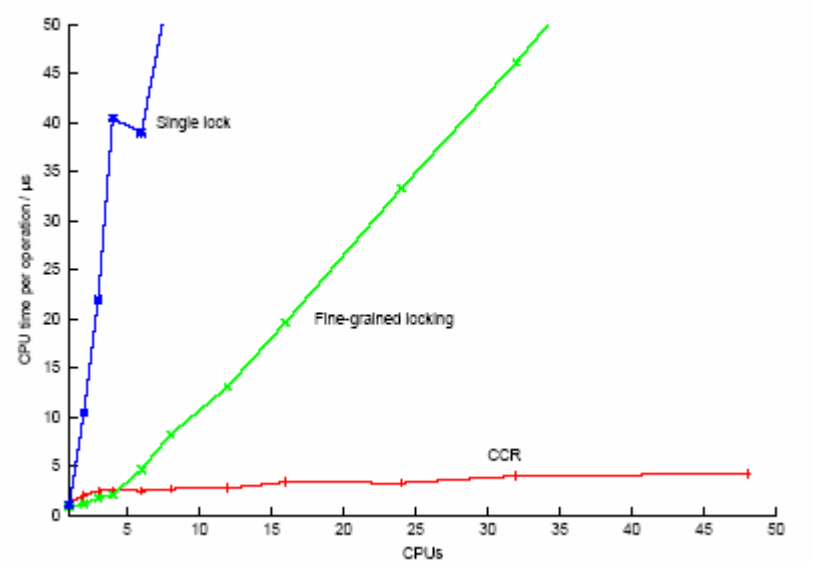

Figure 4.2.4. The Hash table update operations performance [31].

4.3 Dynamic Software Transactional Memory (DSTM)

The concept of DSTM was presented by Herlihy, Luchangco, Moir, and Scherer [36]. DSTM basically overcame the deficiency of the previous STM's where the transaction size and memory requirement was statically defined in advance. DSTM is designed using dynamic sized data structures like Lists and Trees, hence given the name, Dynamic STM.

Table 4.3. The Basic Design features of DSTM.

DSTM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object**
Isolation	**Weak**
Update Strategy	**Deferred Update**
Conflict Detection	**Early**
Conflict Management Strategy	**Contention Manager**
Nested Transaction Type	**Flattened**

DSTM is basically an Application Programming Interface (API) for programming dynamic data structures for synchronized applications without using locks. Prototype version of API is available in C++ and Java Language. DSTM uses obstruction-freedom property in order to achieve non-blocking synchronization. However obstruction freedom is a weaker property than lock-freedom that makes it simple to implement [36]. Obstruction-freedom assumes that a waiting thread does not hinder other thread from making progress.

In contrast to lock freedom, obstruction freedom does not completely prevent the occurrence of live locks. In other words, multiple running threads might interfere and hinder each other to progress ahead [37]. The obstruction-freedom property provides simple techniques for prioritizing the transactions as any transaction can abort another transaction at a particular point in time. However, a high priority transaction often aborts a lower priority transaction.

In contrast to the obstruction-freedom, in the locking mechanism, when a lower priority transaction acquires a lock on a memory location then higher priority transaction will have to wait until lower priority transaction completes its execution. This leads to priority inversion. Similarly in lock-freedom, a higher priority transaction may have to wait for a lower priority transaction in order to help it to complete its processing, under the assumption that some transaction may complete its processing [37].

DSTM offers obstruction freedom, a simple way of guaranteeing progress and prioritizing transactions. In DSTM, one transaction can foresee that it is going to abort another transaction and in this situation it corresponds with contention manager to find out whether it should abort the transaction or wait and let it complete its processing. DSTM has modular implementation of contention manger which implies that new contention management policies can be plugged-in without any possibility of disturbing the correctness of transaction code [37].

4.3.1 Basic Design Features

Following is a summary of basic design artifacts of DSTM, also shown in table 4.3.1 [36]

- DSTM uses Obstruction freedom as synchronization and non-blocking progress.
- DSTM uses an explicit contention manager to resolve the conflicts among transactions and decide whether to abort a transaction or let it proceed further.
- Another novel property which DSTM introduced is its ability to release an object before committing the transaction. This property, in the best-case scenario, can be very efficient however there is possibility of errors which can lead to an inconsistent state. This characteristic puts significant responsibility on the programmer to use it carefully and efficiently.
- DSTM is has weak Isolation property.
- Transaction granularity is at object level.
- DSTM is a Deferred Update system i.e. System keeps a separate copy of the data and after processing the data updates the original data.
- DSTM maintains optimistic concurrency control.
- DSTM has early conflict detection policy.
- Nested Transactions are flattened.

4.3.2 Overview with Example

In this section we will discuss how to use the DSTM in programming and what methods and procedures it has for different purposes.

DSTM maintains a collection of data objects that are concurrently accessed by different transactions. Actually Transaction object encapsulates the simple java objects and whenever there is a need to access it by any transaction, it is opened, read and modified as required. The updates and changes made by a particular transaction are not visible outside the transaction, until it commits. However, if a transaction aborts then its changes are discarded [36].

Transactions can be created dynamically, at any point in time. However, creating a new transaction and its initialization can not be done as a part of another transaction [36]. *TMThead* Class is responsible for the handling the activity of the threads. *TMThread* Class is inherited from regular java class, Thread. *TMThread* class provides methods for starting, terminating, running, committing and getting the status of the thread. Transactional Objects can be implemented by

TMObject class. In the example shown in figure 4.3.1, we encapsulate a simple counter object, inside a *TMObject* class object.

```
Counter aCounter = new Counter (100);
TMObject aTMObj = new TMObject (aCounter);
```

Figure 4.3.1 Encapsulating the Object inside TMObject [36]

Here is another, important point to note, that when a class inherits the *TMObject* class for encapsulating its objects then it must also implement the Interface, *TMCloneaable*. This interface, in fact, make it compulsory for the class that is implementing it to declare and implement a method that returns a clone of the object of the same class i.e. a separate isolated copy. This method is used for opening transactional objects as shown in figure 4.3.2. When a clone is created, it is made sure that object may not change, while it is cloned.

```
Counter aCounter = (Counter) tmObject.open(Write);
aCounter.increment(); // incrementing the counter
```

Figure 4.3.2 Opening a transaction in Write mode [36]

A transaction is started with *beginTransaction*() method, and it remains active until it aborts or commits. While a transaction is active, it can access the Transaction object by opening it. The method *Open()* creates a clone of original version and then transaction can perform its processing on it and commit its changes to the original version. Synchronization is only required when a clone is created that no other transaction modify it at that time later on synchronization is not required [36].

A transaction, commits its changes by triggering the method *commitTransaction()*, and it returns *true* if commit is successful otherwise *false*. Similarly a transaction can be aborted using the method *abortTransaction()*. DSTM ensures [36] that committed transactions linearizable. In other words, transactions are executed as if they are run one by one. However, inconsistencies in reading *TMObject* can lead to synchronization problems. For example a transaction may notice that the *TMObjects* it opened or will access next has already been modified. In order to deal with such situations DSTM uses *validation.*

The validation checks are triggered whenever a transaction object is opened by a transaction, and it finds out whether the same transaction object is not opened by another transaction, at the same time [36]. When it is opened then *Open()* method throws an exception instead of creating a clone of that object. Throwing an exception indicates that transaction object is opened at the moment by another transaction, and then aborts itself. Hence *validation* process avoids conflicting access of Transaction objects.

4.3.3 Design Details

Transaction object has an attribute, named, status that can have values like *active*, *aborted* and *committed*. As we know that Transaction class is container for other user class objects and a Transaction object has three attributes as below;

i) Transaction (refers to the transaction which is active at the moment)
ii) Old Object (refers to the old version of the transaction)
iii) New Object (refers to the new version of the transaction)

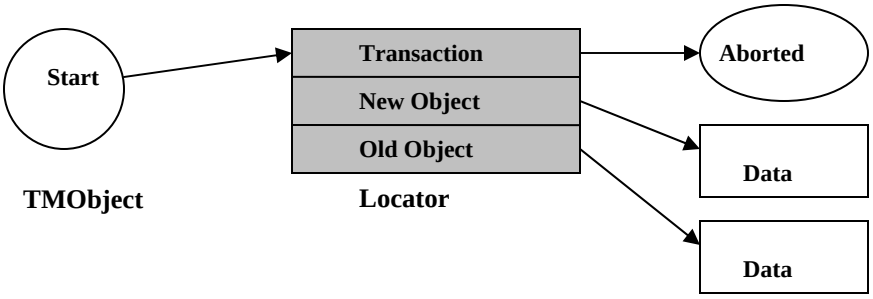

Figure 4.3.3. The Transactional Object Structure.

The transaction version can be find-out by the attribute, status of a transaction object. If the transaction status is committed then this is the new version and old version is discarded. In case, if transaction aborts then the old version becomes current version and new version become useless. Third case can be that a transaction is active, in this situation old version is current version and new transaction object is the temporary version of active transaction.

Consider the example where multiple transaction Objects have been opened by a single transaction and when this transaction commits, all the transaction objects replace their current version of transaction object with new transaction object. These versions were basically the clones which transaction acquired using *Open()* method, figure 4.3(a) and 4.3(b) shows the indirection from old version to new version.

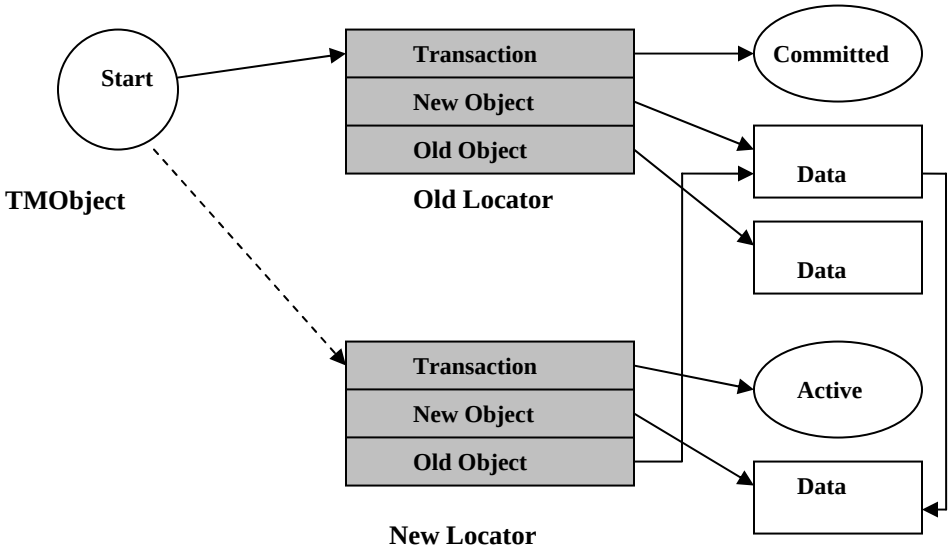

Figure 4.3 (a). The Opening Transactional Object after Commit [36].

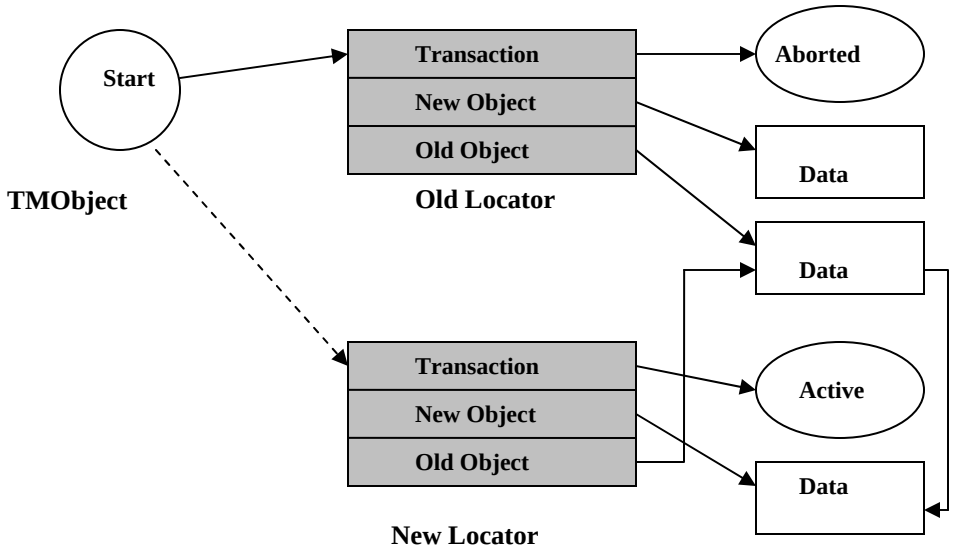

Figure 4.3 (b). The Opening Transactional Object after abort [36].

If we look at the figures 4.3(a) and 4.3(b), we can see that TMObject has single pointer which is referencing to a *Locator*. This locator further has three fields as we discussed earlier i.e. transaction, new object and old object. Using the CAS operation (compare and swap), old values are replaced with new one. The locator is basically a data structure consisting of pointers referring to transaction objects. It points to a read only transaction object if it is opened by another transaction for modifying. If there is a conflicting access then STM system using its conflict resolution policy, decide which transaction to suspend or abort and which one to progress ahead [36]. However there is no hard and fast rule that a particular transaction will always be allowed to progress ahead.

DSTM implements an interface for contention manger which allows numerous conflict resolution policies to act at the same time. The basic aim of contention manager is to make sure that a particular transaction attempting to access a transaction object must be given a chance to progress. This implies that if this particular transaction is trying to abort another conflicting transaction then it must be granted the permission after several attempts. Each running transaction has its own contention manager instance which every transaction deals with and decides whether to abort a transaction or wait and let it finish off first. Moreover contention managers of different transaction may communicate with each other to see the priorities and decide accordingly in different situations [36].

Contention manager implements two sorts of methods i.e. notification and feedback. Notification methods tells contention manager about different events, e.g., commitTransactionSucceeded(), an event which is triggered when some transaction completes it execution successfully. Moreover, other notification methods are *commitTransactionFailed*(), triggered when transaction fail to commit. The contention manager calls the feedback method to decide what to do in different conflicting situations. A good example of feedback method is *shouldAbort*(), which is called by contention manager when it finds out that transaction object is already opened by some other transaction. So it tells the transaction to abort.

4.3.4 Design Limitations

The contention management policies in DSTM are still an open area of research, where one can contribute and come-up with more efficient contention management policies. Another interesting design feature of DSTM is release operation which allows a transaction to release a transaction object before committing its results. Releasing the transaction object lowers the validation cost; however it may become possible source of conflict leading to abort.

4.4 Transactional Locking II (TL2)

Dice, Shavit and Shalev introduced Transaction locking II STM, in 2006 [16]. TL2 is an improvement of their work in *Transaction Locking* (TL) STM, in 2006 [17], hence, given the name, TL2. In this algorithm they introduced commit time locking with a global version clock, validation technique. Synchronization is achieved through commit time locking and global version clock validation. Whenever a transaction writes to the memory, global version clock is incremented and it is visible to other transactions for reading.

The idea of time stamping is not new instead it has been used in database transactions, long before [65]. However time-stamping used in TL2 is much faster and efficient than the databases as STM system is more robust and faster. Similar idea of time-stamping has been introduced by Reigel et al. [53] but TL2 is different than Reigel STM. TL2 is locked based and simple while Reigel's STM is non-blocking and complex in its design and more costly performance-wise [16].

Table 4.4. The Basic Design features of TL2.

TL2	
Synchronization	**Blocking (Lock-based)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object, word, or region**
Isolation	**Weak**
Update Strategy	**Deferred Update**
Conflict Detection	**Early or Late (Selectable)**
Conflict Management Strategy	**Aborting**
Nested Transaction Type	**N/A**

TL2 is a deferred update STM with blocking concurrency control. The basic design features of the TL2 are shown in table 4.3.1. Conflict detection is achieved through delaying other transactions or aborting them. Conflict detection can be early or late i.e. selectable. TL2 protect each memory location by locks and every lock has a version number. Transactions read the memory locations while validating with clock. When a transaction commits its updates the global version clock value and release the locks associated with that transaction [16].

Robert Ennals [21] has argued that deadlock prevention is the sole reason for making operating system non-blocking otherwise there is no need to provide such mechanism at user level. Dice and Shavit at el. also support this idea [16]. Locks eliminate the need for indirection in shared memory. However, locks still need a closed memory system. A closed memory system allows memory to be free up automatically, intermittently. TL2 manages the deadlocks and live locks by time-outs and by requesting other transactions to abort.

Studies [21] have shown that locked-based STM supersede non-blocking STM's due to its simple nature and implementation. But there are two limitations which still there to be addressed. The limitations are as below;

1. Closed Memory System

The closed memory system implies that a memory which is not in use must be free up automatically or in other words it has safe garbage collection mechanism. Some programming languages has this facility built in, while other have different commands to do it explicitly like malloc() and free() in C language. In java if memory is not referenced by any object then it is automatically released. Unfortunately, all the non-blocking STM systems require closed memory system, while lock-based STM systems also require closed memory and some time achieve it through malloc() and free() methods.

2. Specialized Managed Runtime Environment

STM systems require such environment which may over-comes the inconsistencies like infinite loops, illegal memory accesses and other run-time discrepancies [16]. Such kind of inconsistencies can be overcome by aborting and retrying a transaction. Moreover, infinite loops can be caught by validation checks. However validation on every read-set is a big overhead on performance of a STM system. This overhead can be reduced by intermittently checking for infinite loops instead of continuous checking [56].

In TL2, writing a transaction involves a collection of reads and writes sets and then acquiring the lock over the memory locations, for committing updates. A lock is also acquired over the global version clock for incrementing it. After a transaction commits, it update the memory with new value of global version clock and release the associated locks [16].

Dice and Shavit et al. believe that TL2 has overcome the majority of the safety and performance issues that were creating a bottle neck for lock-based STM implementations. TL2 has following design improvements compare to other STM systems [16].

1. In contrast to former lock-based STM implementations TL2 avoids the errors-proneness related to inconsistent memory states. Previously, either compiler assistance was required in this regard or explicit checks by a programmer.

2. Like other STM's, TL2 makes the memory to be automatically re-cycled for garbage collection and this is achieved with no significant complexity.

3. Concurrent Red-black trees are a data structure for STM that are infact derived from TL2. Generally TL2 is concurrent and faster than sequential single lock.

4.4.1 Design Details

TL2 is basically global version clock variant of Transaction Locking (TL), which was introduced by Dice and Shavit earlier [38]. TL2, uses two phase locking mechanism for commit-time locking, as TL used. Two phase locking implies that in the first phase transaction gradually acquires all the locks. This phase is called expending and once all locks are acquired then it commits. The second phase gradually releases, all the locks acquired. This phase is called shrinking.

Every transaction has an associated write lock with its corresponding memory location. A single bit is used to indicate that lock is acquired and it maintains a version number separately. This version number is incremented each time a transaction releases lock. The versioned write-locks provide a great deal of performance and correctness. In order to implement a data structure for write locks in case of shared memory location, there are several strategies like (i) Assigning locks per object (PO) (ii) Assigning locks per stripe (PS).

Writing Transaction in TL2

In this section we would discuss the steps involved in writing a transaction in TL2 [16].

1. Reading Global Version Clock. A transaction reads the current value of the global version clock and stores it in the local thread variable, called read-version number (rv). The rv, later on helps in detecting changes to the data by comparing rv with lock's version-number.

2. Speculative Execution. In TL2, a **t**ransaction execution is done speculatively. It is called speculative because while the transaction is executing, it does not have any effect on the shared memory. A transaction executes separately and later on commits the results. Moreover, while reading the data form the shared memory it checks the version of the data, so that it may not have changed. More-over it is checked that the attribute lock-version is less than the rv and lock is active.

3. Locking the Write-Set

In this step transaction acquires the lock for all the required memory locations, if it is successful then it proceeds to the next step otherwise, it aborts and try again.

4. Updating Global Version-Clock

If a transaction successfully acquires all the locks, then it increments the global version clock and records the value in write-version number (wv) attribute.

5. Validating the read-set

The read-set validation includes checking-out that every location in read-set has version number less than the version of write locks i.e. rv. Also it is verified that these memory locations were locked by other locks [16]. In case of validation failure, transaction is roll-backed. However, in case of abort step 3 and 4 is only repeated and it is made sure that meanwhile, no memory locations are updated. Moreover, if read version of lock, rv +1 is equal to write version of lock, (wv), then read-set is not required to be validated again, as it is made sure that meanwhile, no transaction will have modified it.

6. Committing and Shrinking Locking

The last step is to commit the results and update the write-set and release the locks and clear the lock bit. Similarly write-version, *wv*, is updated.

4.4.2 Design Limitations

The major difficulty with global version clock is that it introduces more contention and expensive cache coherent sharing [16]. However this can be reduced by making global version clock variable to include the properties like version number and thread ID. The transactions do not need to change the version number, if it is different than the last time it wrote. In such a situation it can store its version number in a separate memory location. This will lead to a reduced number of version clock increments.

Dice and Shavit et al. believes that TL2 is at least ten times faster and robust in performance than simple locking. Moreover TL2 can easily be used with hardware transactional mechanism [16]. The TL2 has given a new hope to use the locking in STM. More work is required in TL2, with regards to its integration with hardware approach and making the global version clock and read-sets more efficient [16].

4.5 Dynamic Software Transactional Memory II (DSTM2)

Herlihy et al. [38] introduced the Dynamic Software Transactional Memory II (DSTM2), based on their earlier work on the DSTM [36], discussed in chapter 4.3. DSTM2 is Java™ based software library. It provides a flexible framework for implementing software transactional memory. DSTM2 introduced a novel concept of *transactional factories* that are used to convert the un-synchronized sequential classes into synchronized one. More about *transactional factories* will be discussed, later in section 4.5.1.

Table 4.5. The Basic Design features of DSTM2.

DSTM2	
Synchronization	**Obstruction-freedom OR Locking**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Method**
Isolation	**Weak**
Update Strategy	**Deferred Update**
Conflict Detection	**Early**
Conflict Management Strategy	**Conflict Manager**
Nested Transaction Type	**Not Supported**

DSTM2 uses obstruction-freedom as well as a locked-based, synchronization approach [38]. DSTM2 does not support nested transactions. This feature may be added to it in the future. DSTM2 provides the transaction granularity at method level. DSTM2 provides a thread, called dstm2 that intercepts the method calls and checks their validity and decides whether a method can commit or not. Another significant aspect of the DSTM2 is that it facilitates the users in order to plug-in their own contention management, synchronization, and recovery strategies and techniques in the form of *transactional factories*, in the DSTM2 library [38]. DSTM, the earlier version, also supported the concept of plugging-in contention management techniques.

4.5.1 The DSTM2 Library Features

The DSTM2 library considers that there are several concurrent threads are sharing the data objects. The DSTM2 library introduces a novel way of declaring atomic classes. In order to declare an atomic class, a sequential interface is defined with a collection of methods for maintaining the consistency in the execution of transactions [38]. This interface is then passed to a constructor of a *transactional factory.* A transactional factory uses this interface and creates a synchronized version of anonymous class, implementing the interface and its methods. Later on, different objects of this anonymous class can be created. The Transactional factories, uses two different synchronization and contention management techniques i.e. obstruction-freedom and locks. The programmers are enabled to use their own contention management and synchronization techniques as well while plugging them into the DSTM2 library [38].

The design of the DSTM2 library is inspired by the lessons learned from the DSTM API. DSTM was a wrapper API, which works as a container for the transactional objects. DSTM was a good API for experimental purposes but it has several flaws in it [38]. Some of those flaws are as mentioned below;

1. When a memory location is opened for reading, then it must not be modified by another transaction, meanwhile.
2. When a memory location has been opened by a transaction for writing, its changes would be visible, afterwards, to other transactions that will open it for reading. But if we reverse the order of this scenario, then DSTM may not give correct results. In other words, if a

transaction has read values from a memory location and later on this memory location has been written by another transaction, then these changes may not be noticed by the first transaction.

3. The references, to the opened objects, for read or write operations, are valid until the lifetime of the transactions. Therefore, programmer must take especial care and must not use dangling pointers or references.

4.5.2 Transactional Factories

Transactional factories provide its own implementation for the methods declared in user defined atomic interfaces [38]. It is possible that one interface can be implemented by more than one class. Similarly, multiple factories can be merged with each other to construct a new factory, provided that the transactions created by these factories have a logical order and sequence [38]. DSTM2 provides a package, called, dstm2.Thread, that manages all the basic functions for the factories. Some of these functions are listed below;

1. Registering a method.
2. Aborting the method when method can not commit.
3. Identifying that a particular method can commit.
4. Helping the commit when a method can commit, otherwise cleaning up and aborting it, when it cannot commit.

1. Base Factory

In DSTM2, all the factories are inherited by the *BaseFactory* class. The class *BaseFactory* provides the basic functionality for all the other factories. When a DSTM2 defined factory tries to open an object, it checks that this object is not opened by another conflicting transaction. If it is already opened then the transactional factory consult the contention manager [38] to decide about whether to abort the conflicting transaction or wait for it to commit.

2. Obstruction-free Factory

The *obstruction-free* factory is based on the obstruction-freedom algorithm introduced in DSTM [36]. The structure of the objects created by the *obstruction-free* factory has three levels as shown in the figure 4.5.1. The first level is a cell, containing the reference to the locator. The locator further has three attributes. One of them is a reference to the new version of the object and the second is a reference to the old version of the object. Third attribute contains reference to the transaction descriptor. The *obstruction-free* factory is responsible for non-blocking execution of a transaction. The working mechanism of the *obstruction-free* factory is the same as discussed in section 4.2.3 for *obstruction-free* execution of the transactions for DSTM [38].

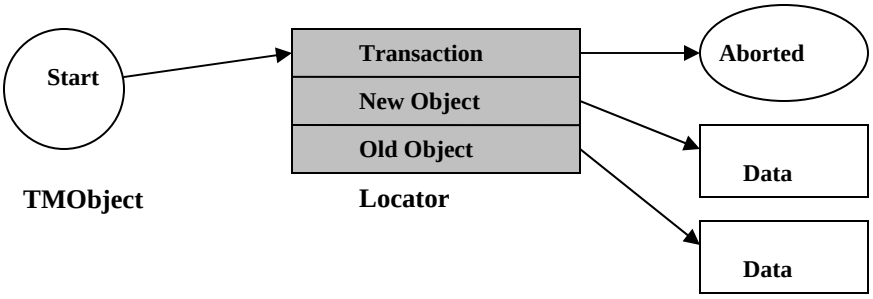

Figure 4.5.1 The Transactional Object Structure.

3. Shadow Factory

The shadow factory uses locks for the execution of the transactions. The shadow factory maintains shadow fields for each attribute defined in the interface beside the regular attributes [38]. The shadow factory avoids the indirection and memory allocation costs of the obstruction-free factory. However, the shadow factory is not well suited for multithreaded programming [38]. The shadow factory uses the method *backup()*, to copy each regular field defined in an interface to its shadow field as shown in the figure 4.5.2. Whenever a transaction opens an object, it finds out that the last operation performed by any transaction on this object was a commit or abort. If it is a commit then this implies that the current values in the regular fields are the latest. Therefore, the *backup()* method is called to copy the current values to shadow fields as shown in the figure 4.5.3. However, if the last transaction aborted while writing to this object, then it means that the shadow fields contain the latest values. Therefore, the *restore()* method is called to copy the original values from the shadow fields to the regular fields. In other words, shadow factory has the ability to restore and maintain the state of transaction in both cases, i.e. abort and commit.

DSTM2 have following novel features and advantages [38];

1. DSTM2 is the first flexible framework for programming STM applications.
2. DSTM2 does not depend upon the changes to the compiler or run-time system.
3. DSTM2 is simple, flexible, and portable

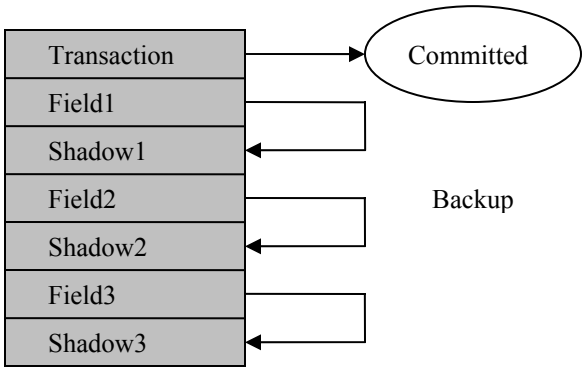

Figure 4.5.2 The Backup operation in shadowfactory.

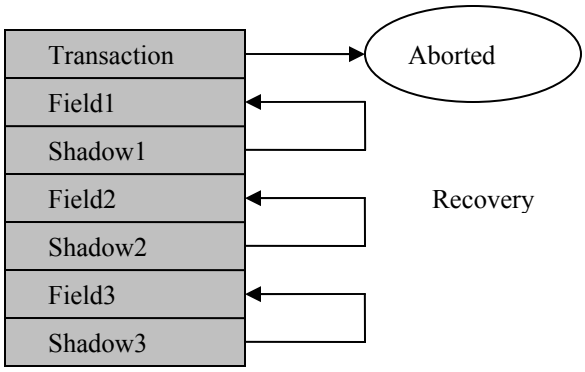

Figure 4.5.3 The Recovery operation in shadowfactory.

4.5.3 Design Limitations

DSTM2 is implemented in Java that has a built-in garbage collection mechanism. In other words DSTM2 is dependant on the runtime environment of the Java language regarding memory management. Moreover, DSTM2 creates a copy of every class and transforms it to a transactional factory which eventually exhausts the memory [18]. Although a single version of a class may be enough, instead of creating two versions. Fixing this problem needs considerable amount of re-work on DSTM2.

4.6 Object Based Software Transactional Memory (OSTM)

Keir Fraser, in 2004, presented the first lock-free, Object based Software Transactional Memory as his Ph.D. thesis [23]. It is also called Fraser's STM (FSTM). OSTM has transaction granularity at the level of an object; hence it was given the name OSTM. OSTM works on contiguous blocks of memory called objects. These objects are the basic unit of currency and update in OSTM [38]. The data structure in OSTM contains references to these objects. OSTM is quite similar to DSTM by Herlihy et al. [36]. The basic difference between two systems is this that DSTM uses obstruction-freedom for synchronization while OSTM is based on lock-freedom. Moreover, OSTM has weak isolation and late conflict detection approach. OSTM does not support nested transactions [38]. The basic design features of OSTM are shown in table 4.6.1.

Table 4.6. The Basic Design features of OSTM.

OSTM	
Synchronization	**Non-blocking (Lock-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object**
Isolation	**Weak**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Late**
Conflict Management	**Abort**
Nested Transaction Type	**Not Supported**

4.6.1 Design Details

The contents of the OSTM objects are stored in contiguous memory locations called *data-block*. In other words, nodes of linked-list are objects in OSTM. The contents of these objects are not accessible directly in an application, unless a pointer to an object-header for the current node is obtained. A reference to an OSTM object is in fact a pointer to a word-size object-header which is further used to track the current *data-block* as shown in figure 4.6.1. This pointer is updated to point a new data-block after each successful commit [23].

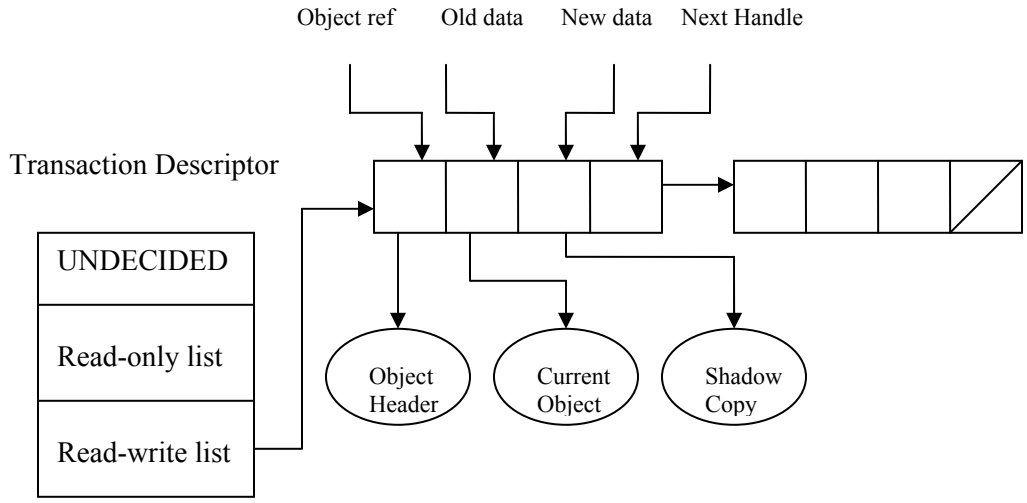

Figure 4.6.1. The OSTM data structure [23].

Multiple objects can be opened by a particular transaction for processing. Each transaction has a transaction descriptor. The transaction descriptor maintains a status field that can have values, e.g., Undecided, Read-checking, Committed, and Aborted. The transaction descriptor helps in tracking the objects opened by a particular transaction for read or write operations. The transaction descriptor maintains two linked list for this purpose, i.e., *read-only* list and *read-write* list. The *read-write* list is used for the objects which are opened for writing purpose [23]. Each node in the *read-write* list contains fields like *Object-ref* that is an object reference to concurrent object header, *old-data* that is a pointer to concurrent object, *new-data* that is reference to the shadow copy of the concurrent object, and finally the pointer to the next object, i.e., *next handle.* The updates to the shadow copy are kept invisible and private until the transaction commits. The *read-only* list has similar structure but it does not have a shadow copy of concurrent object [23].

Whenever, a transaction opens a data object for ready only access it is added to *read-only* list. Similarly, whenever a transaction opens a data object to write, it is added to *read-write* list. The OSTM has two phase commit operation. In the first phase, a transaction acquires control of the data objects need to commit, while in the second phase acquired objects are released. In the *Acquire phase,* data objects are acquired in a global logical order by replacing the data-block pointer with a pointer to the transaction descriptor. The next stage is *decision phase.* In this phase, transaction status is changed according to the final outcome of the transaction indicating success or failure. A transaction commits successfully when it updates all the shared memory objects

successfully in the *read-write* list. The last phase is *release phase.* In this phase, a transaction releases all the objects which it acquired in the *acquire phase* [23].

4.6.2 Design Limitations

One of the major performance bottlenecks that OSTM suffers is its read-only mechanism. The read-only operations have to go through acquire and release phases. This may introduce unnecessary conflicts among the transactions. In other words, it is un-necessary to acquire the concurrent objects for read-only access. Like other search algorithms that have a single entry point, i.e., root, for every operation, will suffer from performance bottleneck [23]. Tree data structure is good example to elaborate this drawback. In a tree data structure, if we want to access the extreme bottom leaf/node, we have to go through the root and traverse until we reach that particular leaf/node.

In order to overcome this drawback, Fraser modified the algorithm to acquire only those objects that are in a read-write list of a transaction. The *acquire phase,* is followed by a *read phase.* The *read-phase* checks the current data object in the transaction's read-only list for consistency. It compares current data object version with the version that was identified when it was first time opened. If the entire references match then the transaction commits successfully, otherwise, the transaction fails. If the *read-phase* notices that an object header is at the moment opened by another transaction then it will not help the conflicting transaction [23].

Fraser has discussed this scenario by an example [23]. Consider the transactions T1 and T2, running at the same time. Let suppose T1 opens a data object X_i for reading and Y_i for writing and the T2 opens X_i for writing and Y_i for reading. According to OSTM implementation both, T1 and T2 will commit successfully. However, these transactions violate the consistency and correctness criteria of OSTM. Since the transactions are not seriablisable, therefore changes made by T2 are not visible to T1 or vice versa. Fraser solved this problem by introducing two more changes [23]:

1. Fraser introduced a new transaction status value with the name of *read-checking*. This status value indicates that a transaction is in a *read-phase*.
2. A transaction successfully commits or aborts when transaction status changes to *read-checking*.

When the transaction completes the acquire phase, it turns to the *read-checking* state. In the *read-phase,* a transaction traverses through its *read-only* list and checks the consistency of the objects in the list. If any of the data object has changed since the transaction read it last time, and the transaction status is in *Undecided* state, then the transaction aborts. If the transaction comes across the conflict in *read-checking* state, then depending upon transactions position in the global total order in the memory is determined. If the current transaction is preceding the conflicting transaction then the current transaction aborts the conflicting transaction. Otherwise, the current transaction helps the conflicting transaction. OSTM allows only single fixed size objects. The fixed size objects are not suitable for applications that support heterogeneous collection of data objects or objects with variable size, e.g., skip lists. Moreover the OSTM does not support the nested transactions. It is another limitation in its design.

4.7 Rochester Software Transactional Memory

The Rochester Software Transactional Memory (RSTM) is designed for non-garbage collection programming languages by Marathe et al. [47]. RSTM is developed as a C++ library but an equivalent library can be developed in the C language. RSTM was an effort to lower the possible overheads experienced by its predecessors STM systems, e.g., OSTM, ASTM and DSTM, while exploiting their good qualities. The basic design features of RSTM are shown in table 4.7.1.

Table 4.7. The Basic Design features of RSTM.

RSTM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Both Early and Late (Selectable)**
Conflict Management	**Conflict Manager**
Nested Transaction Type	**Flattened**

RSTM has transaction granularity at object level and it is a non-blocking obstruction-free deferred update STM system. It can switch between late and early conflict detection strategy. RSTM introduced following prominent features [47];

1. RSTM uses single level indirection to access data objects.
2. RSTM uses its own garbage collector and avoids dynamic memory allocation and collection. Due to its own garbage collector, RSTM can work with non-garbage collecting languages, e.g., C++.
3. RSTM supports several options for contention management and conflict detection.

4.7.1 Design Details

In RSTM, *ObjectHeader* is the basic entry point through which an object is accessed. The *ObjectHeader* directly points to the current version of the object as shown in the figure 4.7.1, above. Inside the *ObjectHeader,* there is a clean bit which indicates that this data object is the current version or not [47]. If the clear bit is set to zero, then it means that this is the latest version of the data object and currently not referenced by any other transaction. However, when the clear bit is set to one, it indicates that this data object is currently opened by another transaction. The *ObjectHeader* of the data object references to the *TransactionDescriptor* of the owner transaction. The *status* of the transaction determines that the transaction is *active*, *committed* or *aborted* [47]. If the owner transaction has committed, then the new data object is the latest version. Otherwise, if the *state* of the transaction is active or aborted then old object is the valid and recent object.

Another prominent feature that RSTM introduced is visible and invisible read lists [47]. The data object header has a read list, called visible read list, which contains entries for the transactions which are currently reading it. Whenever a transaction open the data object to read, it adds itself to the visible read list. If there is no space to add a transaction entry to the read list, then transaction add its entry to its own private read list, called invisible read list. The invisible read list help transaction in validating its version of data. Let suppose, if a data object has changed since a transaction read it last time, then a transaction's invisible read list will help the transaction to validate its read version and abort if required. Similarly, when a transaction commits its

changes to a data object, it aborts all the transactions that are in the visible read list of the data object [47]. Therefore, visible and invisible read lists play an important role in the correct execution of RSTM.

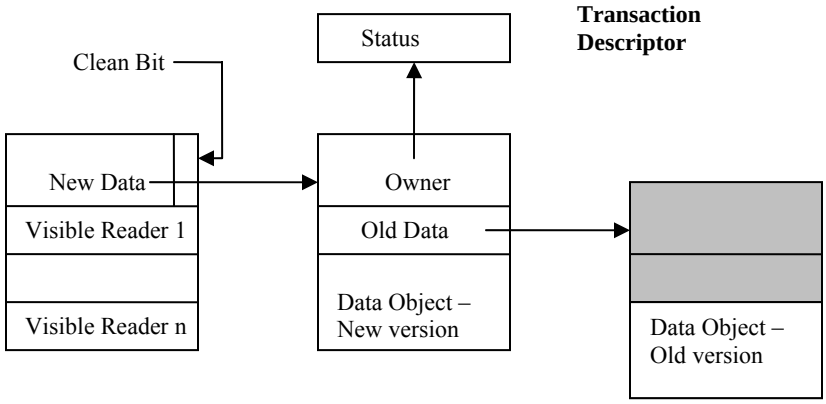

Figure 4.7. The RSTM data structure [47].

A transaction descriptor also maintains a write list [47]. Whenever a transaction writes data to a data object it adds that data object to its write list. When a transaction commits successfully, it uses the write list to clean the data objects it accessed. In other words, on successful commit, a transaction set the clean bit from one to zero in all the data objects it accessed for a particular commit operation. In order to commit changes to a data object, a transaction needs to own that data object. If a data object is already owned by another transaction then transaction will consult the contention manager. Let suppose, contention manager aborts other transaction in favor of this transaction. Now, in order to commit changes this transaction will perform following steps [47];

1. The transaction will create a *NewDataObject* and copy values from the old data object to the newly created data object and initialize its Old Data and Owner fields.
2. The transaction will add the data object to its private write list. The write list will later help in cleaning the header of the data object.
3. The transaction will abort all the transaction in the read list of the data object.
4. The transaction will commit its changes and set the clean bit to zero.

4.7.2 Design Limitations

The most prominent design feature offered by RSTM is that it can run without garbage collection and can reclaim its data structures after completion of a transaction [47]. However, the read lists introduced by RSTM generate more traffic and becomes costly. Moreover, late conflict detection with RSTM is more efficient than early conflict detection. As in late conflict detection more transactions has a chance to commit successfully.

4.8 Time Based Transactional Memory

Riegel et al. presented a new time-based software transactional memory in 2006 [54]. Riegel's STM uses the notion of time to maintain the consistency in data access and the order in which the transactions commit their results. Basic design features of Riegel's STM are shown in table 4.8.1. Former implementations of the time-based transactional memory systems used a single clock which is incremented whenever a transaction commits. However, in large time-based STM systems, contention on a single global counter is a major bottleneck [54].

Table 4.8. The Basic Design features of Time-based STM.

Time-based STM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object , Word (Any)**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Early**
Conflict Management	**Conflict Manager**
Nested Transaction Type	**NA**

4.8.1 Design Details

Riegel's STM achieves synchronization without validation cost [54]. The Transactional memory systems usually suffer from validation overhead. Whenever a data object is accessed, it is checked for validation. However, a time-based STM does not suffer from validation cost. Moreover, Riegel's STM does not require any specific underlying implementation. In other words, Riegel's STM can be implemented at word level, object level or even with hardware approach [54]. However, timestamp information is stored with every data object or word. Infact time-based STM uses time to impose synchronization. The time-based STM system can be implemented using simple counter shared by all the transactions. However Riegel's STM focuses on scalable time-bases that do not suffer from contention overhead. Moreover, clocks that work on time-bases can either tick whenever required by the STM e.g. on commit of every transaction or they can work independently like real time clocks [54].

Riegel et al. has presented a novel STM algorithm called Lazy Snapshot Algorithm (LSA) that uses real-time clocks to optimistically synchronize the concurrent transactions [54]. The LSA algorithm is later on modified to use externally synchronized clocks. Empirical study has shown that real time clocks are more scalable in large systems [54]. However, in both cases i.e. real-time and externally synchronized clocks; both do not actually need to be real-time clocks. In other words, neither the speed nor the values for these clocks need to be synchronized with real-time. However, the global real-time clocks help in implementation of externally synchronized clocks [54]. The local clocks approximate their time with real time clocks, which help in getting the unique time value. In the LSA algorithm, the clock ticks on every commit of transaction. Riegel's STM read the current time at the beginning of the every transaction in order to make the transactions *linearizable* [54].

Moreover every data object has a version that is valid for a certain period of time, called *validity range*. The *validity range* of transaction is a time range within which all the data objects accessed by a transaction are valid. The LSA algorithm uses two methods for getting the time stamps, i.e., GETTIME and GETNEWTS. The GETTIME method returns the current time while GETNEWTS returns a timestamp which is greater than anytime other time used by a transaction. However, timestamps are not necessarily be unique. Other transactions may use the same timestamp. In

some cases, it may be difficult to find-out which time stamp was read later and which one earlier. In order to resolve this uncertainty in LSA algorithm, we consider an example. Let suppose a time t_1 was read no later than the time t_2. In this case, sometime, we can say that t_1 and t_2 are equal. However, it is not possible that t_1 could be greater than t_2. The basic theme of the LSA is to generate snapshots of the data version and lazily extend *validity range* on demand. By doing so, two objectives are achieved [54]. First, consistent snapshots of the data lead to consistent reads by transactions. Second, identifying, that there is a conflict between the data *validity ranges.* However on commit linearizability is maintained. A transaction can only commit when it can extend its *validity range* up to a time that includes commit time. In other words, a transaction enters the commit state when it has determined that it can commit otherwise it aborts. In this way multiple transactions commits as long as their *validity range* is not in conflict [54]. The conflicting transaction is aborted by conflict manager.

4.8.2 Design Limitations

Time-based STM uses notion of time to synchronize the concurrent transactions. The real-time clocks have significant benefit over the simple global counters. The real time clocks avoid the contention suffered by simple global counters, which is a bottle-neck for short transactions or when the system is very large, running numerous transactions [54]. The LSA algorithm uses different time-bases. However, there would be a trade-off in using it for different systems. A simple shared commit time counter may be sufficient for the small systems. While in large systems, hardware-based external clocks may be more efficient.

4.9 Hybrid Transactional Memory

Kumar et al. proposed object-based Hybrid Transactional memory in 2006 [40]. Kumar's hybrid TM initially uses hardware TM and when the transactions exceed the limits of hardware resources these are gracefully shifted to STM. The system utilizes the performance benefits of hardware approach with the flexibility of software approach. Similar approaches have been proposed by Moir et al. HyTM [19], discussed in section 4.10 and NZTM, by Faud et al. discussed in chapter 4.11 [63]. The basic design details of Kumar's Hybrid TM are shown in table 4.9.1.

Table 4.9. The Basic Design features of Hybrid TM.

Hybrid TM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic and Pessimistic**
Conflict Detection level (Granularity)	**Object and Cache Line**
Update Strategy	**Deferred and Direct Update**
Conflict Detection Strategy	**Late and Early**
Conflict Management	**Conflict Manager**
Nested Transaction	**Supports**

4.9.1 Design Details

The execution mode can be chosen independently by a transaction. However, the software TM approach is totally different than the hardware TM approach. The basic difference between the two approaches is that the STM approach detects the data conflicts at an object level while the hardware transactional memory (HTM) approach has granularity at the cache level [40]. Kumar's hardware TM is based on the Herlihy original proposal [34] but the chip design is slightly more complex and large in area. The Instruction Set Architecture (*ISA)* has been used for more flexibility. The *ISA* is the part of a processor that is visible to the programmer or compiler and serves as boundary between the software and the hardware. The cache coherence protocol is used

to detect the conflicts between software and hardware transactions [40]. Kumar's Software Approach in Hybrid TM is also based on Herlihy's DSTM model, discussed in detail in the chapter 4.3. However, Kumar et al. modified Herlihy's model and made changes to the DSTM *Locator* data structure. The major change Kumar et al. introduced is that the *Locator* tracks reader and writers instead of just maintaining the references to old data and new data object and its state as shown in the figure 4.9.1. The change helps in detecting the read and write conflicts at an early stage which may increase the transaction commit rate. In Kumar et al. Hybrid TM, a transaction that is going to write, directly aborts all the readers and does not look for verification of the version of the data [40]. In contrast, DSTM requires verification of the data objects that were open for reading. In order to support this change Hybrid TM *Locator* has new field v*alid* that indicates that whether this data object is currently open for read or write. In contrast to DSTM one variable for state, Hybrid TM has separate fields for States i.e., read and write. In Hybrid TM, there can be multiple readers at the same time with maximum one writer [40]. The v*alid* field indicates which of the two *State* fields is valid at the moment.

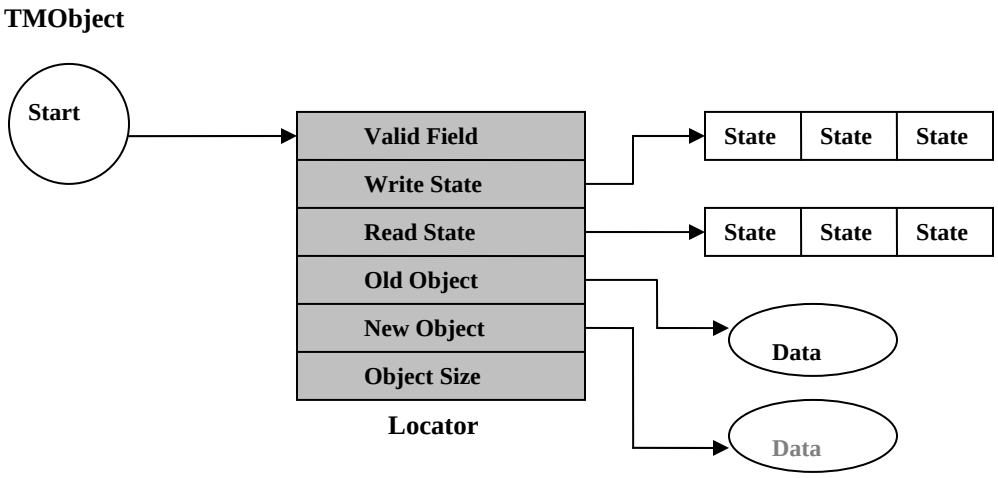

Figure 4.9.1. The Hybrid TMObject [60].

In the beginning of each transaction, the hybrid scheme chooses between the software or hardware mode of the execution. As the data buffer is within the hardware, so it is a more optimistic approach to choose hardware TM. HTM performs faster than STM as it has less overhead. The Hybrid TM currently follows a simple policy in this case, that is, a transaction first attempts in hardware mode three times and if it does not succeed in three attempts then it switches to software mode and retries until it succeed [40]. The Hybrid TM uses the *Polite* manager for contention management in software mode. However in hardware mode a transaction will automatically abort the conflicting transaction. The hardware mode uses aggressive contention management policy in contrast to the *Polite* manager in software mode.

4.9.2 Design Limitations

A number of improvements can be suggested in Hybrid TM of Kumar et al. One improvement can be a classification among the failing transactions. Some transactions abort due to conflicts with other transactions while other transactions abort due to resource limitation of hardware TM. The former transactions may be retried in hardware mode for efficient execution. However, this functionality requires implementation of exception handler in hardware mode which indicates

that what kind of transaction has failed [40]. Moreover, extremely long transactions create a problem for Kumar et al.'s Hybrid TM. A transaction that has longer time slice will never be able to complete successfully in the current system. In order to accommodate such transactions a third mode is required where current hardware may not be used at all. Moreover, all the current threads may be aborted and let this transaction complete execution. Such transactions rarely come across. However, this approach will ensure completeness without disturbing the overall performance [60].

4.10 Hybrid Transactional Memory (HyTM)

Moir et al. presented Hybrid Transactional Memory (HyTM) in 2006 [19]. Software transactional memory uses the best-effort hardware to enhance performance in HyTM but does not depend upon the Hardware Transactional Memory, HTM. The HyTM prototype is based on a compiler and a library. HyTM is word based and uses an explicit contention manager for resolving the conflicts [19]. The basic design details of the HyTM are shown in table 4.10.1.

Table 4.10. The Basic Design features of HyTM.

HyTM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Word**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Late**
Conflict Management	**Contention Manager**
Nested Transaction	**Supports**

4.10.1 Design Details

The synchronization between STM and HTM based transactions, is achieved by imposing the condition that a HTM based transaction does not commit when it detects a conflict with an STM based transaction. However, when an STM based transaction have a conflict with an HTM based transaction, then the HTM transaction is aborted and retries either in STM or HTM [19]. In HyTM, it is presumed that most of the transactions are executed in HTM as HTM is faster than STM. When a HTM transaction fails then a call is made to the HyTM library. The HyTM library decides whether the transaction may be retried in HTM or in STM. The HyTM library also employs different contention management policies, such as back-off and spinning before retrying a transaction. Some time a transaction is retried in HTM again, after a short delay in less contention. However, when a HTM transaction fails repeatedly, then it is moved on to the STM where hardware limitations are ruled out and transaction can be executed in a more flexible contention management environment [19]. HyTM contention manager uses the Polka contention manager which is a combination of Polite and Karma contention managers, explained in Chapter 3.4.7. The Polka manager uses the Polite manager's spinning and back strategy with the Karma manager's priority scheme. The priority scheme is based on the amount of work done by a particular transaction. The number of data-objects opened by a transaction is used as unit for amount of the data processed. The Polka manager gives priority to those transactions which processed more data and let them complete their processing [19].

HyTM uses two main data structures, *transaction descriptor* and *OwnerShipRecord.* The *transaction descriptor* further contains transaction header, read-set and write-set. The read-set and the write-set further have references to the corresponding orec instances for each transaction. The *OwnerShipRecord,* abbreviated as orec, is used for keeping the ownership records of the transactions which are reading or writing the data in a memory location. Before write or read

operation, a transaction acquires the ownership of the memory location. After every read operation, a transaction validates its read-set. The validation includes checking all the values which a transaction read and finding out that these values did not change since the transaction read it last time. After performing the validation a transaction commits and changes it status from *Active* to *Committed*. HyTM supports *nested* transactions and an outer transaction encompasses inner transaction. In other words, an inner transaction commits only when outer most transaction successfully commits [19].

4.10.2 Design Limitations

The basic theme of the HyTM was to increase the scalability of STM with HTM support. However, ideally HyTM should perform well in low contention and without HTM support. The trade-offs for HyTM includes validation cost. Each transaction requires checking the orec in order to find out possible conflicts. However, if software transactions are frequent or in other words, there is no HTM support. Then such reads to orec will slow down the scalability [19]. A different approach for HTM transaction may solve this problem. One way may be that a HTM transaction uses invisible reads or in other words, keep a snapshot of the orec for later validation. However, keeping separate orec will cause memory overhead but the solution lies in making the right choice.

4.11 Non-blocking Zero-Indirection Transactional Memory (NZTM)

Faud et al. presented a Non-blocking Zero-Indirection transactional memory in 2007/2009 [63][64]. NZTM is an object based Hybrid software transactional memory. It uses an explicit contention manager for resolving the conflicts. Kumar et al. also presented a similar hybrid transactional memory. Moreover, NZTM has much similar to Herlihy et al. object based, Dynamic Software Transactional Memory (DSTM) [36]. Basic design details of NZTM are shown in table 4.11.1.

Table 4.11. The Basic Design features of NZTM.

NZTM	
Synchronization	**Non-blocking (Obstruction-freedom)**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Late**
Conflict Management	**Contention Manager**
Nested Transaction Type	**NA**

4.11.1 Design Details

In contrast to former non-blocking STM systems where a conflicting transaction was aborted, NZTM transaction aborts itself and takes a short time to complete roll-back. After that other transaction can access the data object without any uncertainty. This mechanism helps in avoiding the overhead introduced by different levels of the indirection except the case when a transaction is not responsive [63]. By achieving zero level indirection, NZTM, eliminate the performance gap introduced by previous blocking and non-blocking STM systems. The data structure and programming model used by NZTM is similar to DSTM [36].

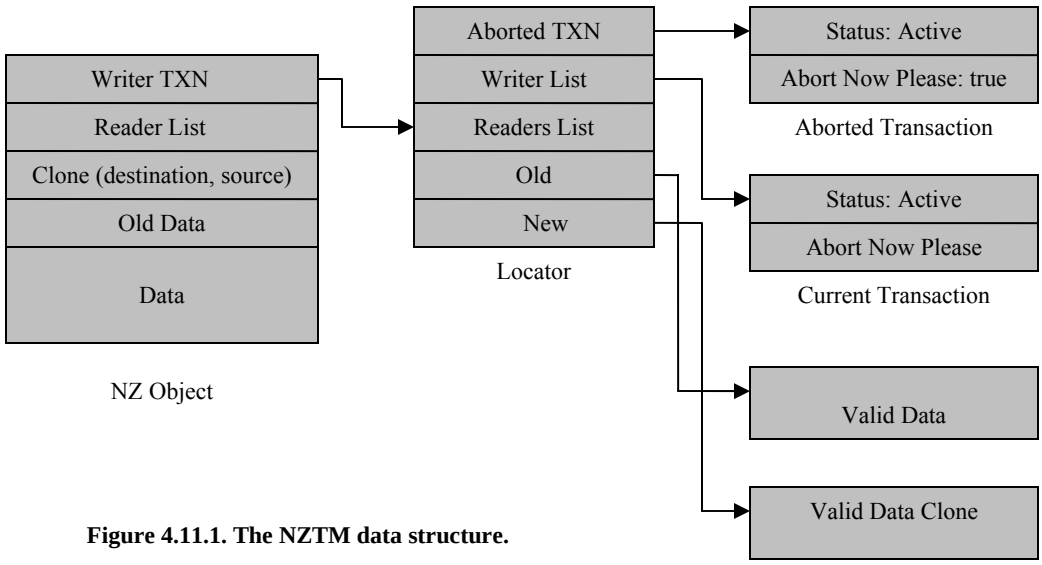

Figure 4.11.1. The NZTM data structure.

Moreover, transaction structure is also similar to DSTM except that NZTM transaction introduces a flag *AbortNowPlease*. This flag is used to request that transaction abort itself as shown in figure 4.11.1. The flag is stored with Status field and both are atomically accessed [63]. Similar to DSTM, a transaction starts with status set to *Active.* The transaction opens different data objects for reading and writing. When a transaction completes its execution, it sets its status from *Active* to *Committed*. When a transaction detects a conflict with another transaction, it waits. If it is not successful after few attempts then request to abort itself, after consultation with contention manager [63]. In contrast to DSTM and other STM systems, a NZTM transaction does not abort conflicting transaction but request to abort itself. This request is initiated by setting the *AbortNowPlease* flag to true. When a transaction notices that *AbortNowPlease* is set to true, it updates its status to *Aborted* [63]. As NZTM is Hybrid transactional memory, therefore system uses Non-blocking Zero Indirection Software Transactional Memory (NZSTM) for software transactions. The transactions attempts repeatedly using Hardware NZTM and if not successful then retries using NZSTM [63].

Faud et al. [64] performed further comparative performance testing of NZTM with several other Transactional Memory systems, e.g., LogTM-SE, BZTM, DSTM2-SF, SCSS. The performance testing revealed that LogTM-Signature Edition outperformed NZTM. NZTM has additional hardware mechanism to detect the conflicts with software transactions while pure hardware based transactional memory such as LogTM does not require such mechanism.

4.11.2 Design Limitations

NZSTM is non-blocking STM avoids the overheads of the indirection levels. NZSTM can be used with current systems without any additional hardware support. The theme of NZTM was inspired by the work of Marathe and Moir but their HTM is word based, in contrast, to object-based NZTM. However, NZTM use complex metadata to achieve obstruction-freedom [60].

4.12 Phased Transactional Memory (PhTM)

Phased Transactional Memory, PhTM was presented by Mark Moir et al. in 2007 [42]. The PhTM switches between different phases, each implemented by a different Transactional Memory. In other words, PhTM uses different transactional memory implementations, in different phases, which suits according to the circumstances prevailing. The PhTM is a Hybrid TM model. Therefore, it has two executions path, i.e., Hardware and the Software. Some time it shifts to software transactional memory and some time to hardware transactional memory. A Hardware Transactional Memory (HTM) significantly outperforms the Hybrid Transactional Memory, because HTM does not take care of conflicts with concurrent software transactions. This is the reason that Mark Moir Hybrid TM is outperformed by TL2 [42].

4.12.1 Design Details

The PhTM support different modes for various situations based on the workload and system support. The key challenges faced in developing PhTM include identifying the suitable and efficient mode for a particular situation. Moreover, it includes managing the transition from one mode to another mode without losing data or transactions and deciding when to switch to another mode [42]. The PhTM is integrated to C/C++ complier. Following are the different modes of execution for different situations [42];

1. HTM support is available and effective for the current workload. In this situation all the transactions are executed in HTM, as HTM is not suppose to take care of the conflicts with software transactions.
2. HTM support is not available or may not be effective for the current workload. In this case there are is no point in executing the transactions with HTM support. Therefore, software mode is suitable in this scenario. All the transactions are executed in software mode and there is no need to interoperate with hardware. The state of the art STM can be used in this case to execute the transactions.
3. HTM support is available but not effective all the time. In this scenario the combination of both software and hardware mode of execution, i.e., Hybrid TM would be suitable e.g. HyTM Moir et al. [19]. However, software and hardware transactions would have to check each other for conflicts.
4. There are few transactions or workload is single-threaded. This mode is called SEQUNETIAL and executes one transaction at a time. In this case there is no need of conflict detection at all as only one transaction executes at a time. Hence a significant overhead is eliminated.
5. Workload is single-thread or few transactions are executed at a time. Some transactions are not supposed to be aborted explicitly. In this case SEQENTIAL-NOABORT mode is used. In this mode transactions can execute without any conflicts detection. As transaction will not be aborted so it eliminates the need for logging. This mode is similar to SEQUENTIAL mode. However, SEQUENTIAL-NONABORT like other modes supports functionality for executing I/O operations or system calls which other modes also supports.

Many practical scenarios fall in to the category 1 and 2. Moreover, supporting multiple dynamic modes for different scenarios helps in load balancing that result in efficient execution of transactions. One of the main challenged faced in implementing the PhTM is transition from one mode to another when required. One solution is to abort all the transactions executing in the current mode are aborted or completed before shifting to the new mode. This solution can be implemented by using the read-only variable *modeIndicator*. By checking this variable each transaction can decide that it should abort or continue. The transactions will execute if mode has

not changed. And when the modes changes the transactions will abort. The transactions will read *modeIndicator* variable once and keep it with themselves for later validation. In other words, one copy of modeIndicator would be a part of a transaction and one would be global. And comparing both each other will indicate whether mode has changed or not. Another approach can be this that all the current transactions should complete their execution while the new transaction may execute in the next mode. Another major challenge in PhTM is to decide when to switch to another mode [42]. One simple solution in this case can be that we run the transactions in phases in each execution mode and monitor the performance. When we notice that the transactions are executing efficiently in a particular mode of execution then PhTM may allot more time to that mode to run the transactions. Moreover contention manager can be used to monitor the performance of particular execution mode and help in deciding shifting to another execution mode.

4.12.3 Design Limitations

The PhTM supports to employ the benefits of several Transactional Memory systems. Moreover, several modes of execution help in load balancing. However shifting from one mode to another causes delay which can slow down the over all performance of the system. Moreover, employing different STM and HTM system at the same time would require more memory.

4.13 Signature Accelerated Transactional Memory (SigTM)

Cao Minh et al. [48] presented Signature Accelerated Transactional Memory, SigTM in 2007. The SigTM is a Hybrid Transactional Memory system. The SigTM uses hardware signatures to keep track of read-set and write-set and help in conflict detection. The signatures are basically data structures that can store the data access information of the transactions. A SigTM signature is shown in table 4.13.2. However, data versioning information is stored in software part of the system. The signature data-structure in SigTM does not require any modification to the hardware caches which reduce the hardware cost. The SigTM supports nested transactions and strong isolation [48]. It utilizes the strong isolation and performance characteristics of HTM with low cost and flexibility of the STM. The basic design details of SigTM are shown in table 4.13.1.

Table 4.13.1. The Basic Design features of SigTM.

SigTM	
Synchronization	**Non-blocking**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Cache line/ Word**
Update Strategy	**Deferred Update**
Conflict Detection Strategy	**Late**
Conflict Management	**Aborting**
Nested Transaction Type	**Supported**
Isolation	**Strong**

4.13.1 Design Details

The SigTM uses hardware signatures for conflict detection and strong isolation by looking up coherence requests. While other functionality, i.e., transactional versioning, commit and rollback are dealt in software part. In contrast to the SigTM, other Hybrid TM systems, e.g., Hybrid TM of Kumar et al [40] discussed in section 4.9, HyTM of Moir et al. [19] discussed in section 4.10, make changes to the hardware cache. The SigTM uses TL2 as STM part of Hybrid TM system, discussed in section 4.4 [16]. The TL2 is locked based STM using optimistic concurrency control with granularity at word level and works fine for range of contention scenarios. The TL2 uses

global version clock to generate time stamps for the data versioning. The STM transactions are slower than the HTM transactions due to the versioning and conflict detection overhead. Moreover, the maintenance and validation of read-set is another bottleneck. Every word read, must be re-validated for its time stamp while committing a transaction. Secondary, overhead includes searching the write-set for the latest values. The STM overheads can be eliminated through compiler support and other manual optimization techniques [48].

However, the SigTM eliminates the global version clock and locking mechanism in the base STM. Moreover, it eliminates the need for software read-set. However software write-set is still required to store the transactional updates, until the transaction commits [48]. In contrast to other Hybrid TM systems [40, 41, 43, 44], the SigTM does not require to switch between STM and HTM. The Hardware Transactional Memory system implements the conflict detection and version-management by modifying the cache and cache coherence protocol. The HTM system uses cache to buffer the write-set until a transaction commits. Moreover, it has cache line granularity. The HTM, have one cache line for read bit (R) and one write bit (W), which indicates the ownership of a transaction in the read or write set.

Table 4.13.2. The Signature in SigTM.

Instruction	Instruction Description
rsSigReset wsSigReset	Reset all bits in read-set or write-set signature
rsSigInsert r1 wsSigInsert r1	Insert the address in register r1 in the read-set or write-set signature
rsSigMember r1,r2 wsSigMember r1,r2	Set register r2 to 1 if the address in register r1 hits in the read-set or write-set signature
rsSigSave r1 wsSigSave r1	Save a portion of the read-set or write-set signature into register r1
rsSigRestore r1 wsSigRestore r1	Restore a portion of the read-set or write-set signature from register r1
fetchEx r1	Pre-fetch address in register r1 in exclusive state; if address in cache, upgrade to exclusive state if needed

The contention management in SigTM is analogous to the base STM, i.e., TL2. A conflicting transaction is backed-off and retried after a delay. And when a transaction is repeatedly backed-off then it is eventually aborted. The SigTM implements lazy data versioning and transactional updates are buffered until the transaction commit. Moreover, the operating system may also suspend a SigTM transaction.

4.13.2 Design Limitations

The SigTM used hardware signatures to track the read-set and write-set while reducing the overhead of Software transactions. Moreover, signature data structure makes the implementation of nested transactions easy. One of the major performance challenges faced by SigTM is the in-exact nature of the signatures. Therefore, it is hard to find that what operations are taking place in read-sets and write-sets. This in-exact nature of signatures, lead to false conflict detection. Eventually false conflict detection deteriorates the over-all performance [48]. Abadi et al. [1] presented off-the-shelf memory protection hardware to detect conflicts in transactional and non-transactional data access. Using, such off-the-shelf hardware for data access tracking, in place of signatures can over-come the deficiencies introduced by signatures.

Another source of false conflict detection is the existence of two different granularity levels in the SigTM Hybrid Transactional Memory system. The SigTM uses conflict detection at cache line level while the STM uses word level granularity. The SigTM is unable to track word addresses in hardware signatures.

4.14 DracoSTM

The DracoSTM was presented by Justin Gottschlich and Daniel A. Corners in 2007 [29]. The DracoSTM is lock-based, C++ STM library. Recent research work has shown that lock-based STM out perform non-blocking STM [30, 47]. The DracoSTM is the first STM which supports direct and deferred updating and switch between two on run-time. The basic design details of the DracoSTM are shown in the table 4.14.1. DracoSTM design is an effort to prove that scaling concerns and other problems like deadlocks, priority inversion can be avoided with specific contention management policies.

Table 4.14. The Basic Design features of DracoSTM.

DracoSTM	
Synchronization	**Lock-based**
Concurrency Control	**Optimistic**
Conflict Detection level (Granularity)	**Object**
Update Strategy	**Deferred and Direct Update**
Conflict Detection Strategy	**Late/Early**
Conflict Management	**Aborting**
Nested Transaction Type	**Supported**

4.14.1 Design Details

The DracoSTM uses one lock per thread for transactional read and write operations. When a transaction is about to commit, all the transactions are temporarily blocked except the committing transaction. A global locking strategy is used to block all the executing transactions. This locking strategy provides the benefits of non-blocking system. The strategy implements the locking strategy in such a way that when transactions are not committing then they do not block each other and progress ahead. The DracoSTM supports switching between early and late conflict detection mechanism on run-time [29]. DracoSTM is the first STM which introduces commit time invalidation. The basic difference between validation and invalidation is that invalidation can detect priority inversion whereas validation can not. Moreover, invalidation can save many wasted operations by early notification about the transactions status. The DracoSTM implements closed nested transactions. In case of the closed nested transactions, child transactions are visible to parent and parent transaction is visible to child transactions but outside transactions can not see them in intermediate state. The DracoSTM can switch between direct and deferred update system. However, deferred update outperforms direct update while used with commit time invalidation. Though, in some systems [16] direct update is faster than deferred update, but those systems do not use commit-time invalidation [29]. The update strategy is closely tied with conflict detection strategy. The direct update strategy allows only one transaction to write to the memory location and hence early conflict detection is required in this case. But in case of deferred update, late conflict detection is performed. Implementing early conflict detection in case of deferred update can prevent many transactions from committing successfully. Therefore, late conflict detection is coupled with deferred update strategy [29].

4.14.2 Design Limitations

The DracoSTM supports a novel feature with regards to update strategy. However, it has a limitation that a transaction cannot switch to another update strategy on run-time. First, all the transactions need to be suspended or may be completed and then DracoSTM should initialize new update strategy. Moreover, switching to another update strategy would cause delay, which would eventually deteriorate the overall system response time. DracoSTM uses locking and proves that it outperforms a non-blocking STM, i.e., RSTM. However, it uses commit time invalidation to achieve the benefits of non-blocking synchronization i.e. preventing priority-inversion, deadlocks and live-locks, which consequently becomes an overhead on the system.

4.15 McRT-STM

Saha et al. [57] developed a locked based STM, named McRT, in Intel research labs in 2006. The McRT is a Just-in-Time (JIT) compiler developed for C/C++ and Java transactions. This complier is a complete STM system based upon compiler and runtime optimizations for transactional memory constructs. The McRT STM system supports word or object based granularity with direct update system. Moreover, it has optimistic read and pessimistic write conflict detection strategy. The McRT uses aborting as conflict resolution strategy and supports closed nested transactions. The basic design details are also shown in table 4.15.1

Table 4.15. The Basic Design features of McRT-STM.

McRT-STM	
Synchronization	**Lock-based**
Concurrency Control	**Optimistic read and pessimistic write**
Conflict Detection level (Granularity)	**Object or word**
Update Strategy	**Direct Update**
Conflict Detection Strategy	**Early write and late read conflict detection**
Conflict Management	**Aborting**
Nested Transaction Type	**Closed Nested Transaction**

4.15.1 Design Details

The McRT-STM directly updates the memory location instead of buffering the updates separately until the commit time. However, before updating a memory location a transaction records the original value from the memory location so that when a transaction aborts it could restore the original values [57]. The direct update strategy is faster than deferred update as it avoids the cost of buffering the original values separately and later on updating the original memory location with buffered results. However, the cost of aborting in case of direct update is higher than deferred update. In case of deferred update, a transaction acquires lock over memory location, hence preventing all other transactions to proceed further. Therefore, discarding the computation done a transaction is more costly than deferred update. Moreover, if transactions abort too much than it can significantly deteriorate the performance of direct update system, compare to a deferred update system. However, locked-based systems have less frequent aborts with direct update in contrast to deferred update system [57].

The McRT-STM uses two phase locking as synchronization strategy. In the first phase it acquires all the desired locks while in the second phase after committing the transaction it releases the locks acquired earlier. When a transaction commits changes to the data object, it does not abort the transactions which previously read the data object. However, the transactions which read the data-object previously, detect the conflict and abort when compare the data version-number with

read-set, as part of their commit process. In other words, McRT-STM uses, locking for write operation and data-versioning for read operation to avoid conflicts.

4.15.2 Design Limitations

In contrast to Transaction Locking, TL [17] and TL2 [16] STM's, McRT-STM locks an object on the first access until it commits and uses direct update mechanism. However, it is against the basic notion of Dice and Shavit basic model of Transactional Memory [59], where multiple transactions could access a shared memory at the same time. Therefore, if McRT-STM uses commit-time locking instead of the current strategy of locking the data-object from first access to commit-time, it may enhance its performance.

5. SUMMARY AND DISCUSSION

In this survey, different Transactional Memory systems are discussed on the basis of their design features as shown in table 5.1, beside their other particular strategies of implementation. Here, we present a summary of the design and implementation trade-offs. Moreover, at the end of this chapter we will discuss the contributions made by this survey.

5.1 Synchronization Approaches

There are basically two main approaches for synchronization, i.e., Blocking and Non-blocking. In this survey, it was observed that the most frequently used approach is Non-blocking. The non-blocking approach has further categories like Lock-freedom, Obstruction-freedom and Wait-freedom. However, wait-freedom and lock-freedom have not been paid much attention as both are practically not so promising. Wait-freedom allows each process to progress without taking contention into consideration. Moreover, the wait-freedom believes that there would not be any resource starvation, that is however practically not possible. Lock-freedom ensures deadlock prevention but suffers from resource starvation. The obstruction-freedom ensures that multiple threads run when there is no contention. However, it introduces the problem of live-lock and deadlock. In order to prevent live-lock and deadlock, roll back is used. Moreover, a contention manager is used to resolve conflicts among contending transactions. The obstruction-freedom is more promising than other non-blocking properties, due to its simplicity and performance.

The other major synchronization strategy is the Blocking approach or Lock-based approach. Though locking introduce problems like deadlocks, live-locks, convoying and priority inversion but some practitioners still believe in locking as a performance oriented synchronization strategy. The DracoSTM [29] discussed in section 4.14, used locking as synchronization strategy while they ensured prevention from priority-inversion through implementation of commit-time invalidation. Moreover, they have showed that deadlocks and live-locks can be prevented through specific contention management policies. However, research continues in both areas of synchronization approaches, i.e., blocking and non-blocking. More research work is required to develop smart implementation of a particular approach with minimum overhead.

5.2 Concurrency Control

When two or more transaction tries to access a shared data object, a conflict arises. In order to solve this conflict two events take place, i.e., *conflict-detection* and *conflict-resolution*. If all three events *conflict-occurrence*, *conflict-detection* and *conflict-resolution* take place instantaneously one after another then it is categorized as *pessimistic* approach. However, if *conflict-detection* and *conflict resolution* take place after some time of *conflict occurrence* then it is classified as an *optimistic* approach.

In this survey, it was noticed that most of the systems use the *optimistic* approach. The *pessimistic* approach seems to be more performance oriented as it resolves the conflict as soon it occurs and saves the unnecessary computation. However, the *optimistic* approach is more efficient than the *pessimistic* approach as there are situations when a conflicting transaction could have succeeded but does not succeed due to the early conflict-detection and resolution. Consider the situation where there two transactions T_1 and T_2 and both have a conflict with Transaction T_3 over two different objects. Meanwhile T_2 aborts as soon as it identify that it has a conflict with T_3 .Similarly T_3 aborts because it had a conflict with T_1. Now in this situation if conflicts would have been identified late then there was a chance that T_1 and T_2 could complete their processing successfully without aborting. Some other STM systems, like McRT-STM [57] discussed in section 4.15 have

introduced *Pessimistic-read* and *Optimistic-write* conflict resolution which is a bit more performance oriented than pure *pessimistic* approach.

5.3 Transaction Granularity

Transaction granularity implies to the storage space on which a TM system detects conflicts. The systems studied in this survey used various granularity levels, e.g., Object based, Word-based, Cache-line based and Method based. Other granularity levels include Block-level granularity, which may include a collection of memory words. However, mostly STM has a granularity at object-based or word-based. On the other hand, most Hardware-based TM systems have a granularity at cache-line level. An Ideal STM system maintains meta-data beside every data object so that it could track and control any possible conflicts. In case of object-based granularity level, meta-data can be kept easily with data object. Another approach is to maintain meta-data in separate data structure in case of word-based or block-level granularity. *Block level granularity* offers more precise sharing of resources than *object level granularity* but mapping meta-data from memory to a separate data-structure is another overhead besides keeping a separate data storage. However, object-based *granularity* is more understandable to the programmer than *block level granularity* as objects are more visible to programmer than memory blocks.

5.4 Update Strategy

When a transaction completes successfully it updates the original values with updated one. There are two approaches based on the update strategy, i.e., *Direct Update* and *Deferred Update*.

In case of the *direct update* a transaction modifies the original value directly. However the system maintains the original value so that when the transaction aborts, it is able to roll back the changes made by the transaction. In order to keep record of the original values, an STM system maintains a log of the activity. The cost of aborting is high in case of *direct update* as every transaction which read the updated value will have to roll back. However recent systems like McRT-STM [57] discussed in section 4.15, have showed that if the system does not abort too frequently then direct update strategy results in higher performance.

Another approach to update the values is *Deferred Update.* In this approach a running transaction maintains a separate copy of the updated values. When a transaction successfully completes its execution then it copies all the values from temporary copy to the original memory locations. However, in this approach, maintaining a separate copy of variables is another overhead besides copying from temporary locations to original locations. As the cost of a roll back in direct update is more than a successful commit, so deferred update seems more performance oriented than direct update.

In this survey, we notice that some TM systems e.g., DracoSTM [29] discussed in section 4.14 PhTM [42] discussed in section 4.12, use both deferred and direct update strategy. However, a TM system can not switch from one update strategy to another, unless it suspends all the current transactions and execute them in another mode. Moreover, another approach can be that the new transactions may not be initiated until system switch to next mode. Moreover, let the current transactions complete. However, switching from one mode to another mode causes time delay which effects the system response time and deteriorate the over all performance.

5.5 Contention Management Strategies

The TM systems covered in this survey used different contention management strategies. The early systems used the concept of *helping*. The *helping* strategy implies that if a transaction can not proceed further then it should abort and help other transaction in completing their execution. However, the *helping* mechanism is very complex and may also introduce the problem of recursive helping. Recursive helping implies that a transaction helping another transaction may already be helped by another transaction. Other approaches include aborting, in which a conflicting transaction aborts itself. Today, most of the TM approaches use contention managers for resolving the conflicts. An ideal contention manger ensures forward progress. A dynamic contention manager may employ several contention management policies depending upon the situation prevailing.

In this survey most of the TM system used contention managers for resolving conflicts. There are different contention managers available as discussed in section 3.4.7, e.g., *Polite manager, Karma manager, Kindergarten manager, Time-stamp manager*. Other contention managers include aggressive contention manager which aborts the victim transaction. The Time-stamp manager generates time stamps and aborts the transactions on the basis of time-stamp. A transaction with higher time-stamp has priority over a transaction with lower-stamp. In short no contention manager is perfect for all situations. However, studied carried out by William Scherer and Michael Scott showed that Polka manager has higher performance against several benchmarks [58]. More research work is required in order to use multiple contention management policies according to different work-loads.

5.6 Isolation and Nested Transactions

Isolation is a property which makes sure that each transaction can execute in parallel independently and its internal execution and data should be isolated and hidden from other transactions and failure of one transaction may not affect the results of other transactions. Isolation is more categorized in to strong isolation and weak isolation [65]. Inside an STM-based system there can be two types of operations, i.e., transactional and non-transactional. The non-transactional operations may have a negative effect over transactional operations which may lead to data races and inconsistencies.

Strong Isolation implies that data access is always restricted to a transaction only. However, this assumption is not much practical as transactions sometime require data which is not available inside a transaction. The idea behind strong isolation is this that the data confliction should be minimized. Moreover, high performance STM systems do not support strong isolation as it impose read and write constraints on non-transactional data which leads to runtime overhead. Consequently, a STM system may produce incorrect and un-predictable results that even a simple parallel program can produce correctly [48].

In this survey, SigTM [48] discussed in section 4.13, used strong isolation using hardware signatures. Strong Isolation is easy to implement in Hardware Transactional Memory and its performance oriented. Abidi et al. [1] presented how off-the-shelf memory protection hardware can be used to detect conflicts in non-transactional data access and ensured *Strong Isolation*. In this approach, changes to the non-transactional code are not made until a conflict is detected dynamically. Moreover, data outside the transaction is not necessarily considered as short transactions until a conflict is detected. Baugh et al. [6] have also used a fine grained memory protection hardware mechanism for achieving *Strong Isolation* in User Fault On (UFO), Hybrid TM.

It is not obvious to think that *Strong Isolation* is a desirable property for Transactional Memory or not, since *Strong Isolation* increases the complexity of the system [1].

The concept of *Weak Isolation* believes that data is divided into transactional blocks as well as non-transactional. Moreover, both sorts of data can be accessed. However, transactional data is atomic and consistent but not necessarily non-transactional. Therefore, in this case data outside a transaction can be formed in to a transaction and thus a transaction can communicate with other transaction through an interface hence reducing the possibility of data race and helps in conflict management.

In this survey it was observed that most of the TM systems support nested transactions. Nested transactions are advantageous over single-level transaction because they distribute the task and runs inside another transaction using the concept of modularity and failure tolerance. In other words, if a nested transaction fails it does not cause the whole transaction to fail. So an ideal TM system supports nested transactions. However *flattened* nested transactions actually threaten the isolation property of the transaction. Isolation implies that if one transaction fails or abort it may not affect other transactions. Similarly, in *open transactions* changes made by inner transactions may not be visible by the outer transactions, and vice versa, as it is against the essence of isolation which emphasizes that inner data of transaction may not be visible to other transactions unless the transaction commits completely with success.

5.7 Hybrid Transactional Memory Systems

There are three sorts of transactional memory systems, i.e., Software Transactional Memory systems, Hardware Transactional Memory systems, and Hybrid Transactional Memory systems. The research work so far done with regards to TM systems shows that the Hardware Transactional Memory systems are faster than Software and Hybrid TM systems. Moreover, Hybrid TM systems are faster than pure software TM systems. The Hardware TM systems suffer from memory limitations and lack of flexibility to adapt different strategies. Though, a hardware approach like FlexTM by Scott et al. [61] has tried to achieve flexibility through decoupling. FlexTM has four separate modules for *conflict-detection*, *conflict-management*, *version-management* and data update detection. FlexTM has inherited Alert-On-Update mechanism from RTM [62] to lower the data validation cost.

Software TM can backup Hardware TM systems with flexibility. In other words, both hardware and software TM systems have weaknesses which can be complemented by the strengths of each other. When a Hardware TM system runs out of memory it switches to the STM. Moreover, STM systems can handle contention management in a better way with flexibility. The STM systems have a major advantage of flexibility and ability to adapt new algorithms and strategies regarding implementation of an efficient TM system. However, STM systems also suffer from many limitations. The STM systems have not yet learnt the economical way of achieve strong isolation mechanism. The programming languages such C and C++ does not offer facilities like garbage collection for safe programming. Therefore, STM systems have to do extra work to take care of such issues, which creates a considerable performance overhead. Moreover, STM systems confront difficulties in calling third party libraries, compiled outside of the STM systems. Other source of overheads are the additional instructions which STM system execute to track read and write set for roll-back and validation.

The future of high performing flexible Transactional Memory system lies in the development of efficient Hybrid Transactional Memory systems. Some of the current Hybrid transactional memory systems like [40, 41, 43], have two modes of execution, which is not much efficient

approach. Two modes of execution require shifting from one system to another which causes time delay. Moreover, two mode of execution based hybrid system may be implemented in two ways. Either the both modes of execution are running at the same time, which requires more hardware resources. Otherwise, one mode execution would be available at a time, which implies that when we are using software mode we can not utilize the benefits of hardware mode and vice versa. However, building a compact hybrid transactional memory system is more advantageous and efficient like, SigTM [48] discussed in section 4.13. Such systems do not have two modes of execution, instead tasks and design features are divided between software and hardware. The software part is assigned those features which it can perform more efficiently, like contention management, data versioning, commit and roll back. On the other hand, conflict detection and strong isolation may be achieved through hardware part of the system.

5.8 Validity of the Study

Validity of the study is ensured by following steps

5.8.1 Reliability and Trustworthiness of the research Work

The reliability and the trustworthiness of the research work is ensured by triangulation technique, in which data is collected from multiple sources so that the information biasness could be minimized [28].

5.8.2 Validity of the research work

The validity of the research work is ensured by supporting the arguments with multiple references from IEEE, ACM research papers, journals and thesis projects from world renowned universities and institutes.

5.8.3 Rigorousness and Quality of research work

Quality of the work is ensured by discussing the arguments and results with both positive and negative information which may counter the main theme of the research work, as in real life every phenomenon has positive a negative aspect [13].

6. Future Challenges

Although a substantial amount of research has been carried-out in last one and half decade regarding Transactional Memory systems, there are still many overheads and problems that need to be further researched and investigated. This survey identifies the following areas for future research with regards to the implementation of Software Transactional Memory systems.

6.1 Achieving Strong Isolation

Strong isolation addresses the problems like inconsistent reads, lost updates, and simultaneous access by other transactions. However, strong Isolation poses constraints over the access of data and become an overhead on the performance of a transactional memory system. Software Transactional memory systems have not yet been able to find out the economical and efficient ways to achieve strong isolation with minimum overheads. However some researchers, like Abidi et al. [1] have find out an off-the-shelf memory protection hardware approach to support strong isolation. Baugh et al. [6] have also used a fine grained memory protection hardware mechanism for achieving *Strong Isolation* for their hybrid transactional memory system.

A detailed study has been carried by Blundell [9] et al. about achieving strong isolation. In brief, strong isolation itself is an overhead on the performance of transactional memory and increases the complexity of the system. The questions like whether strong isolation is a desirable property or not? Does strong isolation always preserve correctness? What are the relative benefits of the strong and weak isolation? Such questions are still an open area of future research work.

6.2 Nested Transactions

Nested transactions are desirable property in transactional memory systems as nested transactions divide the task and support failure tolerance. The open-nested transactions are more performance oriented. However, open-nested transactions are more difficult to program to maintain the consistency between the nested transactions and surrounding transactions. Rich nested transaction with low software and hardware complexities are the challenges faced by the today's transactional memory systems.

6.3 Integration Overheads

There are different software transactional memory systems. Some of them are complier based while others are library based. However, each approach suffers from integration overheads. The Software Transactional Memory systems are facing challenges in interacting with legacy code, third-party libraries and function calls to code, complied outside the STM. More research work is required to investigate smart approaches for lowering the integration overheads of the software transactional memory systems.

6.4 Performance Comparison Studies

The performance comparison studies of various transactional memory systems are few in number. In order to understand the design and performance trade offs of transactional memory systems more deeply, it is very important to conduct more performance comparison studies of various transactional memory implementations. Tabba et al. [64] performed a comparative performance testing of NZTM with several other Transactional Memory systems, e.g., LogTM-SE, BZTM, DSTM2-SF, and SCSS. However, more performance testing studies are required to investigate performance and design trade-offs of the growing number of transactional memory implementations.

6.5 Transactional Memory Benchmarks

Transactional memory benchmarks play a vital role in better development of the transactional memory systems. Transactional memory systems are tested and analyzed using these benchmarks. The performance of a certain Transactional Memory system would be clearer if it is analyzed and tested against many TM benchmarks. Some of the existing benchmarks are TM Micro benchmarks [22], SPLASH-2 [66], STMBench7 [30], STAMP [49], Lee-TM [4], The Haskell STM Benchmark suite [51], and Worm Bench [67]. Some benchmarks focus on one data structure at a time e.g., Micro benchmarks [22]. However, the realistic workloads may consist of more complex and several data structures at the same time. STAMP is major contribution with regards to TM benchmarks. STAMP is complete suite of bench-marks. Recently H. Derin et al. [14] presented a new benchmark called RMS-TM. However, Transactional Memory benchmarks are few in number. There is certainly a need for more benchmarks to be developed.

6.6 Transactional Memory Debugging and Testing Tools

Developing software transactional memory applications is a difficult job. Moreover, not much attention has been paid regarding developing of debugging tools for Software Transactional Memory systems. Moir et al. [45] has discussed different problems associated with debugging the STM and Hybrid Transactional Memory applications. H. Derin et al. [15] presented TMUNIT, a testing tool for software transactional memory applications. However, there is a clear shortage of debugging and testing tools and applications for developing Transactional Memory applications.

7. CONCLUSION

In today's era of multicore processors, high performing and flexible parallel applications are the only means of utilizing the full power of multicore processors. However developing parallel applications is not easy and requires very smart designs to avoid overheads and bottlenecks. Traditional locking approaches suffer from problems like deadlocks, lives-locks and priority inversions. Transaction memory provides an elegant mechanism for writing parallel applications. In last one and half decade, considerable amount of research has been done to design software, hardware and Hybrid transactional memory systems. The Transactional Memory systems have brought a new thought for developing parallel applications. The Transactional Memory systems provide various design features to facilitate the efficient execution of the parallel tasks, e.g. concurrency, contention management, atomicity, isolation. However, so far strong research groups are working on transactional memory systems. There is a need to develop transactional memory systems more commonly. More research is required to make the Transactional Memory systems more scalable and performance oriented.

This survey was an effort towards developing a better understanding of current state-of-the-art transactional memory systems. An overview of different transactional memory systems was presented. Moreover, their design and implementation trade-offs were discussed.

This survey drawn following trends and conclusions

1. The transactional memory systems discussed in this survey, reveal that most of the systems adopted the weak *Isolation*. Only two systems that were Hybrid TM systems, i.e., SigTM and PhTM used strong *Isolation*. The reason behind this trend could be this that the strong Isolation is easy to achieve in Hybrid approaches as Hybrid approaches use hardware based approaches to achieve strong Isolation. Moreover, in the Software Transactional Memory systems, achieving the strong *Isolation* is very complex and results in a overhead on the system performance.

2. In this survey it was observed that some software transactional memory systems supported nested transactions while others not. This trend could be due to the reason that systems were not mature enough to adopt an advance feature like nested transactions and may be their modified versions come up with support for nested transactions. Moreover, support for nested transactions threaten the *Isolation* property of the TM system and makes the system more complex and this can be another reason for not supporting the nested transactions. However, in Hybrid Transactional memory systems, most of the systems supported nested transactions. This trend may be due to the fact that supporting nested transactions in hardware based approaches is easier than software based approaches.

3. Early software transactional memory systems used the helping mechanism for conflict resolution. However, as we can see in the table 5.1, most of the systems later on have a trend of using explicit contention manager. Using explicit contention manager is a more appropriate approach as explicit contention manager are more flexible and can employ different dynamic contention management policies at the same time.

Table 7.1 The Comparison of Design Features of Transactional Memory Systems

System Name	Year Released	Synchronization Strategy	Concurrency Control	Granularity	Update Strategy	Conflict Detection	Conflict Resolution	Nested Transaction Support	Isolation
STM	1995	Non-blocking (Lock-freedom)	Pessimistic	Word	Direct Update	Early	Helping	Not Supported	Weak
WSTM	2003	Non-blocking (Obstruction-freedom)	Optimistic	Word	Deferred Update	Late	Helping	Flattened	Weak
OSTM	2003	Non-blocking (Lock-Freedom)	Optimistic	Object	Deferred Update	Late	Aborting	Not Supported	Weak
DSTM	2003	Non-blocking (Obstruction-freedom)	Optimistic	Object	Deferred Update	Early	Contention Manager	Flattened	Weak
RSTM	2006	Non-blocking (Obstruction-freedom)	Optimistic	Object	Deferred Update	Early Or Late (Selectable)	Conflict Manager	Supports Flattened	Weak
Time-Based STM	2006	Non-blocking (Obstruction-freedom)	Optimistic	Object, Word etc	Deferred Update	Early	Conflict Manager	Not Supported	Weak
DSTM2	2006	Obstruction-freedom Or Locking	Optimistic	Method	Deferred Update	Early	Conflict Manager	Not Supported	Weak
McRT-STM	2006	Locked-Based	Optimistic Read and Pessimistic Write	Object/ Word	Direct Update	Early Write/ late Read	Aborting	Closed Nested	Weak
TL2	2006	Blocking (Locked-Based)	Optimistic	Object, Word or Region	Deferred Update	Early or Late (Selectable)	Aborting	Not Supported	Weak
Draco STM	2007	Locked-Based	Optimistic	Object	Deferred and Direct Update	Late/Early	Aborting	Supported	Weak
NZTM	2007/ 2009	Non-blocking (Obstruction-freedom)	Optimistic	Object	Deferred Update	Late	Conflict Manager	Not Supported	Weak
HyTM	2006	Non-blocking (Obstruction-freedom)	Optimistic	Word	Deferred Update	Late	Conflict Manager	Supports	Weak
Hybrid TM	2006	Non-blocking (Obstruction-freedom)	Optimistic and Pessimistic	Object and Cache line	Deferred and Direct Update	Early/Late	Conflict Manager	Supports	Weak
PhTM	2007	Blocking and Non-Blocking	Optimistic and Pessimistic	Object, Word, Cache-line	Deferred or Direct Update	Late/Early	Conflict, Manager, Aborting, Helping	Some time Supports	Weak and Strong
SigTM	2007	Non-Blocking	Optimistic	Cache-line/Word	Deferred Update	Late	Aborting	Supported	Strong

4. Systems discussed in this survey used different approaches for conflict detection. However, survey reveals that early system used the *Early* conflict detection schemes while the later systems are more tend to towards Late or mix conflict detection approach, i.e., selectable Late/Early. However, it is discussed in more detail in chapter 3 that late conflict detection is more performance oriented than early conflict detection and this seems to be the reason that later Transactional Memory Systems discussed in this survey used, late conflict detection.

5. Most of the systems in this survey used deferred update system. The *deferred update* system is more performance oriented than direct update system as cost of aborting is higher in direct update system than a successful commit.

6. In hardware TM systems cache-line granularity is common while in the Software TM systems word level and object level granularity is more common. The motivation behind adopting word level granularity may be that it provides more precise sharing of the data compare to object level granularity. However, object level granularity is more understandable and visible to the programmer.

7. In this survey it was observed that except one transactional memory system almost all the systems used the Optimistic concurrency control. The Optimistic concurrency control is more performance oriented than pessimistic concurrency control as it has been discussed in detail in chapter 5.

8. The survey revealed that most of the transactional memory systems used Non-blocking Obstruction freedom synchronization approach. The reason behind this trend could be that obstruction-freedom is simple and performance oriented.

9. This survey also concluded that developing the compact Hybrid Transactional Memory systems is an efficient approach. The compact Hybrid Transactional Memory systems avoid two modes of execution. Moreover, hybrid systems are better because the disadvantages of software and hardware approaches are backed up by the advantages of each other. Therefore, combination of both makes a perfect solution.

The Transactional Memory is an open research area and yet it is not mature enough to be commercially adopted. However, the future of parallel applications has a lot to do with developing more efficient and smart Transactional memory systems.

In this survey the following contributions are made;

1. An overview of parallel programming while discussing the problems faced by traditional approaches is presented.
2. The significance of Software Transactional memory in parallel programming is presented with an overview of past research work in this regard.
3. The Software Transactional Memory concepts are discussed e.g. transactions existence and importance in database programming, software transaction memory semantics and constructs. Moreover, designs issues like nested transactions, transaction granularity, update strategies, conflict detection and management schemes are presented.
4. A thorough discussion of a number of Software and Hybrid Transactional memory systems are presented that were not previously covered by Larus and Rajwar [41].
5. The design limitations of a number of Transactional Memory systems are discussed while discussing their design and implementation trade-offs.

6. A thorough discussion is carried out about various design features of Software and Hybrid Transactional memory system while discussing various trends and their impact on the performance.
7. A number of future challenges are identified in this survey, covering different aspects of Software and Hybrid Transactional memory design and development issues.

Once again, to sum-up, this survey presents the state-of-the-art in Software and Hybrid Transactional Memory. Moreover, starting from the basic taxonomy of a transaction to complex design concepts of the Software Transactional Memory are discussed. In total, fifteen different STM and Hybrid Transactional systems are discussed with their respective limitations. Eight new systems are discussed that were not covered by the book written by Larus and Rajwar on Transactional Memory [41]. At the end, a comprehensive discussion regarding design issues of the Transactional Memory systems is presented with their design and implementation trade-offs. Finally, a number of future challenges and research areas are identified and discussed.

REFERENCES

[1] Abadi, M., Harris, T., and Mehrara, M. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, NC, USA. PPoPP '09. ACM, New York, NY, February 2009. DOI= http://doi.acm.org/10.1145/1504176.1504203

[2] Adl-Tabatabai, A., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation , Ottawa, Ontario, Canada, Jun 2006. PLDI '06. ACM, New York, NY. DOI= http://doi.acm.org/10.1145/1133981.1133985

[3] Adl-Tabatabai, A-R, Kozyrakis, C., and Saha, B. E. Unlocking concurrency: Multicore programming with transactional memory. ACM Queue, 4(10): 24-33, Dec 2006. http://doi.acm.org/10.1145/1189276.1189288

[4] Ansari, M., Kotselidis, C., Jarvis, K., Luj´an, M., Kirkham, C., and Watson, I.. Lee-tm: A non-trivial benchmark for transactional memory. In ICA3PP ’08, June 2008.

[5] Barnes, G. A method for implementing lock-free shared-data structures. In Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (Velen, Germany). SPAA '93. ACM, New York, NY, July 1993. DOI= http://doi.acm.org/10.1145/165231.165265.

[6] Baugh, L., Neelakantam, N., and Zilles, C. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. SIGARCH Comput.

[7] Bender, R. Introduction to Data Abstraction. http://mitpress.mit.edu/sicp/full-text/sicp/book/node27.html, April 2000.Archit. News, June 2008. DOI= http://doi.acm.org/10.1145/1394608.1382132

[8] B. He, W. N. Scherer, III and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In Proc. 12th Annu. IEEE Int. Conf. on High Performance Computing (HiPC), Goa, India, 2005 (http://www.cs.rochester.edu/scherer/papers/2005-HiPC-TPlocks.pdf).

[9] Blundell, C.; Lewis, E.C.; Martin, M.M.K. Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters, pages 17-17.

[10] Blundell, C., Lewis, E. C., and Martin, M. M. K. Deconstructing transactional semantics: The subtleties of atomicity. In Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking, 2005.

[11] Chandy, K. M. and Misra, J. The drinking philosophers problem. ACM Trans. Program. Lang. Syst. October 1984. DOI= http://doi.acm.org/10.1145/1780.1804

[12] Chung, J. W., Minh, C.C., McDonald, A., Skare, T., Chafi, H., Carlstrom, B. D., Kozyrakis, C. and Olukotun, K. Tradeoffs in Transactional Memory Virtualization.12th international conference on Architectural support for programming languages and operating systems, ACM Press, New York, USA, 2006. DOI= http://doi.acm.org/10.1145/1168857.1168903
[13] Creswell, J. W. W. Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, SAGE Publications July, pages 1-7, 2002
[14] Derin, H., Gokcen, K., Pascal, F., Vincent, G. and Christof, F. RMS-TM: a transactional memory benchmark for recognition, mining and synthesis applications. 4th ACM SIGPLAN Workshop on Transactional Computing TRANSACT 2009, Raleigh, North Carolina, USA. Feb 2009.
[15] Derin, H., Pascal, F., Vincent, G. and Christof, F. TMUNIT: Testing Software Transactional Memories. 4th ACM SIGPLAN Workshop on Transactional Computing TRANSACT 2009, Raleigh, North Carolina, USA February 15, 2009.
[16] Dice, D., Shalev, O. and Shavit, N. Transactional locking II. Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 204-213.
[17] Dice, D., and Shavit, N. What Really Makes Transactions Fasler?, Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), Ottawa, Canada, 2006 (http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf).
[18] Dragojevi´c, A., Guerraoui, R., Kapałka, M. Dividing Transactional Memories by Zero. School of Computer and Communication Sciences, I&C, EPFL, 2008.
[19] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA. ASPLOS-XII. ACM, New York, NY. October, 2006. DOI= http://doi.acm.org/10.1145/1168857.1168900
[20] Ennals, R. Software transactional memory should not be obstruction-free. Technical Report Nr. IRC-TR-06-052. Intel Research Cambridge Tech Report, 2006.
[21] Ennals, R.: Software transactional memory should not be obstruction-free.2005, www.cambridge.intel-research.net/rennals/notlockfree.pdf
[22] Ennals, J. R. Adaptive Evaluation of Non-Strict Programs. PhD thesis. July 2007.
[23] Fraser, K. Practical Lock-Freedom. Ph.D. Thesis, King's College, University of Cambridge, pages 15-47, September 2003.
[24] Fraser, K. and Harris, T. Language Support for Lightweight Transactions. In the proceedings of OOPSLA'03, October 2003.
[25] Fraser, K. and Harris, T. Concurrent programming without locks. ACM Trans. Comput. Syst. May 2007, DOI= http://doi.acm.org/10.1145/1233307.1233309
[26] Grahn, H. Transactional Memory: Hardware and Virtualized Hardware Approaches. Research Report, Blekinge Institute of Technology, page 1, 2009.
[27] Garcia-Molina, H., Ullman, J. D. and Widom, J., "Database Systems: The Complete Book", Prentice Hall, page 13, 2002.
[28] Golafshani, N. Understanding Reliability and Validity in Qualitative Research. University of Toronto, Toronto, Ontario, Canada, pages 598-604, Dec 2003. http://www.nova.edu/ssss/QR/QR8-4/golafshani.pdf
[29] Gottschlich, J. E. and Connors, D. A. DracoSTM: a practical C++ approach to software transactional memory. In Proceedings of the 2007 Symposium on Library-Centric Software Design. LCSD '07. ACM, New York, NY, Montreal, Canada, Oct, 2007. DOI= http://doi.acm.org/10.1145/1512762.1512768
[30] Guerraoui, R., Kapalka, M., and Vitek, J. Stmbench7: a benchmark for software transactional memory. SIGOPS '07, 2007.

[31] Harris, T. and Fraser, K. Language support for lightweight transactions. In Proc. 18th SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 03), Anaheim, CA, 2003, pages 288–402. DOI= http://doi.acm.org/10.1145/949305.949340
[32] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable Memory Transactions. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA. PPoPP '05. ACM, New York, NY, June 2005. DOI= http://doi.acm.org/10.1145/1065944.1065952
[33] Harris, T. Exceptions and side-effects in atomic blocks. Sci. Comput. Program., December 2005. DOI= http://dx.doi.org/10.1016/j.scico.2005.03.005
[34] Herlihy, M., Eliot, J., and Moss, B. Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289-300, 1993.
[35] Herlihy, M. P. and Wing, J. M. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. Jul 1990. DOI= http://doi.acm.org/10.1145/78969.78972
[36] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. Software transactional memory for dynamic-sized data structures. In Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (Boston, Massachusetts). PODC '03. ACM, New York, NY, pages 1-2, 92-101, 2003. DOI= http://doi.acm.org/10.1145/872035.872048
[37] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.
[38] Herlihy, M., Luchangco, V., and Moir, M. A flexible framework for implementing software transactional memory. SIGPLAN Not. 2005. DOI= http://doi.acm.org/10.1145/1167515.1167495
[39] Hoare, C. A. R. Towards a theory of parallel programming. In Operating Systems Techniques, vol. 9 of A.P.I.C. Studies in Data Processing, Academic Press, pages 61–71, 1972.
[40] Kumar, S., Chu, M., Hughes, C. J., Kundu, P., and Nguyen, A. 2006. Hybrid transactional memory. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, New York, USA. PPoPP '06. ACM, New York, NY, March 2006. DOI= http://doi.acm.org/10.1145/1122971.1123003
[41] Larus, J. R. and Rajwar, R. Transactional Memory, ISBN-13: 9781598291254, pages, 2007
[42] Lev, Y., Moir, M., and Nussbaum, D. PhTM: Phased transactional memory. In Workshop on Transactional Computing (Transact), 2007. http://research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf.
[43] Lomet, D. B. Process structuring, synchronization and recovery using atomic actions. Proceedings of an ACM Conference on Language Design for Reliable Software, Mar 1977, pages, 128–137. DOI= http://doi.acm.org/10.1145/800022.808319
[44] Moir, M. Hybrid Transactional Memory. Research Report, Sun Microsystems Laboratories, July 2005.
[45] Moir, M., and Lev, Y. Debugging with transactional memory. In TRANSACT'06. ACM, 2006.
[46] Marathe, V. J. and Scott, M. L. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report Nr. TR 839, University of Rochester Computer Science Dept., 2004.
[47] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. Lowering the Overhead of Software Transactional Memory. Dept. of Computer Science, Univ. of Rochester, March 2006.
[48] Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K.An effective hybrid transactional memory system with strong isolation

guarantees. SIGARCH Comput. Archit. News, June 2007. DOI= http://doi.acm.org/10.1145/1273440.1250673
[49] Minh, C.C., Chung, J., Kozyrakis, C., and Olukotun, K. Stamp: Stanford transactional applications for multi-processing. In IISWC '08, Sep 2008.
[50] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logTM. SIGPLAN Not., Nov 2006, 359-370. DOI= http://doi.acm.org/10.1145/1168918.1168902
[51] Perfumo, C., S¨onmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In CF '08, 2008.
[52] Rajwar, R. and Goodman, J. R. Speculative lock elision: enabling highly concurrent multithreaded execution. In Proceedings of the 34th Annual ACM/IEEE international Symposium on Microarchitecture ,Austin, Texas. International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, December 2001.
[53] Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: 20th International Symposium on Distributed Computing (DISC), 2006.
[54] Riegel, T., Fetzer, C., and Felber, P. Time-based transactional memory with scalable time bases. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures San Diego, California, USA,. SPAA '07. ACM, New York, NY, June 2007, DOI= http://doi.acm.org/10.1145/1248377.1248415
[55] R. Ennals. Software transactional memory should not be obstruction free. Technical report, Intel Research Tech Report, January 2006.
[56] Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: A high performance software transactional memory system for a multi-core runtime. In: To appear in PPoPP 2006.
[57] Saha, B., Adl-Tabatabai, A., Hudson, R. L., Minh, C., and Hertzberg, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '06. ACM, New York, NY. New York, New York, USA, March 2006. DOI= http://doi.acm.org/10.1145/1122971.1123001
[58] Scherer, W. N. and Scott, M. L. Advanced contention management for dynamic software transactional memory. In Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (Las Vegas, NV, USA, July 17 - 20, 2005). PODC '05. ACM, New York, NY, July 2005. DOI= http://doi.acm.org/10.1145/1073814.1073861
[59] Shavit, N. and Touitou, D. Software Transactional Memory. In the Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 0-8, August 1995.
[60] Spear, M. F., Michael, M. M., and Scott, M. L. Inevitability Mechanisms for Software Transactional Memory. In Proc. of the 3rd ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, UT, February 2008.
[61] Shriraman, A., Dwarkadas, S., and Scott, M. L. 2008. Flexible Decoupled Transactional Memory Support. In Proceedings of the 35th international Symposium on Computer Architecture. International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, June 2008. DOI= http://dx.doi.org/10.1109/ISCA.2008.17
[62] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V. J., Dwarkadas, S., and Scott, M. L. 2007. An integrated hardware-software approach to flexible transactional memory. SIGARCH Comput. Archit. News , June 2007. DOI= http://doi.acm.org/10.1145/1273440.1250676
[63] Tabba, F., Wang, C., Goodman, J. R., and Moir, M. NZTM: Non-blocking Zero-Indirection Transactional Memory. In the 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007.

[64] Tabba, F., Wang, C., Goodman, J. R., and Moir, M. NZTM: Non-blocking Zero-Indirection Transactional Memory. In the 21st ACM Symposium on Parallelism in Algorithms and Architectures, 2009, Canada.
[65] Thomasian, A.: Concurrency control: methods, performance, and analysis. ACM, Comput. Surv. pages 70–119, 1998.
[66] Woo, C., S., Ohara, M., Torrie, E., Singh, P.J., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In ISCA ’95, 1995.
[67] Zyulkyarov, F., Cvijic, S., Unsal, O., Cristal, A., Ayguade, E., Harris, T., and Valero, M. Wormbench - a configurable workload for evaluating transactional memory systems. In MEDEA ’08, 2008.