Composability of transactions using closed nesting in software transactional memory

by

© Ranjeet Kumar

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

St. John’s

Newfoundland
Abstract

With the boom in the development of multi-core machines and the development of multi-threaded applications as such, concurrent programming has gained increasingly more significance than ever before. However, concurrent programming using traditional methods such as locks, mutex and monitors is not easy, as they require a programmer to predetermine the lock management scheme for each case. This approach is error-prone. Besides, it is very difficult to trace the bugs in such programs. Software transactional memory (STM) is a new technology that solves this problem by offering automatic management of locks. As such, in recent years STM has gained a lot of attention in both industry and academia. However, most of the work in STM is restricted to non-nested transactions, while the domain of nested transactions remains largely unexplored.

One of the striking features of STM is its ability to support composability of transactions through three types of nesting, namely flat nesting, closed nesting and open nesting. In this thesis, we study the complexities involved in designing STM protocols for closed nested transactions. To this end, we extend Imbs and Raynal’s STM protocol [1], which is designed for non-nested transactions, to closed nested transactions. We propose several extensions, employing different modes of concurrency for subtransactions in the transaction tree: (i) serial execution (no concurrency) of subtransactions at each level; (ii) pessimistic concurrency control at all nodes; (iii) optimistic concurrency control at all nodes; and (iv) a mixture of optimistic concurrency control at some nodes while pessimistic concurrency control at other nodes in the same transaction tree.
Acknowledgements

The accomplishment of this thesis would not have been possible without the intelligent guidance and sustained support of my supervisor, Dr. Krishnamurthy Vidyasankar, at every stage of my Master’s program. I am truly indebted to him. I am also thankful to faculty and staff of Computer Science Department at MUN, namely- Dr. Edward Brown, Dr. Manrique Mata-Montero, Elaine Boone, Sharon Deir, Regina Edwards, and Darlene Oliver.

I am highly grateful to Sumeet Ghosh, Karan Bhawasinka, Wagdi M. Alrawagfeh, Cristy S. Hynes and Saima Siddiqui for their sustained support and constant encouragement. Finally, I thank my parents, Dileshwar Prasad and Shanti Devi, as well as my uncle/aunty- Priya Sinha, Pradeep Sinha, Amarendra D. Singh, Rajesh Khosla- for their support all through and believing in me.
## Contents

Abstract ii

Acknowledgements iii

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Why Software Transactional Memory? 1

1.2 Software transactional memory (STM) 4

1.3 Nesting in STM: transaction tree 5

1.3.1 Flat nesting 6

1.3.2 Closed nesting 6

1.3.3 Open nesting 6

1.4 Motivation 7

1.5 Contributions 8

1.6 Organization of thesis 9
2 Preliminaries: background

2.1 Transaction tree ................................................... 11
2.2 Super transaction and super tree ................................. 12
2.3 Shared objects: global copy vs local copy ...................... 12
2.4 Common features of a nested transaction in our model ....... 13
2.5 Informal discussion about linearizability of nested transactions .... 14
2.6 Concurrency control mechanism: a relevant study .......... 16
   2.6.1 Opacity ....................................................... 16
   2.6.2 Damien Imbs and Michel Raynal’s STM Protocol .......... 17
      2.6.2.1 About correctness .................................... 20

3 Computation model, base formalism, and proof outline ......... 22

3.1 Computation model ............................................... 22
3.2 Histories and base formalism .................................... 23
   3.2.1 Events and histories at shared memory level ............ 23
   3.2.2 History at transaction level ............................... 24
   3.2.3 Level-wise history .......................................... 25
3.3 Local timeline and linearization point at a level ............. 26
   3.3.1 Local timeline .............................................. 26
   3.3.2 Linearization point at a level ............................ 26
3.4 Construction of level-wise history .............................. 27
   3.4.1 External read ................................................ 27
   3.4.2 Visible read objects ....................................... 28
   3.4.3 Commit write ................................................ 28
5 ParSTM

5.1 The main idea .............................................. 70
   5.1.1 Optimistic behaviour at the global level \( t_\psi \): ............ 70
   5.1.2 Pessimistic behaviour at the nested level \( p\)-node, \( t_\pi \): .... 71
      5.1.2.1 Partial concurrency at the nested level ................. 72
      5.1.2.2 Handling deadlock situations .......................... 73

5.2 Implementing 2PL for nested transactions ......................... 78

5.3 Issue of incompatible read operations/transactions ................. 81

5.4 The protocol: ParSTM ....................................... 85
   5.4.1 Protocol .............................................. 85
   5.4.2 State of shared objects ................................. 88
   5.4.3 State of transaction ................................... 89
   5.4.4 Methods common to both root as well as non-root nodes \( t_* \) . 89
   5.4.5 Methods specific to non-root nodes \( t_\pi \) .................. 91
   5.4.6 Methods specific to root-node \( t_\rho \) ........................ 93
   5.4.7 Regarding abort of a transaction and its descendants ....... 94
   5.4.8 Optimization: abort of incompatible descendants .......... 95

5.5 Consistency checking and linearization points at level \( t \) .......... 95
   5.5.1 Consistency checking during external read operation ........ 95
   5.5.2 Linearization points of events in a level-wise history ....... 96
   5.5.3 Ordering of external read/search at \( t \) with overlapping local operations of \( t \) .................................................. 97
5.5.4 Linearization point of nested transaction .................................. 97
5.6 Proof ......................................................................................... 98
  5.6.1 Proof for committed transactions ........................................... 99
  5.6.2 Proof for aborted transactions .............................................. 108

6 HParSTM .................................................................................. 112
  6.1 Overview of HParSTM ............................................................... 112
  6.2 Discussion of contention management ....................................... 113
    6.2.1 Standard cases ................................................................. 113
      6.2.1.1 Consistency checking at the time of a read operation 113
      6.2.1.2 Avoiding cyclic conflict through transitivity across levels114
      6.2.1.3 Keeping track of incompatible read operations . . . . 116
    6.2.2 Special cases ................................................................. 116
      6.2.2.1 Tracking overwrite at intermediate ancestor level . . 117
      6.2.2.2 Significance of vts ..................................................... 118
  6.3 Protocol .................................................................................. 119
    6.3.1 Transaction state .............................................................. 121
    6.3.2 Working of HParSTM: ..................................................... 122
    6.3.3 About management of sets ............................................... 125
    6.3.4 About deadlock freedom .................................................. 125
  6.4 Consistency checking and linearization points at level t ............. 126
    6.4.1 Linearization points of events in a level-wise history .......... 126
    6.4.2 Definition of linearization point of a transaction .............. 127
  6.5 Proof ...................................................................................... 129
6.5.1 Proof for committed transactions ........................................... 129
6.5.2 Proof for aborted transactions ............................................. 139

7 MxSTM ............................................................................. 140

7.1 The main idea ........................................................................ 140

7.1.1 About nesting of transactions .............................................. 140
7.1.1.1 Behaviour of a p-node .................................................... 141
7.1.1.2 Behaviour of an o-node .................................................. 142

7.2 Design challenge ................................................................... 142

7.2.1 Handling special cases for MxSTM .................................... 143
7.2.1.1 Issue of duplicate request at a p-node ......................... 143
7.2.1.2 Solution ........................................................................ 146

7.2.2 Comparing MxSTM with ParSTM and HParSTM ............. 147
7.2.2.1 Changes w.r.t. both ParSTM and HParSTM ............... 147
7.2.2.2 Specific changes w.r.t. HParSTM ................................. 148
7.2.2.3 Specific changes w.r.t. ParSTM ..................................... 149
7.2.2.4 New methods ............................................................... 149

7.3 Protocol .............................................................................. 150

7.3.1 Pseudocode ...................................................................... 150
7.3.2 State of pessimistic read object, xpr, and helper methods ... 153
7.3.3 Methods common to o-node and p-node (tπ) ..................... 153
7.3.4 State of local objects and methods associated with p-node (tπ) 155
7.3.5 State of local objects and methods associated with o-node (tω) 156
7.3.6 About deadlock freedom .................................................. 158
7.4 Correctness .......................................................... 159
  7.4.1 Definition of linearization points of events ............... 159
    7.4.1.1 At a $p$-node $t_\pi$: .................................. 159
    7.4.1.2 At an $o$-node $t_\omega$: ............................... 160
  7.4.2 Definition of linearization point of a transaction $t$ .... 161
    7.4.2.1 At $p$-node (i.e., parent $t_\rho$ of $t$ is a $p$-node): ...... 161
    7.4.2.2 At $o$-node (i.e., parent $t_\rho$ of $t$ is an $o$-node): ...... 161
  7.4.3 Proof .......................................................... 162
    7.4.3.1 History ($\widehat{H}_\omega$) produced at an $o$-node ($t_\omega$) .... 174
    7.4.3.2 History ($\widehat{H}_\pi$) produced at a $p$-node ($t_\pi$) ....... 175

8 Conclusion and future work ...................................... 176
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
# List of Figures

1.1 Bank Example 1 ............................................................. 3  
1.2 Bank Example 2 ............................................................. 4  

2.1 Transaction trees and super tree (dotted lines denote access of shared  
objects by nodes). .......................................................... 13  
2.2 Execution of nested transactions ........................................ 16  
2.3 Traditional optimistic approach of concurrency ..................... 16  
2.4 Optimistic approach under Imbs and Raynal’s protocol ............ 19  

3.1 Transaction tree ............................................................. 27  
3.2 Level wise history of events (The read and write steps by descendants  
at a level are highlighted in **bold**.) ................................... 32  
3.3 Reading a value inconsistent w.r.t. to an ancestor (The order of events  
at different levels is indicated by the bold numbers in bracket.) .... 36  
3.4 Cyclic conflict through transitivity across levels ................. 37  
3.5 Incompatible transactions ................................................ 38  
3.6 When to abort an incompatible transaction ......................... 40
3.7 Bottom to top approach constructing histories and composing steps of
subtransactions ($\vec{t}_{12} \Rightarrow t_{12}\{t_{121}, t_{122}\}; \vec{t}_1 \Rightarrow t_1\{t_{11}, t_{12}\})$ ............. 43

3.8 Transaction tree (thick circle: committed; thin circle: active; dotted circle: aborted) ....................................... 44

4.1 Closed nested transactions (dark circle: committed; thin circle: active;
dotted circle: aborted) .................................................. 50

4.2 Linearization points of transactions .................................................. 66

5.1 Optimistic mode of concurrency at global level (single circle) and pes-
simistic mode at nested level (double circle) .............................. 71

5.2 Partial concurrency ................................................................. 72

5.3 Implementing 2PL for nested transactions: (1) cascaded locking of $x$ at
all the ancestors up to $t_1$ during $r_{t_{1111}}(t_1.x)$ (shown by dotted arrows),
and (2) unlocking $t_{1111}.x$ only upon completion of $t_{1111}$ (shown by solid
arrow). ................................................................. 79

5.4 Incompatible transactions in ParSTM ........................................... 81

6.1 Regarding consistency of read operations .............................. 113

6.2 Reading from different levels ...................................................... 115

7.1 Zones of different modes of concurrency in nested transactions (single
circle: $o$-node; double circle: $p$-node) ..................................... 141

7.2 Duplicate reads (SP: search_parent) ..................................... 143

7.3 Handling release of locks in case of abort of a transaction with duplicate
reads (SP: search_parent; UTA: unlock_to_ancestors) .............. 144
Chapter 1

Introduction

1.1 Why Software Transactional Memory?

Modern systems are often complex, dynamic and distributed in nature. As such, parallel and distributed computations have become inherent in these systems. To meet the requirements of the changing times, we have seen significant shift of paradigm in technological advancements from single core machines to multi-core machines in hardware sector. To complement the growth in the hardware sector, ever-increasing emphasis is being laid on the parallel programming paradigm to utilize the resources better for faster computations.

In concurrent programming, several threads compete to access shared data. Moreover, parallel programs execute in a non-deterministic manner and hence synchronization of concurrently executing threads is a critical issue. Otherwise, threads may read inconsistent values, or may see a thread’s intermediate values of computation. If several threads modify a resource simultaneously, then the final value may not correspond
to any thread’s computation. Mutual exclusion, facilitated by applying locks on the shared objects, is a mechanism that prevents several threads from accessing a shared resource at the same time. However, lock-based solutions have inherent drawbacks. In case of large grained locking where the set of data controlled by a single lock is too large, the concurrency is drastically hampered, whereas in case of fine grained locking, it is very difficult to manage the locks associated with each data item. To illustrate why it is not easy for programmers to manually manage the locks in parallel programming paradigm, let us consider a simple bank account example (using the usual implementation of synchronized methods) as shown in Figure 1.1.

Consider the case where several clients execute the transfer concurrently on the same (shared) account. The initial balance in the account is 0. Each client first deposits a given amount, and then withdraws the same amount. Thus, at the end of execution of a transaction, the final value of the balance should remain 0. In a multithreaded environment, the above example presents a scenario which suffers from a race condition, i.e., the concurrent threads compete with one another to gain access to the shared object (bank account). When several threads executing concurrently try to invoke the transfer method, their operations are not synchronized. In other words, the deposit and withdraw operations of one thread can be interleaved with those of another thread. Therefore, we often encounter cases in which the final value of the balance at the end of a transaction’s execution is non-zero. The non-zero value does not correspond to the expected execution of any of the threads. This will not be the case if the transactions are executed atomically in a serial fashion, i.e., the atomicity of the transactions is violated here.

One may argue that the above issue can be addressed by enclosing the deposit
Account class

class Account()
{
    int balance = 0;

    void withdraw(int n)
    {
        this.lock();
        balance -= n;
        this.unlock();
    }

    void deposit(int n)
    {
        this.lock();
        balance += n;
        this.unlock();
    }
}

Client program

void transfer(Account acc, int amount)
{
    acc.deposit(amount);
    acc.withdraw(amount);
}

Figure 1.1: Bank Example 1
and withdraw procedures within lock() and unlock() calls. That’s right, but it again emphasizes the same point. The programmer has to worry about different lock management issues for different granularity and set of locking. Concurrent programming involving manual management of locks is more error-prone, and locks are unable to support modular programming, i.e., gluing together smaller programs to form larger programs.

Hence, there is clearly a need for a system for dynamic management of locks in a parallel programming environment. This is where Software Transactional Memory (STM) comes into the picture.

### 1.2 Software transactional memory (STM)

Software Transactional Memory (STM) aims at providing a mechanism for handling low-level concurrency control for accessing shared objects in a multi-threaded environment in such a way that programmers may write programs without having to worry about the underlying concurrency management [9, 17, 10, 5]. It allows pro-
grammers to denote atomic regions declaratively, and the underlying STM system provides transactional guarantees. For example, under STM, the client code for the bank account example in Figure 1.1 would change as shown in Figure 1.2. The locking of the objects is managed dynamically by STM to ensure effectively atomic execution of the set of statements within the transactional block.

1.3 Nesting in STM: transaction tree

One of the unique features of STM is the composability of the transactions [13, 14, 8, 16, 12, 18, 7]. In other words, a set of smaller transactions can be combined to form a larger transaction through nesting. The execution of nested subtransactions can be conceptually represented by a dynamic tree called transaction tree [12], in which the transactions are related by parent-child relationship. A transaction that has no parent is termed as a root transaction. A root transaction has the highest level (0). The level of a child transaction in a transaction tree is one level lower than that of its parent. Whenever a transaction $t$ invokes a new child transaction $t'$, a new node $t'$ is added in the transaction tree as a child of $t$. When a (sub)transaction $t''$ is aborted (or discarded due to an ancestor’s abort), node $t''$ is removed from the transaction tree. The committed transactions are retained in the transaction tree.

Nested transactions are created when an atomic region is created inside an outer atomic region. The different types of nesting are (a) flat nesting (b) closed nesting, and (c) open nesting.
1.3.1 Flat nesting

In flat nesting, the steps of a flat nested subtransaction are treated as if they are steps of the root transaction itself. The commit of a subtransaction is local to its parent only, i.e., its read/write sets are merged with those of its parent. The write operations of a flat nested transaction are reflected on the global objects only when their local read/write sets are propagated to the root level through the commit of the intermediate ancestors (in any), and the root transaction commits. However, when a subtransaction aborts, the entire root transaction is aborted.

1.3.2 Closed nesting

As far as the commit of a closed nested transaction is concerned, it is similar to that of a flat nested transaction. Unlike a flat nested transaction, the abort of a closed nested subtransaction does not cause the abort of its ancestors. In event of an abort, its local read/write sets are not merged with those of its parent, thereby not affecting the state of its parent.

1.3.3 Open nesting

Here, when a nested subtransaction commits, the effects of its write steps are immediately reflected on the globally shared data [15]. Thus, its changes become visible to all other transactions in the system, although its ancestors may still be executing. In case any of its ancestors aborts, compensating actions are required to undo the effects of the changes it made to globally shared data.
1.4 Motivation

So far, a number of lock-based as well as timestamp-based protocols have been proposed for efficiently supporting non-nested transactions in STM. However, very little work has been carried out in exploiting the full potential of parallel nesting. Designing a protocol for supporting parallel nesting is not trivial; there is an inherent complexity involved in contention management across different levels and guaranteeing level-wise serial execution of nested subtransactions [6, 11, 4, 16]. As such, most of the work in STM so far has been carried out for normal (non-nested) transactions. Those that consider nested transactions support only serial execution of nested subtransactions [8, 12]. Recent works [1, 2, 3, 18] go further to support parallelism at the child-level, under the restriction that the parent transaction does not execute while it has active children.

Given the complexities involved in designing STM protocols for nested transactions, our interest lies in not only obtaining a higher degree of concurrency for nested transactions but also exploring and employing various modes of concurrency for nested transactions.

Note that this thesis deals with closed nested transactions only. It is directed towards the theoretical study of developing a comprehensive insight into the complexities involved in designing STM protocols for closed nested transactions. Therefore, the term ‘nested transaction’ henceforth will be used to mean ‘closed nested transaction’, unless specified otherwise.
1.5 Contributions

The major contributions of this thesis are as follows:

• *A detailed analysis of the complexities associated with designing STM protocols for closed nested transactions:* We examine in what ways designing an STM protocol for nested transaction varies from the one for non-nested transactions. Further, we study the various cases to be considered while designing an STM protocol for nested transactions. Finally, we also provide solutions for handling the various cases.

• *A set of protocols, for closed nested transactions, employing different modes of concurrency:* We provide a set of STM protocols for closed nested transactions in an incremental mode of development. Starting with a simple protocol (SimpSTM: Chapter 4) with no concurrency at the nested level, we progress to achieving full concurrency at all the nodes of the transaction tree. Further, we also employ a mixture of optimistic and pessimistic modes of concurrency at different nodes (MxSTM, Chapter 7).

• *A system for formally proving the correctness of nested transactions:* Formally proving the correctness of the STM protocols for nested transactions is in itself a challenging task. We provide a system for constructing (mapping) and analyzing the histories produced by nested transactions and determining their correctness (level-wise serializability).
1.6 Organization of thesis

In Chapter 2, we provide the discussion of the background. In Chapter 3, we discuss the formalism used for developing the protocols for nested transactions in later chapters.

- In these two chapters, we briefly discuss some of the concepts (semantics) of an existing work [9]. We use these semantics in each of the protocols presented in this thesis. We also describe the system model as well as the formalism used. Finally, we discuss the correctness criteria and the proof system for establishing the correctness of the STM protocols.

In each of the Chapters 4, 5, 6 and 7, we present the STM protocols for closed nested transactions, following an incremental mode of development.

- In Chapter 4, we present a simple STM protocol, SimpSTM, for nested transactions, under the constraint that nested subtransactions execute in a serial fashion, one at a time. Thus, there is no concurrency at the nested level. This protocol lays the foundation for more complex protocols that are presented in later chapters. Further, it also provides a comprehensive set of proofs, that are used (referenced) in the remaining chapters (5, 6 and 7) for establishing the correctness of the protocols in these chapters.

- In Chapter 5, we present another protocol, ParSTM, that employs optimistic approach of concurrency control at the root level, and a pessimistic one at the nested level. Thus, there is some (partial) concurrency at the nested level.
• In Chapter 6, we present a protocol called HParSTM in which an optimistic approach of concurrency control is used at each node of the transaction tree, thereby offering full concurrency at all the levels. This is the first protocol, that we know of in the literature, in which the siblings can execute concurrently along with their ancestors.

• In Chapter 7, we present a protocol, MxSTM, obtained by mixing (integrating) the protocols presented in Chapters 5 (for pessimistic part) and 6 (for optimistic part). Under this protocol, in a transaction tree, at some nodes an optimistic approach of concurrency control is followed while at others pessimistic behaviour is exhibited. This means we can obtain different degrees of concurrency at different nodes (levels).

Finally in Chapter 8, we provide conclusions and directions for future work.
Chapter 2

Preliminaries: background

2.1 Transaction tree

As stated earlier, the execution of nested subtransactions can be conceptually represented by a dynamic tree called transaction tree [12], in which the transactions are related by parent-child relationship. A transaction that has no parent is termed as a root transaction. A root transaction has the highest level (0). A root transaction operates at the global level. The level of a child transaction in a transaction tree is one level lower than that of its parent. Whenever a transaction \( t \) invokes a new transaction \( t' \), a new node \( t' \) is added as a child of \( t \) in the transaction tree. When a (sub)transaction \( t'' \) is aborted, then the subtree rooted at \( t'' \) is removed from the transaction tree. The committed (sub)transactions are retained in the transaction tree.

The children of a node \( t \) are represented as \( t_1, t_2, ..., t_k, \) etc. and denoted as \( \text{children}(t) \). The active (not yet committed/aborted) children of \( t \) are given by
activeChildren(t). We shall denote the transaction (sub)tree rooted at node $t$ as $\text{transTree}(t)$. Thus, $\text{transTree}(\text{root})$ represents the entire transaction tree. Alternatively, in case $t$ is a subtransaction, we shall invariably use $\text{subTree}(t)$ to represent $\text{transTree}(t)$. For a subtransaction $t$, the term $\text{outsideTrans}(t)$ is used to denote the $\text{transTree}(\text{root})$, excluding the $\text{transTree}(t)$. A non-leaf node is composed of one or more child transaction(s) and is called a composite transaction. In a transaction tree, all transactions other than the root transaction are nested transactions.

2.2 Super transaction and super tree

For sake of uniformity in notation, we would like to refer to globally shared objects in the same way as we refer to local objects of a transaction. For this, we associate all the globally shared objects with a highest level fictitious transaction called super transaction (denoted by $t_\psi$), such that all the transactions, previously referred to as root-level transactions, are now children of the super transaction. We call the resulting tree the super tree (see Figure 2.1).

2.3 Shared objects: global copy vs local copy

Global object $t_\psi.x$: In our model, for each shared object $x$, the system contains a global copy that is non-null valued and is accessible by all the transactions. It is not associated with the super transaction, and not with any transaction.

Local object $t.x$: This local copy of $x$ associated with $t$ is denoted by $t.x$. The object $t.x$ is accessible to $t$ as well as its descendants in the transaction tree.
2.4 Common features of a nested transaction in our model

In our model, the common semantics associated with a closed nested transaction \( t \) in a transaction tree are as follows.

1. Transaction \( t \) maintains its own local copy of an object it reads or writes.

2. While reading an object \( x \), \( t \) reads from its local copy \( t.x \). If a local copy is not available, then it tries to read from its nearest ancestor \( t' \) having a local copy of \( x \). In the worst case, \( t \) reads from \( t_{\psi}.x \). The read value should be “consistent” w.r.t. \( t \) as well as each of its intermediate ancestors up to \( t' \). (We discuss “consistency” later on.)

3. The write operations are performed in the local space.

4. The read and write operations are logged in local read set and local write set
respectively.

5. Transaction $t$ commits only if the combined steps of its committed children and its own steps are “consistent”. Otherwise it aborts.

6. If $t$ is a non-root transaction and commits, its local read and write sets are merged with the corresponding read and write sets of its parent. When a root transaction commits, its write set values are transferred to the globally shared objects. Thus, the changes due to the write steps of a committed subtransaction $t$ are reflected on the globally shared objects only when all the ancestors of $t$ commit.

7. In case of abort of $t$, its read and write sets are ignored, and no changes are made to the parent’s objects. Here, the results of $t$ as well as its descendants are discarded, and are not propagated to $t$’s ancestors. Therefore, they are not accessible to transactions in outsideTrans($t$).

### 2.5 Informal discussion about linearizability of nested transactions

*Linearizability:* An execution is linearizable if each transaction in the execution appears to have occurred at a single instant of time, called its linearization point, during its lifespan. In addition, no two transactions can have the same linearization point. In other words, the steps of the transactions can be ordered to obtain a sequential history in which transactions seem to execute serially, one after the other, in the order
of their linearization points. Here, the ordering of the steps of the transactions must respect the constraint that the linearization point of each transaction lies at some point of time within its lifespan. This constraint distinguishes linearizability from serializability.

For sake of convenience of argument, we use the expression “a transaction \( t \) is linearizable” to mean that “the steps of \( t \) are such that a unique linearization point for \( t \) can be obtained w.r.t. other transactions in a (linearizable) execution”.

*Linearizability of nested transactions:* Consider any non-leaf node \( t \) in a transaction tree. Transaction \( t \) may perform some read or write operations, or invoke a new child. Let the set of its children be denoted by \( S_c \). Then, the entire execution of each child \( t_c \) in \( S_c \) should appear to occur atomically at \( t \)'s level. Observe here that the steps of its children on \( t \)'s local objects may be interleaved with one another and with \( t \)'s own local operations. To accommodate this, consider each of the local read or write operations of \( t \) to have been carried out by a fictitious new child with that operation being its only operation. Then, the execution of each child should be such that it is linearizable with its sibling transactions. Similarly, at the root level, it is required that the overall execution of each root-level transaction is performed in a linearizable manner. These points are illustrated by Figure 2.2. Figure 2.2b depicts the level-wise serial (linear) execution for the transaction tree shown in Figure 2.2a.
Figure 2.2: Execution of nested transactions

Figure 2.3: Traditional optimistic approach of concurrency

2.6 Concurrency control mechanism: a relevant study

2.6.1 Opacity

Opacity is the most widely accepted correctness criterion for STM systems [1, 9, 11]. The consistency criterion we use for our protocols is level-wise opacity (defined in the next chapter), based on the definition of opacity for non-nested transactions. Opacity states that both committed and aborted transactions should read from a consistent state of the shared memory. The state resulting from a linearizable execution of some committed transactions is taken to be consistent. To illustrate the idea, consider history $\hat{H}_1$ under the case of traditional optimistic approach, as depicted in Figure 2.3.
\[ \widetilde{H}_1 = \langle r_{t_1}(x) \ w_{t_2}(x) \ w_{t_2}(y) \ c_{t_2} \ r_{t_1}(y) \rangle \]

First, \( t_1 \) reads \( x \). Then, \( t_2 \) modifies the values of \( x \) and \( y \), and commits after successful validation. Next, \( t_1 \) reads \( y \). Now, \( t_1 \) comes up for validation and fails because the value of \( x \) it read previously has been already overwritten by \( t_2 \). The \( x \) and \( y \) values that \( t_1 \) read are not from a consistent state.

According to opacity, we want aborted transactions also to read from consistent states only, i.e., in case of above example (\( \widetilde{H}_1 \)), \( t_1 \) should be forbidden to read from \( y \). This notion of forbidden read operation is elegantly captured in the protocol presented by Damien Imbs and Michel Raynal in [9]. The mechanism presented in [9] has been used in each of the algorithms proposed in this thesis. A clear understanding of this protocol will help in seeing through some of the complex protocols presented in the later chapters. We shall discuss this protocol in the next section.

### 2.6.2 Damien Imbs and Michel Raynal’s STM Protocol

The Protocol 2.1 has been designed for non-nested transactions. (Capital letters have been used in [9] to denote transactions and shared objects.) It uses a single copy of each base object \( x \). Each transaction has its own local copy of the base object associated with its read or write steps. (Recall that \( t.x \) denotes the local copy of object \( x \) associated with transaction \( t \), whereas \( t.\psi.x \) denotes the globally shared copy.) To keep track of the conflicts between the transactions, the following control variables are used (with slightly different notations in [9] as \( OW, RS_X, FBD_X \)).

- \( t.\psi.ow \): a (overwritten) set that contains the ids of the transactions that read some object \( t.\psi.x \) that was modified later (so, there is a conflict).
operation $\text{read}_t(x)$:
(01) if ($t.x$ does not exist) then
(02) allocate local space for $t.x$;
(03) $t.lrs \leftarrow t.lrs \cup \{x\}$;
(04) lock $t_\psi.x$; $t.x \leftarrow t_\psi.x$; $t_\psi.x.rs \leftarrow t_\psi.x.rs \cup \{t\}$; unlock $t_\psi.x$;
(05) if ($t \in t_\psi.x.fbd$) then return (abort); end if
(06) end if;
(07) return(value of $t.x$)

operation $\text{write}_t(x,v)$:
(08) $t.read\_only \leftarrow false$; // $t.read\_only$ is set to true initially.
(09) if ($t.x$ does not exist) then allocate local space for $t.x$ end if;
(10) $t.x \leftarrow v$;
(11) $t.lws \leftarrow t.lws \cup \{x\}$;

operation $\text{try\_to\_commit}_t()$:
(12) if ($t.read\_only$) then return(commit);
(13) else lock all the objects in $t.lrs \cup t.lws$ ;
(14) if ($t \in t_\psi.ow$) then release all the locks; return(abort) end if;
(15) for each $x \in t.lws$ do $t_\psi.x \leftarrow t.x$ end for;
(16) $t_\psi.ow \leftarrow t_\psi.ow \cup (\bigcup_{x \in t.lws} t_\psi.x.rs)$;
(17) for each $x \in t.lws$ do $t_\psi.x.fbd \leftarrow t_\psi.ow$; $t_\psi.x.rs \leftarrow \emptyset$; end for;
(18) release all the locks;
(19) return(commit);
(20) end if

Table 2.1: Protocol 2.1: D. Imbs and M. Raynal’s STM Protocol [9]
$t_1$ starts $t_2$ starts $r_{t_1}(x)$ $t_2$ validates & writes $w_{t_2}(x)$ $w_{t_2}(y)$ $r_{t_1}(y)$

add $t_1$ to $x.rs$ add $t_1$ to $ow$, disallow
$x.fbd, y.fbd$ (abort) as $t_1$ is in $y.fbd$

Figure 2.4: Optimistic approach under Imbs and Raynal’s protocol

t_ψ.x.rs: a (read) set associated with each shared object $t_ψ.x$. It stores the ids of the transactions that read from the object $t_ψ.x$ since its last update. Thus, the reads are visible to other transactions.

$t_ψ.x.fbd$: a (forbidden) set associated with each global object $t_ψ.x$. The predicate $t \in t_ψ.x.fbd$ means that the transaction $t$ has read an object $t_ψ.y$ that since then has been overwritten (hence $t \in t_ψ.ow$), and the overwriting of $t_ψ.y$ is such that any future read of $t_ψ.x$ by $t$ will be invalid (i.e., the value obtained by $t$ from $t_ψ.y$ and any value it will obtain from $t_ψ.x$ in the future cannot be mutually consistent).

To illustrate how these data structures, $t_ψ.x.rs, t_ψ.x.fbd$ and $t_ψ.ow$ prevent the forbidden read, consider the example of $H_1$ again through Figure 2.4. For the following discussion, all the references to line numbers are associated with Protocol 2.1. When a transaction $t_1$ performs a read operation on a shared object $t_ψ.x$, it adds its id in $t_ψ.x.rs$ (line 04). Later, when another transaction $t_2$ modifies objects $t_ψ.x$ and $t_ψ.y$, it adds $t_1$ to $t_ψ.ow$ (line 17), followed by updating both $t_ψ.x.fbd$ and $t_ψ.y.fbd$ (line 18). Now, if $t_1$ tries to access $t_ψ.y$, it will detect the conflict by noticing its id in $t_ψ.y.fbd,$
and consequently abort (due to line 05). Observe that $t_\psi.ow$ contains the ids of all the transactions that previously read any object whose value since then has been modified, and $t_\psi.x.fbd$ is updated using $t_\psi.ow$. As such, none of those transactions (in $t_\psi.x.fbd$) will be allowed to read $t_\psi.x$, or any other object that is written after the update of $t_\psi.x$. Therefore, a cyclic conflict between transactions through transitivity is not possible.

Besides, a transaction $t$ maintains two local sets, $t.lrs$ (local read set) and $t.lws$ (local write set) to document its read/write operations (lines 03, 11). Before committing, the validation phase consists of ensuring that the transaction does not belong to set $t_\psi.ow$ (line 15). However, a transaction that has no write operation is committed immediately (lines 12-13), and is thus treated differently.

### 2.6.2.1 About correctness

For each transaction $t$, its linearization point $\ell_t$, is defined as follows:

1. If a transaction $t$ aborts, $\ell_t$ is placed just before $t$ is added to the set $t_\psi.ow$ (line 17 of the try_to_commit($t$) operation of transaction that entails its abort).

2. If $t$ is a read only transaction, $\ell_t$ is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 05 of the read($t(x)$) operation) and (2) the time just before $t$ is added to $t_\psi.ow$ (if it ever is).

3. If $t$ is an update transaction that commits, $\ell_t$ is placed just after the execution of line 17 by $t$ (update of $t_\psi.ow$).

Using these linearization points, a set of proofs (listed under part (1) of Section 3.7.6) are provided in [9] to show that the history produced by this protocol is indeed
linearizable. The formalism as well as the set of proofs presented in the rest of the chapters are based along the same lines as presented in [9].

In the next chapter, we provide the formalism as well as the outline of the proof system used for showing the correctness of the protocols discussed in this thesis.
Chapter 3

Computation model, base formalism, and proof outline

3.1 Computation model

Our system is similar to the one used in [9]. The computational model consists of processes, base objects, locks and atomic registers. There are $n$ asynchronous sequential processes (i.e., threads) denoted $p_1, ..., p_n$ that cooperate through base read/write atomic registers and locks. Each of the shared objects is protected by an individual lock. Each process is made up of a sequence of transactions, and has its own memory. The processes issue transactions one at a time.

A transaction is a sequence of read and write operations that can examine (read) and modify (write), respectively, the state of the base objects. It consists of a sequence of events that are an operation invocation, an operation response, a commit invocation, a commit response, and an abort event. An operation is considered terminated if its
response event has occurred. Similarly, a transaction is considered *completed* if its commit response or abort event has occurred.

Further, each transaction satisfies the following constraints: (1) a transaction performs one operation at a time, and (2) a composite transaction must wait until all of its children have completed before entering the validation for its commit.

The set of transactions is denoted by $T$. The set of objects is denoted by $X$ and the set of possible values associated with them is $V$.

### 3.2 Histories and base formalism

#### 3.2.1 Events and histories at shared memory level

There is an event associated with each operation on a shared memory (objects, locks, sets). Let $\mathcal{H}$ be the set of all events produced by the STM system. Each access on a shared object is atomic. As such, there is a total order on the events in $\mathcal{H}$. Thus, at the shared memory level, an execution can be represented by the pair $\hat{\mathcal{H}} = (\mathcal{H}, <_\mathcal{H})$, where $<_\mathcal{H}$ denotes the total ordering on its events. $\hat{\mathcal{H}}$ is called *shared memory history*.

We use the following notations (similar to those used in [9]).

- $B_t$ marks the event associated with the beginning of transaction $t$, and $E_t$ denotes its completion. $E_t$ can be of two types $A_t$ or $C_t$, where $A_t$ is the event “abort of $t$”, and $C_t$ is event “commit of $t$”.

- $AL_{t_i}(t_j.x, op)$ denotes the event (response) associated with the acquisition of the lock on $t_j.x$, issued by transaction $t_i$ during an invocation of operation $op$, where $op$ is $read_{t_i}(x)$, or $try\_to\_commit_{t_i}()$ (abbreviated in the proof as $ttc$).
- Similarly, we have $RL_{t_i}(t_j.x, op)$ denoting the corresponding event (response) associated with the release of the lock.

As each access to a shared object is atomic in nature (due to use of either an atomic register or a lock), we denote each read or write operation as a single event for sake of simplicity.

- $r_{t_i}(t_j.x, v)$ denotes the read operation performed by transaction $t_i$ on the object $t_j.x$ (i.e. object $x$ of transaction $t_j$), where $v$ is the value returned by that read operation.

- $w_{t_i}(t_j.x, v)$ denotes the write of value $v$ by transaction $t_i$ on the object $t_j.x$.

For sake of simplicity, in the histories discussed in this thesis, $B_t$ has been omitted as it can be intuitively inferred to occur just before the first read/write operation of transaction $t$. However, $C_t$ or $A_t$ has been used as a single event to mark the end (commit or abort) of the transaction.

### 3.2.2 History at transaction level

Given an execution, let $H'$ denote the set of transactions issued during that execution. The order relation between the transactions in $H'$, denoted by $\rightarrow_{H'}$, is defined as follows: $t_1 \rightarrow_{H'} t_2$ if $t_1$ ends before $t_2$ begins, and $t_1$ and $t_2$ are concurrent if $t_1 \nrightarrow_{H'} t_2$ and $t_2 \nrightarrow_{H'} t_1$. Thus, at the transaction level, the execution is defined by a partial order $\widehat{H'} = \langle H', \rightarrow_{H'} \rangle$, that is called transaction history [9].

The reads from relation between transactions, denoted $\rightarrow_{rf}$, is defined as: $t_1 \xrightarrow{t.x}_{r_f} t_2$, if $t_2$ read a value written by $t_1$ in $t$’s object $t.x$. 

24
A transaction history $\widehat{H}^\sigma = (H^\sigma, \to_{H^\sigma})$ is sequential if no two of its transactions are concurrent. Hence, in a sequential history, for $t_1 \neq t_2$, $t_1 \leftrightarrow_{H^\sigma} t_2 \iff t_2 \to_{H^\sigma} t_1$, that is, $\to_{H^\sigma}$ is a total order. A sequential transaction history is legal if each of its read operations returns the value of the most recent write on the same object.

A sequential transaction history $\widehat{H}^\sigma$ is equivalent to a transaction history $\widehat{H}'$ if (1) $H^\sigma = H'$ (i.e., they are made of the same transactions with the same invocations and the same responses), and the total order $\to_{H^\sigma}$ respects the partial order $\to_{H'}$ (i.e, $\to_{H'} \subset \to_{H^\sigma}$).

A transaction history $\widehat{H}^\lambda$ is linearizable if there exists a history $\widehat{H}^{\text{seq}}$ that is sequential, legal and equivalent to $\widehat{H}^\lambda$.

The set of transactions that commit in $\widehat{H}$ are given by $\text{committed}(\widehat{H})$, and the aborted ones are given by $\text{aborted}(\widehat{H})$. The set of transactions that have committed or aborted is given by $\text{complete}(\widehat{H})$. The history restricted to committed transactions is denoted by $\text{permanent}(\widehat{H})$ or $\Pi(\widehat{H})$.

### 3.2.3 Level-wise history

As discussed earlier, each node $t$ in the transaction tree has its own set of local copies of objects. These objects are accessible (shared) by $t$ and its descendants. The formalism discussed in Sections 3.2.1 and 3.2.2 can be applied to these objects as well. Thus, at each level (node) of the super tree, we have a separate history. We call this history a level-wise history. Here, $H_t$ denotes the set of all events associated with the objects of node $t$, and $<_{H_t}$ gives the total order on these events. Thus, a level-wise
shared memory history is given by $\widehat{H}_t = \langle H_t, <_{H_t} \rangle$. Similarly, $\widehat{H}'_t = \langle H'_t, \rightarrow_{H'_t} \rangle$ is used to denote a level-wise transaction history at $t$, whereas $\widehat{H}_t^\sigma = \langle H_t^\sigma, \rightarrow_{H_t^\sigma} \rangle$ denotes a level-wise sequential transaction history.

3.3 Local timeline and linearization point at a level

3.3.1 Local timeline

With each transaction $t$ in the transaction tree, we associate a notion of local timeline spanning the lifespan of that transaction, and it is denoted by $\tau_t$. An instant of time, $i$, in $\tau_t$ is denoted by $\tau^i_t$. Let $t_c$ be a child of $t$. Then, the linearization point, $\ell_{t_c}$, for $t_c$ is the time in $\tau_t$ at which the entire execution of $t_c$ can be treated to have occurred. Thus, the linearization point of a child transaction is defined within the lifespan of itself as well as its parent. The time corresponding to $\ell_{t_c}$ in $\tau_t$ is denoted by $\tau^\ell_{t_c}$. Similarly, as the accesses to shared object $t.x$ are atomic in nature, the times corresponding to $r_{t_c}(t.x)$ and $w_{t_c}(t.x)$ in $\tau_t$ can be denoted by $\tau^{r_{t_c}(x)}_t$ and $\tau^{w_{t_c}(x)}_t$ respectively. The timeline associated with $t_\psi$ is taken to be infinite.

3.3.2 Linearization point at a level

To determine the linearization point of a transaction $t$ at its parent’s ($t_p$) level in the super tree, we do not need to consider $t$’s steps at all the levels; we only need to take into account $t$’s steps on its ancestors’ objects.

For example, consider the transaction subtree rooted at $t_{12}$ in Figure 3.1, and let us denote it by $TR_{t_{12}}$. For determining the linearization point for $t_{12}$ at $t_1$’s level,
we can treat $TR_{t_{12}}$ as a single transaction. The steps of the transactions within the nodes of $TR_{t_{12}}$ are local to $TR_{t_{12}}$ and therefore do not affect $TR_{t_{12}}$’s linearizability at $t_1$’s level. The steps of $TR_{t_{12}}$ (those of $t_{12}, t_{121}, t_{122}$) on the objects of $t_1$ and $t_\psi$ are important.

### 3.4 Construction of level-wise history

#### 3.4.1 External read

While reading, if $t$ does not have a local copy of an object $x$, then it tries to read the value of $x$ from its nearest ancestor having a (non-null valued) local copy of $x$. Such a read operation that takes place outside the local space of $t$ is termed as *external read* operation of $t$. 

![Figure 3.1: Transaction tree](image)
3.4.2 Visible read objects

A subtransaction $t$ may have read an object $x$ from its ancestor $t'$, other than its parent $t_p$. Transaction $t_p$ may not have a local copy $t_p.x$. When $t$ commits, it creates $t_p.x$ using its local copy $t.x$. Thus, the value of $t_p.x$ at this point corresponds to the one obtained in the read step of $t$. Subsequently, this value of $t_p.x$ can be accessed by $t_p$ and its other descendants. In other words, the read of $t$ becomes visible at its parent $t_p$.

Example: With reference to Figure 3.1, \( \hat{H}_1 = \langle r_{t_{121}}(t_{11}.x, 0), c_{t_{121}} r_{t_{122}}(t_{12}.x, 0) \rangle \)

Here, when $t_{121}$ commits and merges with $t_{12}$, the value 0 of $t_{121}.x$ (that it read from $t_{1}.x$) is made available in its parent’s corresponding local copy $t_{12}.x$. Therefore, $t_{122}$ is able to read the value 0 from $t_{12}.x$.

Note: We can notice here that a subtransaction reads a value from the nearest local copy of the object visible to it. That nearest copy could be made available due to a visible read operation or due to a write operation.

If $t$ aborts, then no merging of its read steps’ values with those of its parent takes place. Therefore, its read steps are not visible at the parent or any ancestor’s level.

3.4.3 Commit write

When a closed nested subtransaction $t$ commits and merges with its parent $t_p$, for each object $x$ that is updated in its local space, $t$ updates $t_p.x$ with the value of $t.x$ at
the time of committing. We refer to this write operation as *commit write*. Note that the value of \( t.x \) at the time of \( t \)'s commit corresponds to the “last write” operation that took place on \( t.x \).

Again with reference to Figure 3.1,

**Example:** \( \hat{H}_2 = \langle w_{t_{121}}(t_{121}.x, 0) \, c_{t_{121}} \, w_{t_{12}}(t_{12}.x, 1) \, w_{t_{12}}(t_{12}.y, 2) \, w_{t_{122}}(t_{122}.y, 3) \, c_{t_{122}} \, c_{t_{12}} \rangle \).

Observe that \( x \) is first updated by \( t_{121} \), and then by \( t_{12} \) itself. Therefore, when \( t_{12} \) commits, the last write on \( t_{12}.x \) corresponds to \( w_{t_{12}}(t_{12}.x, 1) \). Similarly, the last write on \( t_{12}.y \) corresponds to \( w_{t_{122}}(t_{122}.y, 3) \).

Now, upon commit, \( t_{121}, t_{122} \) and \( t_{12} \) use the values corresponding to the respective last writes on their local objects \( x \) and \( y \) to update respective objects \( x \) and \( y \) of their respective parents. Therefore, \( \hat{H}_2 \) can be extended as follows to reflect the commit writes at the parent level.

\[
\hat{H}_2 = \langle w_{t_{121}}(t_{121}.x, 0) \, w_{t_{121}}(t_{12}.x, 0) \, c_{t_{121}} \, w_{t_{12}}(t_{12}.x, 1) \, w_{t_{12}}(t_{12}.y, 2) \, w_{t_{122}}(t_{122}.y, 3) \, w_{t_{122}}(t_{122}.y, 3) \, c_{t_{122}} \, w_{t_{12}}(t_{1}.x, 1) \, w_{t_{12}}(t_{1}.y, 3) \, c_{t_{12}} \rangle.
\]

Observe that \( w_{t_{121}}(t_{12}.x, 0), w_{t_{122}}(t_{12}.y, 3), w_{t_{12}}(t_{1}.x, 1) \) and \( w_{t_{12}}(t_{1}.y, 3) \) shown in bold fonts here are commit writes.

### 3.4.4 Mapping of level-wise history

The level wise history of events at node \( t \) (\( \hat{H}_t \), defined in Section 3.2.3) includes the following steps:

i. *read operations* performed on \( t \)'s local objects by \( t \) or its committed descendants.

ii. *local write* operations by \( t \) on its local objects.

iii. *commit write* operations on \( t \)'s local objects by \( t \)'s committed children.
iv. *external read* operations on t’s ancestors’ objects by t and its committed descendants. To take into account these external read operations, we treat them as if they took place on t’s corresponding objects.

Our interest lies in showing that $\hat{H}_t$ is equivalent to a sequential history of t’s children and its local operations. The operations on t’s objects by t’s descendants may be interleaved with t’s own local operations on its objects. With this end in view, $\hat{H}_t$ is obtained after employing the following transformation using a mapping function $f_m(\hat{H}_t^{or})$, where $\hat{H}_t^{or}$ is the original history of events involving node t and its descendants. We call the resulting history *mapped level-wise history* or simply *level-wise history*.

The following transformations (rules) are employed through $f_m(\hat{H}_t^{or})$:

1. **Mapping local read/write operations of t:** Each operation $op_t(t.x)$ (where $op$ denotes read or write) performed by t on its local object t.x is considered to have been successfully performed by a (different) fictitious child transaction $t'$ whose only operation is $op_{t'}(t.x)$ and has “committed”. For example, local operations of t in $H_t^1 = \langle r_t(t.x) \; w_t(t.x,v_1) \; w_t(t.y,v_2) \; r_t(t.x) \rangle$ are mapped respectively to $H_t^2 = \langle r_{t_1}(t.x) \; c_{t_1} \; w_{t_2}(t.x,v_1) \; c_{t_2} \; w_{t_3}(t.y,v_2) \; c_{t_3} \; r_{t_4}(t.x) \; c_{t_4} \rangle$, where $t^1, t^2, t^3$ and $t^4$ are the fictitious children of t. This way the local (read or write) steps of t also are transformed into the steps of its children.

2. **Mapping operations of t’s descendants on t’s objects:** Let $t_c$ be a child of t, and $t_d$ be a descendant of $t_c$. Then, the read operation performed on t.x by $t_d$ is taken to have been performed by $t_c$ (for the reason that $t_d$ is a part of $t_c$ after all). For example, a history $H_t^3 = \langle r_{t_c}(t.x) \; r_{t_d}(t.y) \rangle$ is mapped to $H_t^4 = \langle r_{t_c}(t.x) \; r_{t_c}(t.y) \rangle$.
This means, in \( \widehat{H}_t \) we work as if all the steps on \( t \)'s local objects were performed only by its children.

3. **Mapping external reads of \( t \) and its descendants on objects of \( t \)'s ancestors**: Let \( t' \) be an ancestor of \( t \). Then, a read operation on \( t'.x \) by \( t_c \) or its descendant (\( t_d \)) is treated as if it was performed by \( t_c \) on \( t \)'s object \( t.x \). For example, \( r_{t_d}(t'.x) \) is mapped to \( r_{t_c}(t.x) \). In case this read operation was performed by \( t \) itself, then we first apply step (1) to transform this operation into step of its fictitious (committed) child transaction, and then apply step (2). (This way we incorporate the external read operations made by transactions in \( \text{transTree}(t) \) at the level of \( t \)'s ancestors into \( \widehat{H}_t \).) For example, \( \langle r_{t_d}(t'.x) \ r_{t_c}(t'.y) \ r_{t}(t'.z) \rangle \) is mapped to \( \langle r_{t_c}(t.x) \ r_{t_c}(t.y) \ r_{t}(t.z) \rangle \) in \( \widehat{H}_t \).

For better understanding consider the following example, using Figure 3.1. For sake of simplicity, we shall consider the history of the committed transactions only. We shall assume that a transaction begins just before it executes its first operation. Further, \( c_t \) and \( a_t \) denote \( t \)'s commit and abort respectively.

Now, we consider the execution shown in Figure 3.2 and construct the corresponding history (\( \widehat{H}_3 \)) for it. This history is constructed by considering, at each level \( t \), the following steps on its object \( t.x \): (a) the read operations performed by \( t \)'s descendants, (b) the commit-writes performed by the committed children, and (c) the local read and write operations performed by \( t \) itself. The construction of the resulting history is illustrated as follows. The external reads and commit writes by descendants at ancestors’ levels have been highlighted using **bold** font to improve readability of the history. At a parent level \( t_i \), \( \left[ t_j : \{\} \ \mathbf{w}_{t_j}(t_x) \ldots c_{t_j} \right] \) denotes the place holder for
Figure 3.2: Level wise history of events (The read and write steps by descendants at a level are highlighted in **bold**.)
subsequence produced with the commit (merging of) the child \( t_j \). Here, \{\} points to the history produced at node \( t_j \) and \( w_{t_j}(t_{i1}x) \) is a commit write. The subsequence pertaining to \( t_j \) at \( t_i \)'s level is completed by inserting the history produced at node \( t_j \) in place of \{\}. For example, \( t_2 : \{\} w_{t_2}(t_{\psi}x) c_{t_2} \) at \( t_{\psi} \)'s level is completed using \( w_{t_2}(t_{21}x) w_{t_2}(t_{\psi}x) c_{t_2} \) (underlined part is the subsequence produced at \( t_2 \) and is put in place of \{\}). Following this approach in bottom to top manner, we obtain the resulting history equivalent to \( \hat{H}_3 \) given below.

\[
\hat{H}_3 = \langle w_{t_2}(t_{21}x) w_{t_2}(t_{\psi}x) c_{t_2} r_{t_{11}}(t_{\psi}x) r_{t_{11}}(t_{\psi}y) r_{t_{121}}(t_{\psi}z) w_{t_{11}}(t_{11}x) r_{t_{11}}(t_{11}x) \rangle \\
\langle w_{t_{11}}(t_{11}x) c_{t_{11}} r_{t_{121}}(t_{11}y) r_{t_{122}}(t_{11}x) w_{t_{121}}(t_{121}x) w_{t_{121}}(t_{121}y) c_{t_{121}} w_{t_{12}}(t_{12}x) r_{t_{12}}(t_{12}y) \rangle \\
\langle r_{t_{122}}(t_{12}y) c_{t_{122}} w_{t_{12}}(t_{12}y) c_{t_{12}} w_{t_{1}}(t_{1}x) w_{t_{1}}(t_{\psi}x) w_{t_{1}}(t_{\psi}y) c_{t_{1}} \rangle \\
\hat{H}_{t_{12}} = \langle r_{t_{122}}(t_{12}z) r_{t_{121}}(t_{12}y) r_{t_{122}}(t_{11}x) w_{t_{121}}(t_{12}x) c_{t_{121}} w_{t_{12}}(t_{12}y) c_{t_{12}} r_{t_{12}}(t_{12}y) c_{t_{12}} r_{t_{12}}(t_{12}y) c_{t_{12}} \rangle \\
\Rightarrow \hat{H}^\sigma_{t_{12}} = \langle t_{121}, t_{12}^1, t_{12}^2 \rangle \ (\text{Recall that } t_{12}^1, t_{12}^2 \text{ here denote the fictitious children of } t_{12}, \text{ representing its local read and write operations.})
\]

\{\textit{Observation :}\}

- \( r_{t_{122}}(t_{12}z) \) due to \( r_{t_{122}}(t_{\psi}z) \) (Rule 3);
- \( r_{t_{121}}(t_{12}y) \) due to \( r_{t_{121}}(t_{1}y) \) (Rule 3);
- \( r_{t_{122}}(t_{12}x) \) due to \( r_{t_{122}}(t_{1}x) \) (Rule 3);
\( w_{t_{12}} (t_{12}, y) \cdot c_{t_{12}}, r_{t_{12}} (t_{12}, y) \cdot c_{t_{12}} \) due to \( w_{t_{12}} (t_{12}, y) \) and \( r_{t_{12}} (t_{12}, y) \) respectively (Rule 1)

\[
\widehat{\mathcal{H}}_{t_{1}} = \langle r_{t_{11}} (t_{\psi}, x) \cdot r_{t_{11}} (t_{\psi}, y) \cdot r_{t_{122}} (t_{\psi}, z) \cdot w_{t_{11}} (t_{1}, x) \cdot c_{t_{11}}, r_{t_{121}} (t_{1}, y) \cdot r_{t_{122}} (t_{1}, x) \cdot w_{t_{12}} (t_{1}, y) \cdot c_{t_{12}} \cdot w_{t_{1}} (t_{1}, x) \rangle
\]

After mapping:

\[
\widehat{\mathcal{H}}_{t_{1}} = \langle r_{t_{11}} (t_{1}, x) \cdot r_{t_{11}} (t_{1}, y) \cdot c_{t_{11}} \cdot r_{t_{12}} (t_{1}, y) \cdot r_{t_{12}} (t_{1}, x) \cdot w_{t_{12}} (t_{1}, y) \cdot c_{t_{12}} \cdot w_{t_{1}} (t_{1}, x) \cdot c_{t_{12}} \rangle
\]

\[
\Rightarrow \widehat{\mathcal{H}}_{t_{1}} = \langle t_{1}, t_{11}, t_{12}, t_{1}^{2} \rangle
\]

{Observation :}

\( r_{t_{11}} (t_{1}, x) \) due to \( r_{t_{11}} (t_{\psi}, x) \) (Rule 3);

\( r_{t_{11}} (t_{1}, y) \cdot c_{t_{1}} \) due to \( r_{t_{1}} (t_{\psi}, y) \) (Rule 1);

\( r_{t_{12}} (t_{1}, z) \) due to \( r_{t_{122}} (t_{\psi}, z) \) (Rule 3);

\( r_{t_{12}} (t_{1}, y) \) due to \( r_{t_{121}} (t_{1}, y) \) (Rule 2);

\( r_{t_{12}} (t_{1}, x) \) due to \( r_{t_{122}} (t_{1}, x) \) (Rule 2);

\( w_{t_{12}} (t_{1}, x) \cdot c_{t_{1}} \) due to \( w_{t_{1}} (t_{1}, x) \) (Rule 1)

\[
\widehat{\mathcal{H}}_{t_{\psi}} = \langle w_{t_{2}} (t_{\psi}, x) \cdot c_{t_{2}}, r_{t_{11}} (t_{\psi}, x) \cdot r_{t_{1}} (t_{\psi}, y) \cdot r_{t_{122}} (t_{\psi}, z) \cdot w_{t_{1}} (t_{\psi}, x) \cdot w_{t_{1}} (t_{\psi}, y) \cdot c_{t_{1}} \rangle
\]

After mapping:

\[
\widehat{\mathcal{H}}_{t_{\psi}} = \langle w_{t_{2}} (t_{\psi}, x) \cdot c_{t_{2}}, r_{t_{1}} (t_{\psi}, x) \cdot r_{t_{1}} (t_{\psi}, y) \cdot r_{t_{1}} (t_{\psi}, z) \cdot w_{t_{1}} (t_{\psi}, x) \cdot w_{t_{1}} (t_{\psi}, y) \cdot c_{t_{1}} \rangle
\]

\[
\Rightarrow \widehat{\mathcal{H}}_{t_{\psi}} = \langle t_{2}, t_{1} \rangle
\]

{Observation :}

\( r_{t_{2}} (t_{\psi}, x) \) due to \( r_{t_{11}} (t_{\psi}, x) \) (Rule 2);

\( r_{t_{12}} (t_{\psi}, z) \) due to \( r_{t_{122}} (t_{\psi}, z) \) (Rule 2)
Now that we have obtained the serial order for transactions at all the levels, the serializability of the transactions at the nested levels can be represented in the following manner. The ordering of the children, $t_1$ and $t_2$, of a transaction $t$ can be shown by enclosing them within the {...} brackets immediately after $t$, e.g., $t\{t_1, t_2\}$. Thus, serializability of the entire execution can be shown as follows: $t_2, t_1\{t_1, t_{11}, t_{12}\{t_{121}, t_{122}, t_{122}\}\}$.

**Note:** The composition of level-wise histories at different levels is done in the bottom to top order. The steps of the children are composed to obtain the history at the parent level.

### 3.5 About correctness of nested transactions

#### 3.5.1 Avoiding cyclic conflict between transactions across levels

Considering the level-wise history individually cannot guarantee the consistency of the overall state of the transaction tree. It is quite possible that, in the level-wise histories, a subtransaction $t$ is linearizable at its parent level, $t_p$. However, when $t$'s steps are taken as part of $t_p$ at another ancestor's level, $t$ renders $t_p$ non-linearizable at the higher level due to a cyclic conflict.

Hence, to ensure the correctness, the STM protocols must guarantee that there is no cyclic conflict between transactions across different levels. The various cases are discussed as follows. In this section, we only consider the partial history relevant to the discussion. *Observe that this scenario does not apply to non-nested transactions,*
and is specific to nested transactions.

1. Reading a value that is inconsistent w.r.t. an intermediate ancestor

Consider the partial history $\hat{H}_4$ depicted in Figure 3.3.

$$\hat{H}_4 = \langle r_{t_{12}}(t_1.x), w_{t_{11}}(t_1.x), w_{t_{11}}(t_1.y), c_{t_{11}}, r_{t_{121}}(t_1.y) \rangle$$

Here $t_{12}$ reads $t_1.x$. Then, $t_{11}$ modifies $t_1.x$ and $t_1.y$. At this point, the value of $t_1.y$ becomes inconsistent for $t_{12}$. Next, $t_{121}$ tries to read $t_1.y$. Observe here that $t_{121}$ is a part (child) of $t_{12}$. If it commits, its steps would become part of its ancestor $t_{12}$’s steps. That means $t_{121}$’s step $r_{t_{121}}(t_1.y)$ would render its ancestor $t_{12}$ non-linearizable at higher level, $t_1$, in the following manner.

Imagine $t_{121}$ commits and merges with $t_{12}$ after $r_{t_{121}}(t_1.y)$. Now, $r_{t_{121}}(t_1.y)$ can be replaced by $r_{t_{12}}(t_1.y)$ (due to merging of steps of $t_{121}$ with $t_{12}$). With this
transformation, history $\overrightarrow{H_4}$ maps to $\overrightarrow{H_4'} = \langle rt_{t_{12}}(t_1.x), wt_{t_{11}}(t_1.x), wt_{t_{11}}(t_1.y), ct_{t_{11}}, rt_{t_{12}}(t_1.y) \rangle$.

In $\overrightarrow{H_4'}$, observe that $t_{12}$ is not linearizable with $t_{11}$. The value of $t_1.x$ read by $t_{12}$ is modified later by $t_{11}$, requiring the serial order $t_{12}, t_{11}$. However, $t_{12}$ read $t_1.y$ after $t_{11}$ modified it. Thus, we get the serial order $t_{11}, t_{12}$. Hence, there is a cycle between $t_{11}$ and $t_{12}$.

**Remark:** When a transaction $t$ reads an object $t'.x$ from an ancestor $t'$, its value should be consistent w.r.t. not only $t$ but also each of $t'$s intermediate ancestors in the path from $t$ to $t'$.

2. *Reading inconsistent value through transitivity*

Consider the following history depicted in Figure 3.4.

$$\overrightarrow{H_5} = \langle rt_{t_{121}}(t_1.x)\ wt_{t_{11}}(t_1.x)\ wt_{t_{11}}(t_1.y)\ ct_{t_{11}}\ rt_{t_{12}}(t_1.y) \rangle$$

First, $t_{121}$ reads from $t_1.x$. Later, $t_{11}$ commits after modifying $t_1.x$ and $t_1.y$. Next, $t_{12}$ reads $t_1.y$. At this point, considering $t_{121}$ is a descendant (part) of $t_{12}$, there is cyclic conflict between $t_{11}$ and $t_{12}$. The value of $t_1.x$ read by $t_{121}$ is later
modified by \( t_{11} \), requiring the serial order \( t_{12}, t_{11} \). Next, \( t_{12} \) read \( t_{1}.y \) after \( t_{11} \) modified \( t_{1}.y \), thus giving the serial order \( t_{11}, t_{12} \). Hence, there is a cycle. Note that, \( t_{121} \) being a part (child) of \( t_{12} \), if \( t_{121} \)'s read steps are merged with those of \( t_{12} \), it brings \( t_{12} \) in cyclic conflict with \( t_{11} \) and renders it non-linearizable at \( t_{1} \)'s level.

**Remark:** The STM protocol for nested transactions should ensure that no cyclic relationship occurs between transactions across different levels through transitivity.

3. **Incompatible transactions**

Consider the example depicted in Figure 3.5. The corresponding history is as follows.

\[
\mathcal{H}_6 = \langle r_{t_{121}}(t_{1}.x), w_{t_{11}}(t_{1}.x), w_{t_{11}}(t_{1}.y), c_{t_{11}}, r_{t_{122}}(t_{1}.y) \rangle
\]

Here, \( t_{121} \) first reads from \( t_{1}.x \). Later, \( t_{11} \) modifies \( t_{1}.x \) and \( t_{1}.y \). Next, \( t_{122} \)
reads $t_1.y$. Observe that $t_{11}$ modified $t_1.x$ which was previously read by $t_{121}$, giving the order $t_{121}, t_{11}$. Next, $t_{122}$ read $t_1.y$ after $t_{11}$ modified it. This gives the serial order $t_{11}, t_{122}$. Now, note here if both $t_{121}$ and $t_{122}$ are allowed to commit and merge their read sets with that of $t_{12}$, then it brings $t_{12}$ now in cyclic conflict with $t_{11}$ and renders the history non-linearizable at $t_1$'s level. Here, the two read operations, $r_{t_{121}}(t_1.x)$ and $r_{t_{122}}(t_1.y)$ are mutually incompatible at the higher level, and hence are called *incompatible read operations*, and the two transactions, $t_{121}$ and $t_{122}$, are called *incompatible transactions*. (We shall revisit incompatible transactions and formally define them in Chapter 5.)

**Incompatibility point**

Let $t$ and $t'$ be two transactions such that $t'$ is a descendant of $t$ and $t'$ is incompatible with $t$. Then, the incompatibility point of $t'$ at $t$'s level is defined as the earliest instant of time in $\tau_t$ at which $t'$ becomes incompatible with $t$, and is denoted by $\tau^{i_{t'}}_t$.

**Remark:** The STM protocol should ensure that at any point of time, the read set of a transaction does not contain incompatible read operations. To this end, we observe the following two constraints: (i) a transaction is not allowed to perform a read operation from an ancestor that is not compatible with it, and (ii) two incompatible child transactions are not both allowed to merge (commit) with the parent.
3.5.2 When to abort an incompatible subtransaction

Consider the execution depicted in Figure 3.6 and observe that the read operations \( r_{t_{111}}(t_1.x) \) and \( r_{t_{112}}(t_1.y) \) are incompatible. Therefore, \( t_{111} \) and \( t_{112} \) are mutually incompatible. Incompatibility comes into picture here at the time of \( r_{t_{112}}(t_1.y) \) (say \( \tau^1 \)). Note that the two incompatible transactions in picture here are not related by ancestor-descendant relation. Now the question here is whether we should abort one of the two incompatible transactions at time \( \tau^1 \), given the case that \( t_{111} \) and \( t_{112} \) here are not related by ancestor-descendant relation at time \( \tau^1 \). The answer is no.

At time \( \tau^1 \), we cannot abort \( t_{112} \) because \( t_{111} \) is not part of its ancestor yet, and hence its step \( r_{t_{112}}(t_1.y) \) is consistent w.r.t. its ancestors \( t_{11} \) and \( t_1 \). Similarly, we cannot abort \( t_{111} \) as \( t_{112} \) is not part of its ancestor yet. Thus, at this point, the reads of each of the subtransactions are consistent w.r.t. its ancestor. There is no point in forcefully aborting one of the subtransactions as we cannot guarantee that the other transaction will be able to commit eventually. The only requirement is that only
one of the subtransactions should be allowed to commit. On commitment of either of them, the other should be aborted. Hence, deferring the determination of aborting an incompatible subtransaction until the commit phase offers the flexibility of allowing both subtransactions to continue their execution as long as they are compatible with their ancestor.

3.6 Consistency criterion: level-wise opacity

Several definitions of Opacity for non-nested transactions have been proposed [1, 9, 11]. The definition [9] that is close to the spirit of this thesis is given below.

**Definition 3.1 (Opacity).** A history $\hat{H}$ is opaque if it satisfies the following properties:

1. The history $\Pi(\hat{H})$ is equivalent to a sequential history (where all non-concurrent transactions are ordered as in $\hat{H}$) that is legal.

2. All transactions that abort in $\text{complete}(\hat{H})$ are invisible and their reads are consistent.

The above definition is meant for non-nested transactions. It separates the history of committed transactions from that of the aborted ones: (i) history of committed transactions is equivalent to a sequential and legal history, and (ii) aborted transactions are invisible and have consistent reads. A sequential history is legal if every transaction reads an object’s value that corresponds to the last transaction that updated the value of that object. An invisible transaction is the one that does not update
the value of any base object \[ ? \]. As such, the results of an aborted transaction are invisible to other transactions.

Now, we extend Definition 3.1 to *level-wise opacity* for nested transactions as follows:

**Definition 3.2 (Level-wise opacity).** A level-wise history \( \hat{H}_t \) is opaque if it satisfies the following properties:

1. The history \( \Pi(\hat{H}_t) \) is equivalent to a sequential history (where all non-concurrent transactions are ordered as in \( \hat{H}_t \)) that is legal.

2. For all \( t' \in \text{aborted}(H'_t) \), (i) until the time just before the abort, the execution of \( t' \) was consistent, and (ii) after the abort, \( t' \) and its descendants are invisible for \( \text{outsideTrans}(t') \).

The proofs for correctness of the protocols presented in this thesis are based on Definition 3.2.

### 3.7 Outline of the proof technique

Owing to the hierarchical structure of nested transactions, the correctness of the overall execution can be shown by considering the execution at each level of the super tree. We shall discuss the various aspects of proof system in the following sections.
Figure 3.7: Bottom to top approach constructing histories and composing steps of subtransactions (\(t_{12} \Rightarrow t_1 \{t_{121}, t_{122}\}; t_1 \Rightarrow t_1 \{t_{11}, t_1^\ast\}\))

3.7.1 Bottom up approach for constructing level-wise histories

In the nesting of transactions, the linearization point of a child transaction lies within the life span of its parent. Therefore, we show the linearizability of transactions level by level. We construct the histories in a bottom up manner using the mapping function, and in the process determine the linearizability of the transactions at each level. We illustrate the level-wise linearizability of transactions using Figure 3.7. First, we consider the history \(\overline{H}_{t_{12}}\) at node \(t_{12}\) and determine the linearizability of its children \(t_{121}\) and \(t_{122}\). Next, we consider \(\overline{H}_{t_1}\), and determine the linearization points for \(t_{11}\) and \(t_{12}\). Note that, in this case, while determining the linearization point for \(t_{12}\), we consider the combined steps of \(t_{12}\) and its children \(t_{121}\) and \(t_{122}\). Finally, we look at the global history through \(\overline{H}_{t_\psi}\) and determine the linearization points for the root level transactions \(t_1\) and \(t_2\).

This way, while working through the level-wise histories, we consider the steps of
a transaction \( t \) (that may have accessed objects from different ancestors) at different levels and show its consistency (level-wise opacity) at each level.

To show correctness, we separate the history of committed transactions from that of the aborted ones. First, we consider the history restricted to committed transactions, \( \Pi(\mathcal{H}_t) \), at each level, and show the level-wise linearizability of the committed transactions at each level.

### 3.7.2 Level-wise history of committed transactions

The level-wise history of committed transactions at node \( t \) is denoted by \( \Pi(\widehat{\mathcal{H}}_t) \). To obtain \( \Pi(\widehat{\mathcal{H}}_t) \), consider the transaction subtree rooted at node \( t \) (i.e., \( \text{transTree}(t) \)). In \( \text{transTree}(t) \), excepting the case of \( t \), we prune all the aborted subtransactions, and their descendants. The resulting tree is denoted by \( \text{prunedTree}(t) \). Note that \( t \) is included in \( \text{prunedTree}(t) \), even if \( t \) itself is an aborted or an active transaction. For

Figure 3.8: Transaction tree (thick circle: committed; thin circle: active; dotted circle: aborted)
the sake of the example, treat the active nodes shown in Figure 3.8 as committed ones. Now, \(\text{prunedTree}(t_1)\) comprises of nodes \(t_1, t_{11}, t_{12}, t_{13}, t_{121}\) and \(t_{123}\); \(\text{prunedTree}(t_{12})\) contains \(t_{12}, t_{121}\) and \(t_{123}\). Similarly, \(\text{prunedTree}(t_{122})\) contains \(t_{122}, t_{1221}\) and \(t_{1222}\).

Note that transactions \(t_{111}\) and \(t_{122}\) (including subtree rooted at \(t_{122}\)) are not considered in \(\text{prunedTree}(t_1)\) and \(\text{prunedTree}(t_{12})\) as they are aborted. Note that \(t_{122}\) is not considered in \(\text{prunedTree}(t_{12})\) and hence not considered in \(\text{prunedTree}(t_1)\) as well.

Now, \(\Pi(\hat{\mathcal{H}}_t)\) is defined to contain the steps of only the transactions in \(\text{prunedTree}(t)\) on the objects of \(t\) and \(t\)'s ancestors. In other words, \(\Pi(\hat{\mathcal{H}}_t)\) contains only the steps of \(t\) and its committed children (after applying the mapping function). For example, \(\Pi(\hat{\mathcal{H}}_{t_1})\) contains the operations of \(t_1, t_{11}, t_{12}, t_{13}, t_{121}\) and \(t_{123}\) on objects of \(t_1\) and \(t_\psi\), whereas \(\Pi(\hat{\mathcal{H}}_{t_{12}})\) contains the operations of \(t_{12}, t_{121}\) and \(t_{123}\) on objects of \(t_{12}, t_1\) and \(t_\psi\).

### 3.7.3 Reduction of a non-committed transaction

Let \(t\) be a non-committed (active or aborted) child of a transaction \(t_p\) in the transaction tree. Recall that a subtransaction \(t_c\) can update the values of the objects of its ancestors only upon its commit. Thus, transaction \(t\) could have only read the objects of its parent \(t_p\) or other higher level ancestors. With this end in view, given a \(t \in \text{aborted}(\mathcal{H}_{t_p}')\) or \(t \in \text{active}(\mathcal{H}_{t_p}')\) (where \(t\) is a child of \(t_p\)), we construct \(t^\gamma = \gamma(t)\) (\(\gamma\) stands for “reduced”) as follows. Transaction \(t^\gamma\) is obtained by taking into account only the external read steps performed by \(t\) (and the committed transactions that are part of \(t\)) on the objects of its ancestors (prior to its abort, if it is an aborted
transaction). Its local read and write steps are discarded (as they are local to \( t \)), and it is treated as committed. In case of an aborted transaction, the corresponding abort event is replaced by a commit event. This way, \( t^\gamma \) can be viewed as a committed read only transaction at its parent level.

### 3.7.4 Closure (history) for a transaction

In our model for nested transactions, if a local copy of an object is not available, then a subtransaction reads from the local space of its nearest ancestor having a copy of that object. Hence, considering \( t \) being a subtransaction in a transaction tree, the consistency of read step of \( t \), at time \( \tau \), depends upon the state of its ancestors at that time. The state of an ancestor at time \( \tau \) depends upon (a) its local (read/write) and external (read) operations, and (b) steps of its committed children, until time \( \tau \). Aborted or active transactions of \( t \) or its ancestors, at time \( \tau \), do not have any bearing on the consistency of \( t \).

Given a history \( \mathcal{H} \) for a transaction tree, the closure of history (or simply *closure*) for a transaction \( t \) is denoted as \( \mathcal{H}^{C_t} \), and obtained in following three steps:

1. \( \mathcal{H}' \) : Consider the prefix of \( \mathcal{H} \) up to the last read/write operation of \( t \).

2. \( \mathcal{H}'' \) : Discard the steps of (a) aborted transactions and (b) active transactions (other than \( t \) or its ancestors) in \( \mathcal{H}' \).

3. \( \mathcal{H}^{C_t} \) : For the active transactions (\( t \) as well as its active ancestors), append commit events in \( \mathcal{H}'' \), committing each child before its parent. These transactions are thereby treated as read only committed children at their parent level.
(Note that the local writes, if any, of $t$ and its ancestors are not mapped to corresponding commit writes at their parent level.)

Observe that a closure represents a history of committed transactions. For illustration, consider $H_6$ corresponding to the execution of transaction tree in Figure 3.8.

$$\hat{H}_6 = \langle w_{t_{11}}(t_{11}.x) r_{t_{111}}(t_{11}.x) a_{t_{111}} w_{t_{11}}(t_1.x) c_{t_{111}} w_{t_1}(t_1.y) r_{t_{121}}(t_1.x) c_{t_{121}} r_{t_{1221}}(t_{12}.x) c_{t_{1221}} w_{t_{122}}(t_{1222}.x) r_{t_{1222}}(t_{1222}.z) r_{t_122}(t_{122.x}) \rangle \star r_{t_122}(t_{12.x}) a_{t_{122}} c_{t_{122}}$$

Take the case of aborted transaction $t_{122}$. In $\hat{H}_6$, the last (read) operation of $t_{122}$ is $r_{t_{122}}(t_{122.x})$. Hence, we cut $\hat{H}_6$ right after $r_{t_{122}}(t_{122.x})$. (The cutting point is marked by $\star$.) Thus, $H_{6'}^{c_{t_{122}}}$ is as follows:

**Step (i)**: Cut the history $\hat{H}_6$ at $\star$.

$$\hat{H}_6' = \langle w_{t_{11}}(t_{11}.x) r_{t_{111}}(t_{11}.x) a_{t_{111}} w_{t_{11}}(t_1.x) c_{t_{111}} w_{t_1}(t_1.y) r_{t_{121}}(t_1.x) c_{t_{121}} r_{t_{1221}}(t_{12}.x) c_{t_{1221}} w_{t_{122}}(t_{1222}.x) r_{t_{1222}}(t_{1222}.z) r_{t_122}(t_{122.x}) \rangle$$

**Step (ii)**: Discard the steps of $t_{111}, t_{13}, t_{123}, t_{1222}$ as they are non-committed in $\hat{H}_6'$ and do not belong to $\{t_1, t_{12}, t_{122}\}$. (Removed part is underlined.)

$$\hat{H}_6'' = \langle w_{t_{11}}(t_{11}.x) r_{t_{111}}(t_{11}.x) a_{t_{111}} w_{t_{11}}(t_1.x) c_{t_{111}} w_{t_1}(t_1.y) \underline{r_{t_{121}}(t_1.x)} c_{t_{121}} r_{t_{1221}}(t_{12}.x) c_{t_{1221}} w_{t_{122}}(t_{1222}.x) r_{t_{1222}}(t_{1222}.z) r_{t_122}(t_{122.x}) \rangle$$

$$\Rightarrow \langle w_{t_{11}}(t_{11}.x) w_{t_{11}}(t_1.x) c_{t_{11}} w_{t_1}(t_1.y) r_{t_{121}}(t_1.x) c_{t_{121}} r_{t_{1221}}(t_{12}.x) c_{t_{1221}} w_{t_{122}}(t_{1222}.x) r_{t_{1222}}(t_{1222}.z) r_{t_122}(t_{122.x}) \rangle$$

**Step (iii)**: Complete non-committed transactions $t_1, t_{12}$ and $t_{122}$ in $\hat{H}_6''$. (Added part is underlined.)

$$\hat{H}_6^{c_{t_{122}}} = \langle w_{t_{11}}(t_{11}.x) w_{t_{11}}(t_1.x) c_{t_{11}} w_{t_1}(t_1.y) r_{t_{121}}(t_1.x) c_{t_{121}} r_{t_{1221}}(t_{12}.x) c_{t_{1221}}$$

47
Here, \( \hat{H}_{6}^{C_{122}} \) is history of committed transactions, comprising of the steps of \( t_1, t_{11}, t_{12}, t_{121}, t_{122} \) and \( t_{1221} \). Now, following the discussion in Section 3.4.4, the level-wise sequential history is given by \( \{ t_1 \{ t_{11}, t_{12}, t_{121}, t_{122} \} \} \).

### 3.7.5 Handling aborted and active transactions

An aborted transaction is totally discarded, i.e., its read steps as well as write steps are ignored at its parent’s and other ancestors’ levels. However, to show the correctness of aborted transactions, we consider one aborted transaction at a time in a transaction tree. Let \( t_a \) be such an aborted transaction. Then, we obtain the closure for \( t_a (\hat{H}_{C_{ta}}^{C_t}) \) by looking at the execution of transaction tree until the time just before the abort of \( t_a \).

Note that an active or aborted transaction only performs read operations on its ancestors’ objects. Updating of the parent’s objects occurs only upon its commit, provided it is an update transaction. Therefore, it is fair to treat an aborted (or active) transaction as a read only committed child at its parent’s level and show that each of its (external) read steps were consistent. By construction (step (iii)) of \( \hat{H}_{C_{ta}}^{C_t} \), \( t_a \) and its ancestors are reduced to read only committed children at their parent’s level. Further, \( \hat{H}_{C_{ta}}^{C_t} \) represents a history a committed transactions. With this end in view, the correctness of an aborted transaction follows directly from the proofs for committed transactions. We construct the level-wise history of committed transactions using \( \hat{H}_{C_{ta}}^{C_t} \), and show the correctness in the same way as done for committed transactions.

Observe that the closure can also be used for showing the correctness of any
(active) transaction at any point of its execution.

### 3.7.6 Summary of the proof technique

The set of proofs, based on Definition 3.2, can be divided into following three parts.

1. For committed transactions, $\Pi(\overline{H}_t)$ is equivalent to a legal sequential history $\overline{H}_t^\sigma$ which is obtained by ordering the transactions in $H'_t$ using the definition of linearization points based on the STM protocol (discussed for the protocols in later chapters):

   (a) $\rightarrow_{H_t^\sigma}$ is total order.

   (b) $\rightarrow_{H'_t} \subseteq \rightarrow_{H_t^\sigma}$

   (c) $t_w \xrightarrow{t_x} r_{w, r} \Rightarrow \exists t'_w$ such that $(t_w \rightarrow_{H'_t} t'_w \rightarrow_{H^o} r_{w, r}) \wedge (w_{w'}(t.x) \in H_t)$.

   (d) $t_w \xrightarrow{t_x} r_{w, r} \Rightarrow t_w \rightarrow_{H_t^\sigma} r_{w, r}$.

   Here, $r_{w, r}, t_w$ and $t'_w$ are the children of node $t$.

2. There is no cyclic conflict between transactions across different levels:

   (a) If $t_1, t_2$ are incompatible children of $t$, then $\neg(t_1 \in \Pi(H'_t) \wedge t_2 \in \Pi(H'_t))$.

   (b) A subtransaction cannot operate after it becomes incompatible with any of its ancestors.

3. For an aborted transaction $t_a$:

   (a) After the abort of $t_a$, the results of $t_a$ and its descendants are not visible to transactions in $outsideTrans(t_a)$.

   (b) Steps of transaction $\gamma(t_a)$ at its ancestral levels are consistent.
Chapter 4

SimpSTM: A simple STM protocol for (closed) nested transactions

Figure 4.1: Closed nested transactions (dark circle: committed; thin circle: active; dotted circle: aborted)
The STM protocol discussed in Section 2.6.2 was designed for non-nested transactions. In this section, we extend that protocol and design a protocol (called “SimpSTM”) for (closed) nested transactions. SimpSTM is designed under the constraint that the subtransactions of the nested transactions are executed in a sequential fashion. More precisely, this constraint is defined as follows:

**Constraint 4.1 (Sequential execution of subtransactions).** Given a transaction tree, let $t_p$ be any node with $t_1$ and $t_2$ as any two of its children. Then, we have either $E_{t_1} \prec_E t_p \prec_B t_2$ or $E_{t_2} \prec_E t_p \prec_B t_1$, denoting that a new child is invoked only after the previously created (if any) child has completed. Further, $t_p$ does not execute any step while it has an active child.

Technically, the steps of all the transactions in a transaction tree are executed by the same thread. When $t_p$ invokes a child $t_c$, the thread previously executing the steps of $t_p$, executes the steps of $t_c$ (and its descendants). Transaction $t_p$ stays idle (waits) until $t_c$ completes (commits or aborts) and the thread (control) is returned to $t_p$.

Besides the features listed in Section 2.4, the key features of SimpSTM are as follows.

- At a time, only one node in a transaction tree executes its steps. This is due to the constraint that the subtransactions in a transaction tree are executed in a sequential fashion.
- When $t$ reads from the globally shared object $t_{\psi}.x$, the read value is consistent w.r.t. the entire transaction tree.
4.1 SimpSTM

4.1.1 Pseudocode

**Protocol 4.1: SimpSTM**

1. $t_\psi.ow$, a set of ids $\in T$
2. State of globally shared object $t_\psi.x$:
   3. $val \in V$
   4. $rs$ and $fbd$, sets of ids $\in T$

**Common to root and non-root nodes:**

5. State of a transaction’s local object $t.x$:
   6. $val \in V$
7. State of transaction $t$:
   8. $parent \in T$, parent’s id (initially $\perp$)
   9. $mts$, set of ids $\in T$ (initially $\emptyset$)
10. $lws, lrs$, sets of ids $\in X$ (initially $\emptyset$)
11. Operation $\text{begin}_t(t_p)$:
   12. $t.parent \leftarrow t_p$
   13. $t.mts \leftarrow \{t\}$
14. Operation $\text{invoke}_t(t_c)$:
   15. $\text{begin}_t(t)$
16. Operation $\text{read}_t(x)$:
   17. if ($t.x$ exists) then return $t.x.val$
   18. $v = \text{search}_t(x, t, t.mts)$
   19. if ($v = \text{null}$) then $t.abort();$ end if
   20. $t.x.val \leftarrow v$
   21. $t.lrs \leftarrow t.lrs \cup \{x\}$
   22. return $v$
23. Operation $\text{write}_t(x, v)$:
   24. $t.x.val \leftarrow v$
   25. $t.lws \leftarrow t.lws \cup \{x\}$
26. Operation $\text{merge}_t(t_c)$:
   27. for each $x \in (t_c.lws \cup t_c.lrs)$ do
   28. $t.x.val \leftarrow t_c.x.val$; end for
   29. $t.lws \leftarrow t.lws \cup t_c.lws$
   30. $t.lrs \leftarrow t.lrs \cup t_c.lrs$
   31. $t.mts \leftarrow t.mts \cup t_c.mts$
32. Operation $\text{abort}_t()$:
33. return (abort)
34. Operation $\text{search}_t(x, t_d, s.mts)$:
35. if ($t.x$ exists) then return $t.x.val$; end if
36. $s \leftarrow t.mts \cup s.mts$
37. return $t_p, \text{search}_t(x, t_d, s)$

**For reading global objects (at $t_\psi$’s level):**

# Invoked by root transaction only
38. Operation $\text{search}_t(x, t_d, s.mts)$:
39. lock $t_\psi.x$
40. if ($t_\psi.x.fbd \cap (s.mts) \neq \emptyset$) then
   41. unlock $t_\psi.x$; return $null$; end if
42. $v \leftarrow t_\psi.x.val$
43. $t_\psi.x.rs \leftarrow t_\psi.x.rs \cup \{t_d\}$
44. unlock $t_\psi.x$
45. return $v$

**Specific to non-root node ($t$):**

46. Operation $\text{try.to.commit}_t()$:
47. $s_{rs} \leftarrow t.rs \cup t_lrs; s_{lws} \leftarrow t.lws \cup t_p.lws$
48. if ($s_{lws} \neq \emptyset \land s_{rs} \neq \emptyset$) then
   49. if ($t.mts \cap t_\psi.ow \neq \emptyset$) then
      50. $t.abort();$ end if end if
51. $\text{merge}_t();$
52. return (commit)

**Specific to root node ($t_r$):**

53. Operation $\text{try.to.commit}_t()$:
54. if ($t_p, lws = \emptyset$) then
   55. return (commit); end if
56. lock all the objects in $t_p.lws \cup t_p.lrs$
57. if ($t_p.mts \cap t_\psi.ow \neq \emptyset$) then
   58. release all the locks;
59. $t_p.abort();$ end if
60. $t_\psi.ow \leftarrow t_\psi.ow \cup (\bigcup x \in t_p.lws t_\psi.x.rs)$
61. for each $x \in t_p.lws$ do
62. $t_\psi.x.val \leftarrow t_p.x.val$
63. $t_\psi.x.fbd \leftarrow t_\psi.ow$
64. $t_\psi.x.rs \leftarrow \emptyset$; end for
65. release all the locks;
66. return (commit)
4.1.2 Data structures

4.1.2.1 Variable state

At the global level ($t_\psi$):

We use the same data structures ($t_\psi.x$ and $t_\psi.ow$) and the associated semantics for globally shared objects, as used in [9] (discussed in Section 2.6.2). There is a lock associated with each globally shared object $t_\psi.x$. The value of a base object $t_\psi.x$ is given by $t_\psi.x.val$. The set $t_\psi.ow$ is kept in an atomic register, and is not protected by a lock. Here, we assume an atomic register that can perform both reading and writing together atomically, as in read-modify-write. Various techniques for implementing such an atomic register have been discussed in the Section 3.4 of [9]. However, these techniques have not been discussed in this thesis to keep it within the scope. Note that only such atomic objects have been assumed throughout this thesis.

At the local level ($t$):

In the local space of a transaction $t$, a local copy of an object $t.x$ has only a value field (denoted as $t.x.val$). It does not have the pair of sets, $rs$ and $fbd$, associated with the globally shared objects.

4.1.2.2 Transaction state

Each transaction $t$ ($t \neq t_\psi$) stores the id of its parent in parent. The set $mts$ (merged transaction set) contains the id(s) of $t$ as well as $t$’s descendants that have successfully merged with $t$. Thus, $mts$ is used to take into account the fact that a transaction can be composed of one or more committed descendants, and therefore, the combined steps of the transactions in $mts$ should be considered while checking the consistency
of $t$. Further, the set $lrs$ (local read set) is used to record the ids of objects read by $t$, whereas $lws$ (local write set) is used for recording the ids of objects written by $t$.

4.1.3 Working of SimpSTM

In SimpSTM, the allocation of space for local copies of objects in the local space of a transaction is automatically done whenever required. The procedures of the protocol are discussed as follows.

$begin_t(tp)$: Each transaction begins with this procedure. Here, $tp$ is the id of the parent transaction that invoked $t$. If $t$ is a root level transaction, then $tp$ is $t_\psi$. The set $t.mts$ is initialized with \{t\}.

invoke_child$_t(t_c)$: This method is used by the parent $t$ to invoke a new child transaction $t_c$. Note that $t$ invokes a new child only when it does not already have a child that is currently active.

$read_t(x)$: When $t$ needs to read an object $x$, it checks its local space. If a local copy exists, then it reads from $t.x$. Otherwise, it tries to read from the local space of its parent, $tp$. If $tp$ also does not have a local copy of $t_p.x$, then $tp$, in turn, requests its own parent, and so on. This is done by calling $search.parent$, a recursive procedure, that searches from the parent level to higher level ancestors, until a local copy of $x$ is found. In the worst case, the search leads to reading from the globally shared copy $t_\psi.x$. If the read operation is successful, then $t$ assigns the read value to its local copy $t.x$ and adds $x$ to $t.lrs$. In case $search.parent$ returns $null$, $t$ aborts.

$search.parent_t(x,t_d,s_{mts})$: This method associated with $t$ is invoked by its children to search for a local copy of an object $x$ for the descendant $t_d$. Set $s_{mts}$ is the
union of $t'.mts$ of each intermediate ancestor $t'$ in the path from $t_d$ to $t$.

**Case I: Reading from non-global level ($t \neq t_\psi$)**

Transaction $t$ returns the value of its local copy $x$ if $t.x$ exists. Otherwise, it merges its $t.mts$ with $s.mts$ and invokes the `search_parent` method of its parent.

**Case II: Reading from global level ($t = t_\psi$)**

Here, this method is invoked by the root transaction. In this case, the globally shared object $t_\psi.x$ is locked. If none of the transactions in $s.mts$ belongs to $t_\psi.x.fbd$ then, $t_d$ is added to $t_\psi.x.rs$ and the value of $t_\psi.x$ is returned. Otherwise, `null` is returned to indicate an attempt to read an inconsistent value.

**Note:** In order to focus on the key concept and facilitating readability of the pseudocode, the method `search_parent` has been implemented everywhere using a recursive approach (memory overhead) instead of the iterative approach.

*write$_t(x,v)$*: All the writes take place in local space initially. Here, $t$ updates the value of its local object $t.x$ to $v$, and then adds $x$ to $t.lws$.

*merge$_t(t_c)$*: A child transaction $t_c$ calls this method to merge its local results with its parent $t$. The subtransaction $t_c$ may have read some object, from an ancestor, whose local copy is not available with its parent. A copy of each such object is created in its parent’s local space. Next, for each object $x$ in $t_c.lws$, the value of $t.x.val$ is set to $t_c.x.val$. Subsequently, the sets $t_c.lrs, t_c.lws, t_c.mts$ are merged with the corresponding sets of the parent $t$.

*try_to_commit$_t()$*: The nature of a commit process of a transaction $t$ depends upon its type, i.e., whether $t$ is a non-root transaction or a root transaction.

*Case I: t is a non-root level transaction (t)* Here we need to take into account if the merging of child transaction $t$ can turn a read-only or a write-only parent transaction
$t_p$ into an update transaction, and possibly make $t_p$ inconsistent if $t$ read some global object that has been modified. Therefore, we consider the joint local read sets and local write sets of $t$ and $t_p$ to check if the resultant parent is bound to be an update transaction. In that case, we first validate $t$’s steps by ensuring that none of the ids in $t.mts$ belong to $t_\psi.ow$. Upon successful validation, $t$ merges its local results with its parent, and commits. If the validation is not successful, then $t$ aborts. No merging of steps takes place in the event of an abort. In case $t_p$ is found to be a read-only or a write-only transaction, then it commits (and merges its steps with those of its parent) without having to validate its steps.

**Case II: $t$ is a root-level transaction ($t_\rho$)**

Here, if $t_\rho$ is a read only transaction (i.e. $t_\rho.lws = \emptyset$), it commits immediately. Otherwise, it obtains locks on all global objects whose ids are present in its local read and write sets. Next, it checks if any transaction belonging to $t_\rho.mts$ is present in $t_\psi.ow$. If yes, then the transaction releases all the locks and aborts. Otherwise, for each $x$ present in $t_\rho.lws$, it updates $t_\psi.x.val$ using the value of its local copy $t_\rho.x$. All the ids present in $t_\psi.x.rs$ of each $t_\psi.x$ updated by $t_\rho$ are added to $t_\psi.ow$. Next, for each $x \in t_\rho.lws$, $t_\psi.x.fbd$ is updated using $t_\psi.ow$, followed by clearing $t_\psi.x.rs$. Finally, all the locks are released and $t_\rho$ commits.

Observe that, compared to Imbs and Raynal’s Protocol 2.1, the root transaction in SimpSTM behaves differently by making use of set $mts$. If there are no nested sub-transactions, $mts$ will contain the id of only the root transaction, thereby SimpSTM will work like Protocol 2.1.

`call_abort_t()`: This method is invoked when a transaction $t$ has to abort.
4.2 Proof of correctness

We prove that the properties (1)-(3) stated in Section 3.7.6 are satisfied. First, we consider the level-wise history of committed transactions and show that the properties (1) and (2) are satisfied. Next, we consider the aborted transactions, and show that property (3) is satisfied.

Composite transaction:

Note that for a composite transaction $t_1$, its consistency depends also upon the consistency of steps of its descendants that have merged with it. In other words, the steps of all the subtransactions that have successfully merged with $t_1$ are also represented in $t_1$ now. For this purpose, we shall use the notation $\hat{t}_1$ to denote “some transaction in $t_1.mts$.” Observe that $t_1.mts$ always contains $t_1$.

Let $\beta(\hat{t}_1, t_2.s, \tau)$ be the predicate denoting “at time $\tau$ of an event/operation, $\hat{t}_1$ belongs to a set $s$ of transaction $t_2$”. In that case, at time $\tau$, we have $t_1.mts \cap t_2.s \neq \emptyset$.

4.2.1 Definition of linearization point

Extending the proof of Protocol 2.1 by Imbs & Raynal [9] for nested transaction, the linearization point $\ell_t$ of a transaction $t$ in a transaction tree is defined within the lifespan of its parent, $t_p$. Depending on whether $t$ is a root level transaction or a non-root level transaction, its linearization point is defined as follows.

Case I: $t$ is a non-root transaction (i.e., $t \neq t_p$)

1. If $t$ commits, then $\ell_t$ is the point at which $t$ merges with its parent (line 51).

2. If $t$ aborts, then $\ell_t$ is the point at which it performed its last successful read operation on its ancestor’s object (at the time of invocation line 18).
Case II: \( t \) is a root level transaction (i.e., \( t = t_\rho \))

3. If a transaction \( t \) aborts, \( \ell_t \) is placed just before \( \hat{t} \) is added to the set \( t_\psi.ow \) (line 60 of the \texttt{try.to.commit}(\( t \)) operation that entails its abort).

4. If \( t \) is a committed read only transaction, \( \ell_t \) is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 40 of the \texttt{search.parent} operation) and (2) the time just before \( \hat{t} \) (any id in \( t.mts \)) is added to \( t_\psi.ow \) (if it ever is).

5. If an update transaction \( t \) commits, \( \ell_t \) is placed just after the execution of line 60 by \( t \) (update of \( t_\psi.ow \)).

Note: For any aborted transaction, if it does not have any external read operation at its level then its linearization point lies at the time of its creation (line 11).

The above definition of linearization points is used to obtain the level-wise sequential history \( \hat{H}_t^\sigma \) by ordering \( t \)'s children according to their linearization points. Next, we shall provide the set of proofs for properties stated in Section 3.7.6.

4.2.2 Proof for committed transactions

All the histories considered in this section are the histories restricted to committed transactions only (i.e. \( \Pi(\hat{H}_t), \Pi(\hat{H}_t') \) and \( \Pi(\hat{H}_t^\sigma) \)). For committed transactions, we show the following.

(a) \( \rightarrow_{\Pi_t} \) is total order.

(b) \( \rightarrow_{\Pi_t} \subseteq \rightarrow_{\Pi_t'} \).
(c) $t_w \xrightarrow{tx} t_r \Rightarrow \exists t'_w$ such that $(t_w \rightarrow \mathcal{H}_l^t t'_w \rightarrow \mathcal{H}_r^t t_r) \land (w_{t'_w}(t.x) \in \mathcal{H}_l)$.

(d) $t_w \xrightarrow{tx} t_r \Rightarrow t_w \rightarrow \mathcal{H}_l^t t_r$.

Let us recall the notations. Here, $\mathcal{H}_l^t$ denotes a level-wise shared memory history (of events) at node $t$ in the transaction tree. Similarly, $\mathcal{H}_l'^t$ denotes a level-wise transaction history (that follows partial order), whereas $\mathcal{H}_l^\sigma_t$ denotes a level-wise sequential transaction history (that follows total order). Further, $t_w \xrightarrow{tx} t_r$ denotes that $t_w$ and $t_r$ are the two children of $t$ such that $t_r$ reads from $t.x$ ($t$’s object) the value that was written by $t_w$.

**Lemma 4.1.** $\rightarrow_{\mathcal{H}_l^t}$ is total order.

*Proof.** Trivial from the ordering of linearization points for transactions, and Constraint 4.1. □

**Lemma 4.2.** $\rightarrow_{\mathcal{H}_l'^t} \subseteq \rightarrow_{\mathcal{H}_l^t}$.

*Proof.** This lemma follows from the fact that, given any transaction $t_1 \in \mathcal{H}_l'^t$, its linearization point is placed within its lifetime. Therefore, if $t_1 \rightarrow_{\mathcal{H}_l'^t} t_2$ ($t_1$ ends before $t_2$ begins), then $t_1 \rightarrow_{\mathcal{H}_l^t} t_2$. □

**Lemma 4.3.** If $t_1 \in \Pi(\mathcal{H}_\psi)$ then $\beta(\widehat{t}_1, t_\psi.ow, \tau) \Rightarrow \ell_{t_1} <_{\mathcal{H}_\psi} \tau$.

*Proof.** Note that $\mathcal{H}_\psi$ denotes the history produced at the global level (i.e., associated with globally shared copy of objects), and $t_1$ is a root level transaction. We have to show that the linearization point for $t_1$ cannot lie after the time $\tau$ at which $\widehat{t}_1$ has been added to $t_\psi.ow$. There are two cases:

- If $t_1$ is read-only and commits, again by construction, its linearization point $\ell_{t_1}$ is
placed, at the latest, just before the time at which \( \hat{t}_1 \) (first time a transaction in 
\( t_1.mts \)) is added to \( t.ow \) (if it ever is), which proves the lemma.

- If \( t_1 \) writes and commits, its linearization point \( \ell_{t_1} \) is placed during \texttt{try.to.commit()},
while \( t_1 \) holds the locks of every object of \( t_\psi \) that it has read. If \( \hat{t}_1 \) was in \( t_\psi.ow \) before it acquired all the locks, it would not commit (due to lines 56-59). Let us notice that \( \hat{t}_1 \) can be added to \( t_\psi.ow \) only by another root-level update transaction, holding a lock on the globally shared object previously read by \( \hat{t}_1 \). As \( t_1 \) releases the locks just before committing (lines 65-66), it follows that \( \ell_{t_1} \) occurs before the time at which \( \hat{t}_1 \) is added to \( t_\psi.ow \), which again proves the lemma. \( \square \)

**Lemma 4.4.** \( t_w \xrightarrow{t.x} r_f t_r \Rightarrow \not\exists t_w' \text{ such that } (t_w \rightarrow_{\mathcal{H}^o} t_w' \rightarrow_{\mathcal{H}^o} t_r) \land (w_{t_w'}.(t.x) \in \mathcal{H}_t) \).

**Proof.** We have two cases here: (1) \( t \neq t_\psi \) and (2) \( t = t_\psi \).

**Case I: \( t \neq t_\psi \) (History of non-root level transactions)**

It means \( \hat{H}_t \) is the history at some node other than the super transaction (\( t_\psi \)). In this case, the proof follows directly from the Constraint 4.1, which states that the subtransactions are executed in a sequential manner, and only one of the transactions in a transaction tree executes at a time. This implies that \( t_w \) was the latest transaction to modify \( t.x \) before \( t_r \) was started.

**Case II: \( t = t_\psi \) (History of root-level transactions)**

By contradiction, let us assume that there are three root-level transactions, \( t_w, t_w', t_r \), and a global object \( t_\psi.x \) such that:

\[ t_w \xrightarrow{t_\psi.x} r_f t_r \]

\[ w_{t_w}(t_\psi.x, v') \in \mathcal{H}_{t_\psi} \]

\[ t_w \rightarrow_{\mathcal{H}^o} t_w' \rightarrow_{\mathcal{H}^o} t_r. \]
As both $t_w$ and $t'_w$ write $t_\psi.x$ in shared memory, they have necessarily committed (a write in shared memory occurs only at lines 61-64 during the execution of $\text{try\_to\_commit}()$, i.e., $t_w, t'_w \in \Pi(\mathcal{H}_\psi)$). Moreover, their linearization points $\ell_{t_w}$ and $\ell'_{t_w}$ occur while they hold the lock on $t_\psi.x$ (before committing), from which we have the following implications:

$t_w \to \mathcal{H}^*_\psi t'_w \Leftrightarrow \ell_{t_w} <_{\mathcal{H}_\psi} \ell'_{t_w}$,

$\ell_{t_w} <_{\mathcal{H}_\psi} \ell'_{t_w} \Rightarrow RL_{t_w}(t_\psi.x, ttc) <_{\mathcal{H}_\psi} AL_{t'_w}(t_\psi.x, ttc)$

$\Rightarrow w_{t_w}(t_\psi.x, v) <_{\mathcal{H}_\psi} w'_{t'_w}(t_\psi.x, v')$,

$t_w \xrightarrow{t_\psi.x \ r_f} t_r \Rightarrow RL_{t_w}(t_\psi.x, ttc) <_{\mathcal{H}_\psi} AL_{t_r}(t_\psi.x, \text{read}(x))$.

Please note that access to the object $t_\psi.x$ is protected by a lock and, hence, is atomic in nature. Since $t_r$ reads the value written by $t_w$, it means that $t_w \xrightarrow{t_\psi.x \ r_f} t_r \Rightarrow$ the event $r_{t_r}(t_\psi.x, v)$ follows $w_{t_w}(t_\psi.x, v)$ but precedes $w'_{t'_w}(t_\psi.x, v')$, i.e,

$(t_w \xrightarrow{t_\psi.x \ r_f} t_r) \wedge (w_{t_w}(t_\psi.x, v) <_{\mathcal{H}_\psi} w'_{t'_w}(t_\psi.x, v')) \Rightarrow w_{t_w}(t_\psi.x, v) <_{\mathcal{H}_\psi} r_{t_r}(t_\psi.x, v) <_{\mathcal{H}_\psi} w'_{t'_w}(t_\psi.x, v')$.

When a subtransaction in $\widehat{t_r}$ ($\widehat{t_r}$ accounts for the possibility that $t_\psi.x$ could be read by $t_r$ or any of its descendants that merged with $t_r$) reads an object $t_\psi.x$, it always adds its id to $t_\psi.x.rs$ before releasing the lock on $t_\psi.x$ (lines 43-44). Therefore, the predicate $\beta(\widehat{t_r}, t_\psi.x.rs, RL_{t'_w}(t_\psi.x, \text{read}(x)))$ is true ($t_\psi.x.rs$ is set to $\emptyset$ only after being added to the set $t_\psi.ow$). Using this observation, we have the following:

$r_{t_r}(t_\psi.x, v) <_{\mathcal{H}_\psi} w_{t'_w}(t_\psi.x, v') \wedge \beta(\widehat{t_r}, t_\psi.x.rs, RL_{t'_w}(t_\psi.x, \text{read}(x)))$

$\Rightarrow \beta(\widehat{t_r}, t_\psi.x.rs, AL_{t'_w}(t_\psi.x, ttc)),$

$\beta(\widehat{t_r}, t_\psi.x.rs, AL_{t'_w}(t_\psi.x, ttc)) \wedge (w_{t'_w}(t_\psi.x, v') \in \mathcal{H}_\psi) \Rightarrow \beta(\widehat{t_r}, t_\psi.ow, \ell'_{t'_w}) \Rightarrow \ell_{t_r} <_{\mathcal{H}_\psi} \ell'_{t'_w}$

$\ell'_{t_w} \Leftrightarrow t_r \to \mathcal{H}^*_\psi t'_w$. 61
which proves that, contrary to the initial assumption, \( t'_w \) cannot precede \( t_r \) in the sequential transaction history \( \hat{H}_{t'_w}^\sigma \).

\[\text{Lemma 4.5. } t_w \xrightarrow{t'_w} t_r \Rightarrow t_w \rightarrow_{\hat{H}_t} t_r.\]

**Proof.** Again we have two cases:

**Case I:** \( t \neq t'_\psi \)

Follows directly from Constraint 4.1.

**Case II:** \( t = t'_\psi \)

The proof is made up of two parts. First it is shown that \( t_w \xrightarrow{t'_w} t_r \Rightarrow \neg \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \), and then it is shown that \( \neg \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \land t_w \xrightarrow{t'_w} t_r \Rightarrow t_w \rightarrow_{\hat{H}_{t'_\psi}} t_r. \)

Proof of \( t_w \xrightarrow{t'_w} t_r \Rightarrow \neg \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \). Let us assume by contradiction that the predicate \( \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \) is true. Due to lines 56, 60, 63, 65 we have \( \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \Rightarrow \beta(\hat{t}_t, t'_\psi.x.fbd, RL_{t_w}(t'_\psi.x,ttc)) \)

If the read of \( t'_\psi.x \) from shared memory by \( t_r \) is before the write by \( t_w \), we cannot have \( t_w \xrightarrow{t'_w} t_r \). So, in the following we consider that the read of \( t'_\psi.x \) from shared memory by \( t_r \) is after its write by \( t_w \). We have then \( RL_{t_w}(t'_\psi.x,ttc) <_{\hat{H}_{t'_\psi}} AL_{\hat{t}_t}(t'_\psi.x,.read(x)) \), and consequently \( \beta(\hat{t}_t, t'_\psi.x.fbd, RL_{t_w}(t'_\psi.x,ttc)) \Rightarrow \beta(\hat{t}_t, t'_\psi.x.fbd, AL_{t_w}(t'_\psi.x,ttc)) \).

As \( \hat{t}_t \in t'_\psi.x.fbd \) when it locks \( t'_\psi.x \), it follows that read operation fails (due to lines 40-41, 37, 17-18), and consequently we cannot have \( t_w \xrightarrow{t'_w} t_r \). Summarizing the previous reasoning we have \( \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \Rightarrow \neg (t_w \xrightarrow{t'_w} t_r) \), and taking the contrapositive we finally obtain \( t_w \xrightarrow{t'_w} t_r \Rightarrow \neg \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \).

Proof of \( \neg \beta(\hat{t}_t, t'_\psi.ow, \ell_{t_w}) \land t_w \xrightarrow{t'_w} t_r \Rightarrow t_w \rightarrow_{\hat{H}_t^\sigma} t_r. \) As defined earlier, the
linearization point $\ell_{tr}$ depends on whether $t_r$ is a read only or an update transaction.

The proof considers the two possible cases.

- If $t_r$ is an update transaction that commits, its linearization point $\ell_{tr}$ (that is defined at line 60 when it updates the set $t_\psi.ow$) occurs after its invocation of $\text{try\_to\_commit()}$. Due to this observation, the fact that $t_w$ releases its locks after its linearization point, and $t_w \xrightleftharpoons{t_\psi.x}{}_{rf} t_r$, we have $\ell_{tw} < H_{t_\psi} \ell_{tr}$, i.e., $t_w \rightarrow H_{t_\psi} t_r$.

- If $t_r$ is a read only transaction that commits, its linearization point $\ell_{tr}$ is placed either before the time $t_r$ is added to $t_\psi.ow$, or at the time of the test during its last read operation (line 40). In either case, we have $w_{tw}(t_\psi.x,v) < H_{t_\psi} \ell_{tw} < H_{t_\psi} RL_{tw}(t_\psi.x,ttc) < H_{t_\psi} AL_{t_r}(t_\psi.x,\text{read}(x)) < H_{t_\psi} r_{t_r}(t_\psi.x,v) < H_{t_\psi} \ell_{tr}$, from which we have $\ell_{tw} < H_{t_\psi} \ell_{tr}$, i.e., $t_w \rightarrow H_{t_\psi} t_r$. Hence, in all cases, we have $t_w \xrightleftharpoons{t_\psi.x}{}_{rf} t_r \Rightarrow t_w \rightarrow H_{t_\psi} t_r$. \hfill $\square$

**Proof of non-existence of cyclic relation between transactions across levels**

Cyclic conflict between transactions across levels comes into the picture when the read set of a subtransaction contains incompatible read operations. In SimpSTM, at any time only one subtransaction in a transaction tree actively executes its steps while its ancestors remain idle. The values of local objects of its ancestors, other than $t_\psi$, remain consistent as they do not change while $t$ is active. Therefore, the only way cyclic conflict between transactions at different levels can occur is when a subtransaction $t$ reads a value from a global object $t_\psi.x$ that is inconsistent for $t$ or any of its intermediate ancestors. Hence, we prove the following.

**Lemma 4.6.** When a subtransaction $t$ reads a value from an object $t_\psi.x$, it is con-
sistent w.r.t. $t$ as well as each of its intermediate ancestors in the transaction tree.

**Proof.** The state of a transaction consists of its steps as well as the steps of its committed children. In SimpSTM, the ids of transactions that form the current state of a transaction $t$ is given by the set $t.mts$. Before a subtransaction $t$ tries to read from a global object $t\psi.x$, it recursively searches the local space of each of its intermediate ancestors (lines 17, 35). In the process, the content of $mts$ of $t$ as well as each of its ancestors is added to set $s_{mts}$ (lines 18, 36). Thus, by the time we come to read from $t\psi.x$, $s_{mts}$ contains the ids of all the non-aborted transactions currently in the transaction tree. Now, before reading from $t\psi.x$, we check that the value of $t\psi.x$ is consistent with each of the transactions whose id is in $s_{mts}$ by ensuring $t\psi.x.fbd \cap s_{mts} = \emptyset$ (line 40). The value is read only if it is consistent. Otherwise, $t$ aborts (lines 40-41, 37, 18-19).

\[ \square \]

**Theorem 4.1.** Every level-wise history of committed transactions, $\Pi(\hat{H}_t^\sigma)$, produced by SimpSTM satisfies the level-wise opacity consistency criterion.

**Proof.** The proof follows from definition of linearization points, and Lemmas 4.1 through 4.6. \[ \square \]

### 4.2.3 Proof for aborted transactions

As stated earlier (Constraint 4.1), the subtransactions are executed in a sequential manner. That means in a transaction tree, only one subtransaction commits or aborts at a time. To prove the correctness (opacity criterion) for an aborted subtransaction (say $t_a$), we prove the following property (2) of Definition 3.2: (i) Until the time just
before the abort of $t_a$, the execution of $t_a$ was consistent, and (ii) after the abort, $t_a$ and its descendants are invisible for $\text{outsideTrans}(t_a)$.

During the execution of the transaction tree, several nested subtransactions might abort. To prove the correctness for aborted transactions, it is important that we consider the steps of one aborted transaction at a time. This criterion is inherently captured in the construction of a closure for a transaction. To make the idea clear, consider the following history (shown in Figure 4.2):

$$\hat{H}_1 = \langle r_{t_1}(t_{\psi}.x), r_{t_{11}}(t_{11}.x), r_{t_{11}}(t_{\psi}.y), w_{t_{11}}(t_{11}.y), w_{t_2}(t_{\psi}.y), c_{t_2}, a_{t_{11}}, r_{t_{12}}(t_{\psi}.z), c_{t_{12}}, r_{t_{13}}(t_{\psi}.y), w_{t_{13}}(t_{13}.x), w_{t_3}(t_{\psi}.y), c_{t_3}, a_{t_{13}}, c_{t_1} \rangle.$$

In history $\hat{H}_1$, $t_1, t_{11}, t_{12}$ and $t_{13}$ form parts of the same transaction tree, with $t_1$ being the root transaction, and $t_{11}, t_{12}, t_{13}$ being its children. Transactions $t_2$ and $t_3$ are two other root-level transactions. Here, subtransactions $t_{11}$ and $t_{13}$ abort in order. Let us observe how the linearization point for $t_1$ (the transaction tree associated with $t_1$) shifts when we consider the steps of its children (in case of aborted children, we consider their reduced forms). Let us determine the linearization point for $t_1$ at the times of the aborts of $t_{11}$ and $t_{13}$ respectively.

**At the time of abort of $t_{11}$**: Since $t_2$ modified $t_{\psi}.y$ after it was read by $t_{11}$ (descendant of $t_1$), $t_1$’s linearization point, denoted by $\ell_{t_1}^1$, occurs before that of $t_2$. Thus, the serial order at root level is $t_1, t_2$.

**At the time of abort of $t_{13}$**: Here $t_{13}$ is able to read the fresh value of $t_{\psi}.y$ written by $t_2$ (as $t_{11}$ is not considered a part of $t_1$ at this point). Further, $t_{\psi}.y$ is modified by $t_3$ before the abort of $t_{13}$. Hence, linearization point for $t_1$ at this juncture lies at $\ell_{t_{13}}$. Consequently, the serial order of root-level transactions is $t_2, t_1, t_3$. Notice how the serial order of root transactions $t_1$ and $t_2$ is different in this case, owing to the shift.
Figure 4.2: Linearization points of transactions
of linearization point of $t_1$.

Observation: Notice that if we consider the steps of both aborted subtransactions, $t_{11}$ and $t_{12}$, then it is not possible to determine the linearization point of $t_1$. The operations $r_{t_{11}}(t_\psi.y) \; w_{t_2}(t_\psi.y) \; c_{t_2}$ dictates the serial order $t_1, t_2$, whereas $w_{t_2}(t_\psi.y) \; c_{t_2} \; a_{t_{11}} \; r_{t_{13}}(t_\psi.y)$ demands the order $t_2, t_1$. For this reason, we consider one aborted subtransaction at a time, and the steps of the previously aborted subtransactions are ignored (also implied by part(ii) of property(2) of Definition 3.2). Hence, the part of the $\widehat{H}_1$ considered at the time of abort of $t_{11}$ is $\widehat{H}_1^{c_{t_{11}}} = \langle r_{t_1}(t_\psi.x) \; r_{t_{11}}(t_1.x) \; r_{t_{11}}(t_\psi.y) \; w_{t_{11}}(t_{11}.y) \rangle \; w_{t_2}(t_\psi.y) \; c_{t_2} \; a_{t_{11}} \rangle$, whereas the history considered at the time of abort of $t_{13}$ is $\widehat{H}_1^{c_{t_{13}}} = \langle r_{t_1}(t_\psi.x) \; w_{t_2}(t_\psi.y) c_{t_2} \; r_{t_{12}}(t_\psi.z) \; c_{t_{12}} \; r_{t_{13}}(t_\psi.y) \; w_{t_{13}}(t_{13}.x) \; w_{t_3}(t_\psi.y) \; c_{t_3} \; a_{t_{13}} \rangle$. Note that the steps of the previously aborted subtransaction, $t_{11}$, are ignored in the latter case.

Treatment of aborted transaction:

To prove the part(i) of property (2) of Definition 3.2, we consider the state of the transaction tree, at a time ($\tau_{t_a}$) just before the abort of a subtransaction (say) $t_a$, and obtain the closure for $t_a$.

Now, we prove part(i) of the property(2) of Definition 3.2.

Lemma 4.7. Until the time just before the abort of $t_a$, the execution of $t_a$ was consistent, i.e., $\gamma(t_a)$ is linearizable at its ancestors’ levels.

Proof. The closure for $t_a$, $\widehat{H}_a^{c_{t_a}}$ represents a history of committed transactions, including all the steps of $t_a$. Hence, level wise histories of committed transactions can be obtained using $\widehat{H}_a^{c_{t_a}}$ and the correctness can be proved in the same way as done for the history of committed transactions in Section 4.2.2. \hfill $\square$
Next, we prove part (ii) of property (2) of Definition 3.2.

Lemma 4.8. If $t' \in \text{aborted}(\mathcal{H}_t)$, then results (read and write sets) of $t'$ and its descendants are not made available to its ancestors (including $t$).

Proof. By construction of SimpSTM, when a subtransaction commits, it writes in the memory location of its parent only (lines 51, 26-31 for closed nested transactions, and 61-64 in case of a root level transaction whose parent is the fictitious transaction $t_\psi$). At this point, we can observe that, at the time of committing, transaction $t$ also contains the results of its descendants (if any) that have merged with it. Before writing into the memory location of its parent, transaction $t$ undergoes consistency checking (lines 48-49, 57-58). If the consistency checking fails at this point, then the transaction aborts (lines 50, 59) without modifying the objects of its parent. This means the results of $t'$ are not propagated to $t$. This in turn implies that, even if $t$ commits later, results of $t'$ will not be carried upward to $t$'s parent or other ancestors for that matter. Thus, if $t'$ aborts, its results cannot be available to its ancestors. \qed

Lemma 4.9. If $t' \in \text{aborted}(\mathcal{H}_t)$, then $t'$ and its descendants (if any) are invisible to transactions $\in \text{outsideTrans}(t')$.

Proof. Again, by construction, if a local copy of an object is not available with a transaction $t$, then it tries to read from the local space of its ancestors only, through the recursive call to search_parent method (lines 18, 34-37, 38-45), not from any other transaction in the super tree. Now, given the hierarchical composition of transactions, a transaction $t''$ in $\text{outsideTrans}(t')$ can witness the results of $t'$ only if they are made available to the (least) common ancestors of $t'$ and $t''$ in the super tree.
As the results of an aborted transaction are not made available to its ancestors (by Lemma 4.8), none of the transactions in \textit{outsideTrans}(t') can see the results of \(t'\). Hence, \(t'\) is invisible to transactions in \textit{outsideTrans}(t').

\textbf{Theorem 4.2.} If \(t' \in \text{aborted}(H_t)\), then \(t'\) satisfies level-wise opacity.

\textit{Proof.} The proof follows directly from Lemmas 4.7 through 4.10.

\textbf{Theorem 4.3.} Each level wise history \(\hat{H}_t\) produced by SimpSTM satisfies level-wise opacity.

\textit{Proof.} The proof follows from the conjunction of Theorem 4.1 and Theorem 4.2.
Chapter 5

ParSTM

5.1 The main idea

In Chapter 4, we saw that SimpSTM was designed with the constraint that the subtransactions in the transaction tree are executed sequentially, i.e., one after the other. As such, under SimpSTM there is no concurrent execution of the nodes in the transaction tree. So, the idea here is to relax that constraint (Constraint 4.1) and introduce some degree of concurrency among the nodes. To this end, we employ an optimistic approach at the global level, and a pessimistic approach (2PL for nested transactions) at the intra-transaction (transaction tree) level.

5.1.1 Optimistic behaviour at the global level ($t_\psi$):

Here, the locking of global objects (associated with $t_\psi$) is done in the same optimistic manner as done under SimpSTM. In other words, when a transaction $t$ accesses a global object $t_\psi.x$ for a read or write operation, it releases the lock on $t_\psi.x$ immediately
Figure 5.1: Optimistic mode of concurrency at global level (single circle) and pessimistic mode at nested level (double circle)

after the execution of its operation, i.e., $t$ does not retain the lock on $t_\psi.x$ throughout its lifetime. This allows other transactions to access $t_\psi.x$ in the meantime.

5.1.2 Pessimistic behaviour at the nested level ($p$-node, $t_\pi$):

With each node $t$ in the transaction tree, there is a local copy $t.x$ of each object $x$ that is accessed by $t$ or its descendants. Each of these objects is protected by a lock. To indicate the point that each node in the transaction tree exhibits pessimistic behaviour, we call it a $p$-node and denote it by $t_\pi$ in general (Figure 5.1). A $p$-node denotes a transaction whose objects are accessed through a pessimistic approach of concurrency control. More precisely, when an object $t_p.x$ of a node, $t_p$, is locked by its child transaction $t_c$, then $t_c$ retains the ownership of that lock until $t_c$ completes its execution (commit or abort). That means, until $t_c$ completes, neither $t_p$ nor any other child of $t_p$ can access (obtain the lock on) $t_p.x$. They have to wait to access $t_p.x$. 

71
Figure 5.2: Partial concurrency

Note: When \( t_p \) itself locks an object \( t_p.x \) for its local operation, it unlocks \( t_p.x \) after the completion of its operation, thereby making it available for subsequent access by its descendants.

5.1.2.1 Partial concurrency at the nested level

Owing to the pessimistic approach (similar to 2PL for nested transactions) at the nested level, partial concurrency is achieved at the intra-transactional level. The different cases can be studied as follows.

(i) Same copy of object is accessed sequentially: For example, consider a node \( t_{11} \) in a transaction tree given in Figure 5.2. Transactions \( t_{111} \) and \( t_{112} \) are the two children of \( t_{11} \). Now, \( t_{111} \) and \( t_{112} \) can access different objects of \( t_{11} \) concurrently. However, if the two children want to access the same object of \( t_{11} \), then one of them has to wait. In another words, if \( t_{111} \) wants to access \( t_{11}.x \) and \( t_{112} \) wants to access
\( t_{11}.y \), then they can access the respective objects of \( t_{11} \) concurrently. If both \( t_{111} \) and \( t_{112} \) want to access the same object \( t_{11}.x \), then the lock on \( t_{11}.x \) is granted to only one of them, say \( t_{111} \), at a time, and \( t_{112} \) has to wait until \( t_{111} \) terminates and releases the lock on \( t_{11}.x \).

(ii) \textit{Different copies of the same object can be accessed concurrently:} Now consider the case at nodes \( t_{11} \) and \( t_{122} \). Local copies of \( x \) are created at these nodes due to their respective local writes, \( w_{t_{11}}(t_{11}.x) \) and \( w_{t_{122}}(t_{122}.x) \). Now, these two different local copies, \( t_{11}.x \) and \( t_{122}.x \), can be accessed concurrently. For example, accessing of \( t_{11}.x \) by \( t_{111} \) and \( t_{122}.x \) by \( t_{1222} \) can take place concurrently.

5.1.2.2 Handling deadlock situations

A deadlock situation may occur between children (descendants) when they try to lock the same set of objects of the parent (ancestor). A child node may be waiting for a lock on the parent’s object. The reverse is not true as the parent does not access its child’s object. Further, the parent performs one read or write operation on one local object at a time, and releases the lock on its local object as soon as the operation completes. Therefore, there is no deadlock situation between the parent and the child.

To handle the possibility of a deadlock between the children, each node in the transaction tree maintains a wait-for graph, whose access is controlled by a lock.
Please refer to Protocol 5.1 for the following discussion.

(a) Construction of wait-for graph $wfg_t$ at a node $t$:

The wait-for-graph $wfg_t$ consists of nodes and directed edges between them. The nodes in the graph represent $t$ and its children. An entry in $wfg_t$ is denoted by the tuple $\langle x, t_1, t_2 \rangle$. Initially, for each object $x$, there is an entry (owner node) $\langle x, t_{init}, t_{init} \rangle$, $t_{init}$ is a fictitious transaction.

**Meaning of a tuple $\langle x, t_1, t_2 \rangle$:**

- $t_1 = t_2 = t'$: Denotes a node $t'$ that is the current owner of lock on $t.x$. The owner transaction in such a tuple is denoted by $t_{own(x)}$.
- $t_1 \neq t_2$: Denotes an edge from node $t_1$ to node $t_2$, indicating $t_1$ is waiting for a lock on $t.x$ which is currently held by $t_2$.

We construct the graph by considering only the edges, i.e., those entries $\langle x, t_1, t_2 \rangle$ in which $t_1 \neq t_2$. Further, to ensure the correctness (in face of concurrent execution), the access to $wfg_t$ is protected by a lock.

(b) Managing $wfg_t$: 

<table>
<thead>
<tr>
<th>Protocol 5.1: management of wait-for graph, $wfg_t$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{own(x)} = t', \exists t': \langle x, t', t' \rangle \in t.wfg$</td>
</tr>
<tr>
<td>i Operation request_lock$_t$(x, $t_c$) :</td>
</tr>
<tr>
<td>ii lock $t.wfg$;</td>
</tr>
<tr>
<td>iii add edge $\langle x, t_c, t_{own(x)} \rangle$ to $t.wfg$;</td>
</tr>
<tr>
<td>iv if $(t.wfg$ contains a cycle) then</td>
</tr>
<tr>
<td>v remove $\langle x, t_c, t_{own(x)} \rangle$ from $t.wfg$;</td>
</tr>
<tr>
<td>vi unlock $t.wfg$; return false; end if</td>
</tr>
<tr>
<td>vii unlock $t.wfg$; return true;</td>
</tr>
</tbody>
</table>

| x Operation align_request$_t$(x, $t_c$) : |
| ix lock $t.wfg$; $t_{old} \leftarrow t_{own(x)}$; |
| x $\langle x, t_{old}, t_{old} \rangle \leftarrow \langle x, t_c, t_c \rangle$; |
| xi replace all $\langle x, t', t_{old} \rangle$ with $\langle x, t', t_c \rangle$; |
| xii if $(t.wfg$ contains a cycle) then |
| xiii Undo the effects of lines x-xi; |
| xiv remove edge $\langle x, t_c, t_{old} \rangle$; |
| xv unlock $t.wfg$; return false; end if |
| xvi unlock $t.wfg$; return true; |

| xvii Operation secure_lock$_t$(x, $t_c$) : |
| xviii if $(\neg$request_lock$_t$(x, $t_c$)) then return false; end if |
| xix lock t.x; |
| xx if $(\neg$align_request$_t$(x, $t_c$)) then |
| xxii unlock t.x; return false; end if |
| xxiii return true; |
Recall that a local object $x$ associated with node $t$ is denoted as $t.x$. To successfully access $t.x$, a node $t_c$ secures a lock on $t.x$ by following three steps: (i) request a lock on $t.x$, (ii) lock $t.x$, and (iii) align the requests of other transactions for locking $t.x$.

(i) $\text{request\_lock}_t(x, t_c)$: When a node $t_c$ requests a lock on $t.x$, $wfg_t$ is locked first. Let $t_{\text{own}(x)}$ be the current owner of $t.x$. We add an edge $\langle x, t_c, t_{\text{own}(x)} \rangle$. In this case, it simply indicates a node $t_c$ is waiting for a lock on $t.x$, and other subtransactions might be trying to lock $t.x$ at the same time. Now, if addition of edge $\langle x, t_c, t_{\text{own}(x)} \rangle$ leads to a cycle in $wfg_t$, then we remove the edge $\langle x, t_c, t_{\text{own}(x)} \rangle$ from $wfg_t$ and return $false$ to indicate failure (deadlock situation). Otherwise, $true$ is returned to indicate that the request does not cause deadlock and $t_c$ waits until it obtains a lock on $t.x$.

(ii) $\text{lock\_t\_x}$: A number of subtransactions might be waiting for a lock on $t.x$. When the lock on $t.x$ is released by the previous owner, $t_{old}$, one of the waiting transactions is selected in a non-deterministic way as the new owner of the lock.

(iii) $\text{align\_request}_t(x, t_c)$: When $t_c$ gets the lock, it first needs to rearrange the edges in $wfg_t$ to reflect ownership of lock on $t.x$, and let other transactions, waiting for a lock on $t.x$, wait for $t_c$ to release the lock. To this end, it replaces all the edges $\langle x, t', t_{old} \rangle$ with $\langle x, t', t_c \rangle$. If this change leads to a cycle in $wfg_t$, then it is rolled back along with removal of edge $\langle x, t_c, t_{old} \rangle$ from $wfg_t$, and $false$ is returned to indicate failure. If the alignment is successful, then $t_c$ sets itself as the new owner, $t_{\text{own}(x)}$, and $true$ is returned.

(iv) $\text{secure\_lock}_t(x, t_c)$: If $\text{request\_lock}$, $\text{lock\_t\_x}$ and $\text{align\_request}$ are completed successfully, it returns $true$. Otherwise, $false$ is returned.

While trying to secure a lock, a transaction $t_c$ does not need to keep a lock on $wfg_t$ all the time. Observe that the lock on $wfg_t$ is released at the end of $\text{request\_lock}$ as
well as \textit{align}\_\textit{request}. This allows other transactions also to lock \textit{wfg}_t and complete their \textit{request\_align} steps, while \(t_c\) waits for the opportunity to lock its object.

\textbf{(c) Correctness of \textit{wfg}_t:}

To show the correctness of wait-for graph constructed this way and its management, we show the following:

\begin{itemize}
  \item[i] \textit{Acyclicity of wait-for graph is maintained}.
  \item[ii] \textit{Access to wait-for graph through \texttt{secure\_lock} method is non-blocking}.
  \item[iii] \textit{Access to wait-for graph by the owner for its local operation does not bring it in cyclic conflict with its children}.
\end{itemize}

\textbf{Lemma i. Acyclicity of wait-for graph is maintained.}

\textit{Proof.} As discussed earlier, a successful acquisition of lock on parent’s object \(t.x\) requires the following: (i) checking that the request for the lock does not cause a cycle in \textit{wfg}_t, (ii) acquiring the lock on \(t.x\), and (iii) resetting the wait-for relations corresponding to the others' requests for a lock on \(t.x\), ensuring that the resetting does not lead to a cycle in \textit{wfg}_t.

The correctness follows from the argument that acyclic property of the \textit{wfg}_t is maintained at all the time. Note that access to \textit{wfg}_t is protected by a lock (lines \(ii, ix\)). This means only one transaction updates \textit{wfg}_t at a time.

Initially, when none of the objects have been locked, there are only nodes \(\langle x, t_{\text{init}}, t_{\text{init}} \rangle\), and no edges. This means there is no cycle in \textit{wfg}_t initially. Since the request for accessing \(t.x\) is made through method \texttt{secure\_lock}, the first edge in \textit{wfg}_t is added through method \textit{request\_lock}. Next we show that before and after the completion of methods \textit{request\_lock} and \textit{align\_request}, \textit{wfg}_t is acyclic.
During \textit{request\_lock}, \(wfg_t\) is locked (line \textit{ii}) and a new edge is added (line \textit{iii}) to \(wfg_t\) only if it does not cause a cycle (lines \textit{iv} – \textit{vi}). This means, if \(wfg_t\) is acyclic at the starting of \textit{request\_lock}, then it is acyclic at the end of the operation as well.

Next, during \textit{align\_request}, the edges in \(wfg_t\) are rearranged (lines x-xi) only if the new arrangement does not lead to a cycle in \(wfg_t\). Also, note that the corresponding edge \(\langle x, t, t' \rangle\) is reduced to a node \(\langle x, t, t \rangle\). Otherwise, if a cycle is detected, the previous state of \(wfg_t\) is restored (lines \textit{xii} – \textit{xv}). This means that the acyclic property of \(wfg_t\) is preserved after \textit{align\_request} operation.

Now, we have following conclusions: \((m, n > 0 \text{ and denote the number of executions of a method})\)

\[(\text{Initial state}) \rightarrow wfg_t \text{ is acyclic.}\]

\[\Rightarrow (\text{initial state}) \rightarrow (\textit{request\_lock})^m \rightarrow wfg_t \text{ is acyclic.}\]

\[\Rightarrow (\text{initial state}) \rightarrow (\textit{request\_lock})^m(\textit{align\_request})^n \rightarrow wfg_t \text{ is acyclic.}\]

Thus, the acyclic property of \(wfg_t\) is always preserved. In other words, \(wfg_t\) ensures there is no deadlock situation (cycle).

\textbf{Lemma ii.} \textit{Access to the wait-for graph through the secure\_lock method is non-blocking.}

\textit{Proof.} The lemma implies that when a subtransaction \(t_1\) accesses \textit{secure\_lock} to obtain a lock on object \(x\), it does not prevent other subtransactions from accessing \textit{secure\_lock} until \(t_1\)'s invocation of \textit{secure\_lock} has completed.

The lock on \(wfg_t\) is held only for the duration of \textit{request\_lock} and \textit{align\_request} (lines \textit{ii}-\textit{vi} and \textit{ix-xvi}). After successfully calling \textit{request\_lock}, while \(t_1\) waits for actually getting a lock on \(x\) (line \textit{xix}), other subtransactions are free to lock \(wfg_t\).
through request_lock or align_request. This allows other transactions to validate their requests for locking objects. Similarly, upon completion of align_request, $t_1$ releases the lock on $wfg_t$ immediately.

**Lemma iii.** Access to wait-for graph by the owner for its local operation does not bring it in cyclic conflict with its children.

**Proof.** When a transaction $t$ tries to lock its object $t.x$ for its local read/write operation, the edge $\langle x, t, t' \rangle$, where $t'$ is the current owner of the lock, is added in $wfg_t$. At that time, there is no edge $\langle y, t'', t \rangle$ for any $y$ and $t''$. Since $t$ performs only one local operation at a time, it does not have any other lock. When $t$ gets the lock on $t.x$, in the align_request operation, some edges $\langle x, t'', t \rangle$ may be added, but $\langle x, t, t' \rangle$ is deleted at that time. Therefore, there will be no edge out-directed from $t$ to any other node. Thus, $t$ can never be in a directed cycle.

### 5.2 Implementing 2PL for nested transactions

Recall that each local copy of an object with a transaction is protected by a simple mutex based lock. Then, the two phase locking for nested transactions is achieved as follows:

1. When a subtransaction $t$ reads an object $t'.x$ from an ancestor $t'$, it (recursively) locks the object $t''.x$ at each ancestor $t''$ in the path from $t$ to $t'$ (including $t'$). Transaction $t$ does not release the lock on them upon termination of its (successful) read operation.
(a) When $t_{1111}$ commits, lock on $x$ is released only up to the parent $t_{111}$

(b) When $t_{1111}$ aborts, lock on $x$ is released up to the original owner $t_1$

Figure 5.3: Implementing 2PL for nested transactions: (1) cascaded locking of $x$ at all the ancestors up to $t_1$ during $r_{t_{1111}}(t_1,x)$ (shown by dotted arrows), and (2) unlocking $t_{111}.x$ only upon completion of $t_{1111}$ (shown by solid arrow).
2. Transaction $t$ releases locks to its ancestor(s) only upon its completion.

Case i : If $t$ commits, it releases lock on $x$ only at its parent level.

Case ii : If $t$ aborts, lock on $x$ is released all the way up to the ancestor $t'$ from which it was originally read.

For example, consider Figure 5.3. Transaction $t_{1111}$ recursively searches for a local copy of $x$ with its ancestors, before finally reading from $t_1.x$, and, in the process, locks object $x$ at each of its ancestors up to $t_1$. Upon successful read operation, $t_{1111}$ does not release these locks; rather it retains them through its lifetime. That means no other transaction can obtain the lock on $x$ at any of its ancestors up to $t_1$, i.e., $t_{1111}, t_{11}$ and $t_1$. In other words the value of $x$ at $t_{1111}$'s ancestors up to $t_1$ cannot be changed while $t_{1111}$ is active.

Moreover, when $t_{1111}$ commits, it unlocks $x$ only at its parent $t_{111}$, not at other ancestors. Therefore, at this point, only $t_{111}.x$ becomes available to $t_{111}$ and its other descendants. However, if $t_{1111}$ aborts, it unlocks $x$ at all the ancestors up to $t_1$. Similarly, if $t_{111}$ commits later on, only $t_{11}.x$ is unlocked. Otherwise, if $t_{111}$ aborts, locks on $t_{11}.x$ as well as $t_1.x$ are released. In other words, if the subtransactions in the path from $t_{1111}$ to $t_1$ commit one by one, they release the lock on $x$ only at their respective parent level. However, if any of the subtransactions in that path abort, the lock on $x$ is released up to $t_1$. 

5.3 Issue of incompatible read operations/transactions

We resume the discussion of incompatible transactions, referred to in Section 3.5.1. For sake of simplicity, consider a simple two-level transaction tree with $t_1$ as root transaction. Let $t_{11}$ and $t_{12}$ be the two children of $t_1$, and $t_2$ be another root transaction. Now, referring to Figure 5.4, consider the following history.

\[
\hat{H}_1 = \langle r_{t_{11}}(t_\psi.x) \ w_{t_2}(t_2.x) \ w_{t_2}(t_2.y) \ w_{t_2}(t_\psi.x) \ w_{t_2}(t_\psi.y) \ c_{t_2} \ r_{t_{12}}(t_\psi.y) \ (c_{11} \ c_{12})? \rangle
\]

Recall that the lock on global objects $(t_\psi.x, t_\psi.y)$ are released soon after the read or write operations on them. Here, local copies of objects $x$ and $y$ are not available with transactions initially. As such, $t_{11}$ (locks and) reads from $t_\psi.x$, adds its id to $t_\psi.x.rs$ before releasing the lock on $t_\psi.x$. Next, another root transaction $t_2$ modifies $t_\psi.x$ and $t_\psi.y$, consequently adding $t_{11}$ to $t_\psi.x.fbd, t_\psi.y.fbd$ and $t_\psi.ow$ respectively. Now, $t_{12}$ wants to read $t_\psi.y$. When $t_{12}$ locks $t_\psi.y$, it can notice $t_{11}$ in $t_\psi.y.fbd$. However, as $t_{11}$ has not yet committed and become part of $t_1$ ($t_{12}$’s ancestor), it is legal for $t_{12}$ to read $t_\psi.y$. 

Figure 5.4: Incompatible transactions in ParSTM
Now, let us look at how the steps of the children affect the linearizability of the root transaction $t_1$. When a closed nested transaction commits, its steps become part of the parent transaction. Following this point, consider the steps of $t_{11}$ first to determine the linearization point of $t_1$. Transaction $t_{11}$ read $t_\psi.x$ before it was modified by $t_2$. This implies that linearization point of $t_1$ precedes that of $t_2$, if $t_{11}$ commits. Next, consider the steps of $t_{12}$. The read step $r_{t_{12}}(t_\psi.y)$ returns the value written by $t_2$, implying the linearization point of $t_1$ should be placed after that of $t_2$, if $t_{12}$ commits. Hence, a contradiction (if both $t_{11}$ and $t_{12}$ commit). Here, the conflicting write operations of $t_2$ are sandwiched between the respective read operations of $t_{11}$ and $t_{12}$ on the global objects. Therefore, $t_1$ cannot be linearized at the global level ($\hat{H}_\psi$), if both $t_{11}$ and $t_{12}$ are allowed to commit. In other words, the two read operations, $r_{t_{11}}(t_\psi.x)$ and $r_{t_{12}}(t_\psi.y)$, are incompatible for $t_1$. Thus, incompatible operations and transactions are defined as follows.

**Definition 5.1 (Incompatible operations and transactions).** Let $t_1, t_2$ be any two transactions in the super tree, and $t$ ($t \neq t_\psi$) be the least common ancestor of $t_1$ and $t_2$. Let $r_{t_{11}}(t_\psi.x)$ and $r_{t_{12}}(t_\psi.y)$ be successful read operations of $t_1$ and $t_2$ respectively on global objects $t_\psi.x$ and $t_\psi.y$. Then, these two read operations are incompatible if $t_1 \in t_\psi.y.fbd$ at the time of $r_{t_{12}}(t_\psi.y)$ or vice-versa. The two transactions, $t_1$ and $t_2$, are termed as incompatible transactions.

Each transaction maintains an atomic object called cm (consistency management) that has two sets its (incompatible transaction set) and mts (merged transaction set). The set its is used to keep track of incompatible transactions, whereas mts, initially containing its own id, is used to keep track of descendants that have merged with it. It
works as follows. Let us denote the root level transaction as $t_\rho$. Suppose a transaction $t$ in $\text{transTree}(t_\rho)$ reads from the global object $t_\psi.x$. To keep a record of all the transactions in $\text{transTree}(t_\rho)$ that have read some global object, each transaction maintains a special set called $\text{vts}$ (visited transaction set). Note that each transaction, before reading any global object $t_\psi.x$, adds its id to $t_\rho.\text{vts}$. Thus, the set $t_\rho.\text{vts}$ contains the ids of all the transactions in $\text{transTree}(t_\rho)$ that have read some global object. Now, upon reading $t_\psi.x$, the ids of transactions that are present in both $t_\rho.\text{vts}$ and $t_\psi.x.\text{fbd}$ are added to $t.\text{cm.its}$.

Later, when $t$ tries to merge with its parent $t_\rho$, it ensures that $t$ is not incompatible with $t_\rho$, by ensuring that $t_\rho.\text{cm.mts} \cap t.\text{cm.its} = \emptyset$ and $t_\rho.\text{cm.its} \cap t.\text{cm.mts} = \emptyset$.

Now, let us examine history $\hat{H}_1$ again under this scheme. Here, $t_1$ plays the role of the root transaction $t_\rho$. Initially, we have $t_{11} \in t_{11}.\text{cm.mts}, t_{12} \in t_{12}.\text{cm.mts}, t_1 \in t_1.\text{cm.mts}, t_{11}.\text{cm.its} = \emptyset$ and $t_{12}.\text{cm.its} = \emptyset$. In this history, at the time $t_{12}$ accesses $t_\psi.y$, clearly $t_{11} \in t_\psi.y.\text{fbd}$ and $t_{11}.\text{vts}$. Therefore, $t_{12}$ adds $t_{11}$ to $t_{12}.\text{cm.its}$.

Note: To avoid concurrent merging of incompatible subtransactions, we impose the constraint that at each level only one child of a (parent) transaction can merge with it at a time. This is achieved by use of a $\text{mrg}$ lock for merging.

If $t_{11}$ (commits and) merges with $t_1$ first, then $t_{11}$ is added to $t_1.\text{cm.mts}$. Later, when $t_{12}$ tries to merge, it would fail (abort) as $t_1.\text{cm.mts} \cap t_{12}.\text{cm.its} = \{t_{11}\} \neq \emptyset$. Similarly, if $t_{12}$ merges first, then $t_{12}$ is added to $t_1.\text{cm.mts}$ and $t_{11}$ to $t_1.\text{cm.its}$. As a result, $t_{11}$ will not be able to commit as $t_1.\text{cm.its} \cap t_{11}.\text{cm.mts} = \{t_{11}\} \neq \emptyset$. Thus, we can ensure that two incompatible children of a transaction cannot both commit.

If we consider the previous case by replacing $t_{12}$ with $t_1$ (and so replacing $r_{t_{12}}(t_\psi.y)$ with $r_{t_1}(t_\psi.y)$) in $\hat{H}_1$, the solution still works by not allowing $t_{11}$ to commit in the
first place (even if \( t_{11} \) is a read only transaction).

**Remark.** Observe that in \( \hat{H}_1 \), if \( t_1 \) (a read only transaction) commits with the results of \( t_{11} \) (i.e., \( r_{t_11}(t_\psi.x) \)), then its linearization point, at its parent \( t_\psi \)'s level, lies before \( w_{t_2}(t_\psi.x) \). In the alternate case, if \( t_1 \) commits with the results of \( t_{12} \) (i.e., \( r_{t_12}(t_\psi.y) \)), then \( t_1 \)'s linearization point lies after \( w_{t_2}(t_\psi.y) \). This way, the protocol offers flexibility for read only subtransactions.
5.4 The protocol: ParSTM

5.4.1 Protocol

### Protocol 5.2: ParSTM

<table>
<thead>
<tr>
<th>State of global object/ set :</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( t_\psi.x ) with fields</td>
</tr>
<tr>
<td>2. ( \text{val} \in V )</td>
</tr>
<tr>
<td>3. ( rs, fbd \subseteq T )</td>
</tr>
<tr>
<td>4. ( t_\psi.ow \subseteq T )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State of response object:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. ( \text{res}_s(\text{value, level, s}) )</td>
</tr>
<tr>
<td>6. ( \text{val} \in V ), set to value</td>
</tr>
<tr>
<td>7. ( \text{lvl} \in L ), set to level</td>
</tr>
<tr>
<td>8. ( s_{its} \subseteq T ), set to ( s )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Helper methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. Operation ( \text{check compatibility}(cm_{t_a}, cm_{t_d}) )</td>
</tr>
<tr>
<td>10. return ((cm_{t_a}.mts \cap cm_{t_d}.its = \emptyset) \land (cm_{t_a}.its \cap cm_{t_d}.mts = \emptyset));</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common to both nodes ( (t_x/t_p) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_x ) denotes ( t_x/t_p ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State of local object ( t_x.x ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>14. ( \text{val} \in V ), initially null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State of local atomic object ( t_x.cm ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>15. ( \text{mts} \subseteq T )</td>
</tr>
<tr>
<td>16. ( \text{its} \subseteq T )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State of transaction ( t_* ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>17. ( \text{parent} \in T ), parent’s id ( (t_p) )</td>
</tr>
<tr>
<td>18. ( \text{lvl} \in L )</td>
</tr>
<tr>
<td>19. ( \text{ls}, \text{lws} \subseteq X )</td>
</tr>
<tr>
<td>20. ( \text{vts} \subseteq T )</td>
</tr>
<tr>
<td>21. ( \text{pls} \subseteq X )</td>
</tr>
<tr>
<td>22. ( \text{prs} \subseteq X \times L )</td>
</tr>
<tr>
<td>23. ( \text{wfg} \subseteq X \times T \times T )</td>
</tr>
<tr>
<td>24. ( \text{mrg} : \text{lock} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation ( \text{begin}<em>{t</em>\psi}(t_p, \text{level}) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>25. ( t_{x.parent} \leftarrow t_p )</td>
</tr>
<tr>
<td>26. ( t_{x.lvl} \leftarrow \text{level} )</td>
</tr>
<tr>
<td>27. ( t_{x.cm.mts} \leftarrow {t_*} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation ( \text{invoke}_{child}(t_c) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>28. ( \text{begin}_{t}(t_c, \text{level} - 1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation ( \text{unlock}_{parent_locks}(s) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>29. for each ( x : x \in (s \cap t_.pl) ) do</td>
</tr>
</tbody>
</table>

| 30. \( t_.pls \leftarrow t_.pls \setminus \{x\} \) |
| 31. unlock \( t_.l; \) end for |

<table>
<thead>
<tr>
<th>32. Operation ( \text{unlock}_to_ancestors(s) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>33. ( s_{\text{ancestors}} \leftarrow \bigcup_{x \in \text{anc}(s)} t_.l(\text{level}) )</td>
</tr>
<tr>
<td>34. ( t_.unlock_parent_locks() )</td>
</tr>
<tr>
<td>35. if ( (s_{\text{ancestors}} \neq \emptyset) ) then</td>
</tr>
<tr>
<td>36. ( t_.unlock_to_ancestors(s) ) end if</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>37. Operation ( \text{get_local_locks}(x, t_1) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>38. for each ( x \in s ) do</td>
</tr>
<tr>
<td>39. ( \text{get_local_locks}(x, t_1) )</td>
</tr>
<tr>
<td>40. ( t_.pls \leftarrow t_.pls \cup {x} ) end for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>41. Operation ( \text{write}(x, v) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>42. ( \text{get_local_locks}(x, t_1) )</td>
</tr>
<tr>
<td>43. ( t_.unlock_to_ancestors(s) )</td>
</tr>
<tr>
<td>44. ( t_.unlock_to_ancestors(s) ) end if</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>45. Operation ( \text{abort}(t) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>46. for each ( t' \in \text{activeChildren}(t) ) do</td>
</tr>
<tr>
<td>47. ( t'.force_abort() ) end for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>48. Operation ( \text{force_abort}(t) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>49. for each ( t' \in \text{activeChildren}(t) ) do</td>
</tr>
<tr>
<td>50. ( t'.force_abort() ) end for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>51. Operation ( \text{abort_compat_desc}(s) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>52. ( \text{abort_compat_desc}(s) )</td>
</tr>
<tr>
<td>53. return ( (s \cap t_.pl) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>54. Operation ( \text{abort_incompat_desc}(s) ) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>55. ( \text{abort_incompat_desc}(s) )</td>
</tr>
</tbody>
</table>

85
Specific to non-root node \((t_{\pi})\):

77. \(t_{\pi}.vts \leftarrow t_{\pi}.vts \setminus s_n;\)

78. Operation \(\text{read}_{t_{\pi}}(x)\):
79. \(\text{get\_local\_lock}_{t_{\pi}}(\text{mrg}, t_{\pi});\)
80. \(\text{get\_local\_lock}_{t_{\pi}}(x, t_{\pi});\)
81. \(\text{res}_x \leftarrow \emptyset;\)
82. if \((t_{\pi}.x.\text{val} = \text{null})\) then
83. \(\text{res}_x \leftarrow \text{search\_parent}_{t_{\pi}}(x, t_{\pi}, t_{\pi}.cm);\)
84. if \((\text{res}_x = \text{null})\) then
85. \(\text{abort}_{t_{\pi}}();\)
86. \(t_{\pi}.x.\text{val} \leftarrow \text{res}_x.\text{val};\)
87. \(t_{\pi}.\text{lrs} \leftarrow t_{\pi}.\text{lrs} \cup \{x\};\)
88. \(\text{update\_cm}(t_{\pi}.cm, \emptyset, \text{res}_x.\text{its});\)
89. \(t_{\pi}.\text{prs} \leftarrow t_{\pi}.\text{prs} \cup \{x, \text{res}_x.\text{lvl}\};\)
90. \(t_{\pi}.\text{pls} \leftarrow t_{\pi}.\text{pls} \cup \{x\};\)
91. end if
92. unlock \(t_{\pi}.\text{mrg};\)
93. \(v \leftarrow t_{\pi}.x.\text{val};\)
94. unlock \(t_{\pi}.x;\)
95. \(t_{\pi}.\text{abort\_incompat\_desc}(\text{res}_x.\text{its});\)
96. return \(v;\)
97. Operation \(\text{search\_parent}_{t_{\pi}}(x, t_{\pi}, t_{\alpha}, \text{cm}_d)\):
98. if \((\neg \text{secure\_lock}_{t_{\pi}}(x, t_{\pi})\) then
99. return \(\text{null; end if;\}
100. \(\text{cm} \leftarrow t_{\pi}.cm;\)
101. if \((t_{\pi}.x.\text{val} \neq \text{null})\) then
102. if \((\neg \text{check\_compatibility}(\text{cm}, \text{cm}_d))\) then
103. unlock \(t_{\pi}.x;\)
104. return \(\text{null; end if;\}
105. \(\text{res}_x \leftarrow \{t_{\pi}.x.\text{val}, t_{\pi}.\text{lvl}, \emptyset\};\)
106. else
107. \(\text{update\_cm}(\text{cm}, \text{cm}_d, \text{mts}, \text{cm}_d.\text{its});\)
108. \(t_{\pi}.vts \leftarrow t_{\pi}.vts \cup \{t_{\alpha}\};\)
109. \(\text{res}_x \leftarrow \text{search\_parent}_{t_{\alpha}}(x, t_{\alpha}, t_{\alpha}.cm);\)
110. if \((\text{res}_x = \text{null})\) then
111. unlock \(t_{\pi}.x;\)
112. return \(\text{res}_x;\)
113. \(t_{\pi}.\text{prs} \leftarrow t_{\pi}.\text{prs} \cup \{x, \text{res}_x.\text{lvl}\};\)
114. \(t_{\pi}.\text{pls} \leftarrow t_{\pi}.\text{pls} \cup \{x\};\)
115. end if
116. return \(\text{res}_x;\)
117. Operation \(\text{try\_to\_merge}_{t_{\pi}}(t_{\pi});\)
118. \(\text{get\_local\_lock}_{t_{\pi}}(\text{mrg}, t_{\pi});\)
119. \(s \leftarrow \{x : (x, *) \in t_{\pi}.\text{pls}\}\)
120. \(t_{\pi}.\text{get\_locks}(t_{\pi}.\text{lws} \setminus s, t_{\pi});\)
121. if \((\neg \text{check\_compatibility}(t_{\pi}.cm, t_{\pi}.cm))\) then
122. unlock \(t_{\pi}.\text{mrg};\)
123. \(\text{abort}_{t_{\pi}}();\)
124. for each \(x \in t_{\pi}.\text{lws}\) do
125. \(t_{\pi}.x.\text{val} \leftarrow t_{\pi}.x.\text{val};\)
126. end for
127. for each \(x \in t_{\pi}.\text{lrs}\) do
128. unlock \(t_{\pi}.x;\)
129. \(t_{\pi}.\text{lrs} \leftarrow t_{\pi}.\text{lrs} \cup t_{\pi}.\text{lws};\)
130. \(\text{update\_cm}(t_{\pi}.cm, \text{mts}, t_{\pi}.cm.\text{its});\)
131. unlock \(t_{\pi}.\text{mrg};\)
132. \(t_{\pi}.\text{unlock\_parent\_locks}(t_{\pi}.\text{pls});\)
133. Operation \(\text{try\_to\_commit}_{t_{\pi}}();\)
134. \(\text{try\_to\_merge}_{t_{\pi}}(t_{\pi});\)
135. \(t_{\pi}.\text{abort\_incompat\_desc}(t_{\pi}.cm.\text{its});\)
136. return \(\text{commit};\)

Specific to root node \((t_{\rho})\):

Protocol 5.3: ParSTM (Special case of root node, \(t_{\rho}\))

137. Operation \(\text{search\_parent}_{t_{\rho}}(x, t_{\rho}, t_{\alpha}, \text{cm}_d)\):
138. if \((\neg \text{secure\_lock}_{t_{\rho}}(x, t_{\rho})\) then
139. return \(\text{null; end if;\}
140. \(\text{cm} \leftarrow t_{\rho}.cm;\)
141. if \((t_{\rho}.x.\text{val} \neq \text{null})\) then
142. if \((\neg \text{check\_compatibility}(\text{cm}, \text{cm}_d))\) then
143. unlock \(t_{\rho}.x;\)
144. return \(\text{null; end if;\}
145. \(\text{res}_x \leftarrow \{t_{\rho}.x.\text{val}, t_{\rho}.\text{lvl}, \emptyset\};\)
146. else
147. \(s_m \leftarrow \text{cm\_mts} \cup \text{cm}_d.\text{mts};\)
148. \(t_{\rho}.vts \leftarrow t_{\rho}.vts \cup \{t_{\alpha}\};\)
149. \(\text{lock}_{t_{\rho}}();\)
150. if \((t_{\rho}.x.\text{fb} \cap s_m = \emptyset)\) then
151. \(t_{\rho}.x.\text{rs} \leftarrow t_{\rho}.x.\text{rs} \cup \{t_{\alpha}\};\)
152. \(s_1 \leftarrow t_{\rho}.x.\text{fb} \cap t_{\rho}.\text{vts};\)
153. \(\text{res}_x \leftarrow \{t_{\rho}.x.\text{val}, t_{\rho}.\text{lvl}, s_1\};\)
154. unlock \(t_{\rho}.x;\)
155. return \(\text{res}_x;\)
156. else
157. unlock \(t_{\rho}.x;\)
158. if \((t_{\rho}.x.\text{fb} \cap t_{\rho}.\text{mts} \neq \emptyset)\) then
159. unlock \(t_{\rho}.x;\)
160. \(\text{abort}_{t_{\rho}}();\)
161. Operation \(\text{read}_{t_{\rho}}(x)\):
162. \(\text{get\_local\_lock}_{t_{\rho}}(\text{mrg}, t_{\rho});\)
163. \(\text{get\_local\_lock}_{t_{\rho}}(x, t_{\rho});\)
164. if \((t_{\rho}.x.\text{val} = \text{null})\) then
165. unlock \(t_{\rho}.x;\)
166. if \((t_{\rho}.x.\text{fb} \cap t_{\rho}.\text{mts} \neq \emptyset)\) then
167. unlock \(t_{\rho}.x;\)
168. \(t_{\rho}.x.\text{rs} \leftarrow t_{\rho}.x.\text{rs} \cup \{t_{\alpha}\};\)
169. \(t_{\rho}.x.\text{val} \leftarrow t_{\rho}.x.\text{val};\)
170. \(s_1 \leftarrow t_{\rho}.x.\text{fb} \cap t_{\rho}.\text{vts};\)
171. \(\text{update\_cm}(t_{\rho}.cm, \text{mts}, t_{\rho}.cm.\text{its});\)
172. unlock \(t_{\rho}.x;\)
173. \(t_{\rho}.\text{lrs} \leftarrow t_{\rho}.\text{lrs} \cup \{x\};\)
174. end if
175. unlock \(t_{\rho}.\text{mrg};\)
176. \( v \leftarrow t_p.x.val; \)
177. unlock \( t_p.x \)
178. \( t_p.abort\_incompat\_desc(s_i); \)
179. return \( v; \)
180. Operation try\_to\_commit\(_t_p()\):
181. if \( (t_p.lws = \emptyset) \) then
182. return commit; end if
183. for each \( t_\psi.x : x \in (t_p.lrs \cup t_p.lws) \) do
184. lock \( t_\psi.x; \)
185. \( t_p.pls \leftarrow t_p.pls \cup \{x\} \) end for
186. if \( (t_p.cm.mts \cap t_\psi.ow \neq \emptyset) \) then
187. abort\(_t_\psi(); \) end if
188. \( t_\psi.ow \leftarrow \cup_{x \in t_p.lws} t_\psi.x.rs; \)
189. for each \( x \in t_p.lws \) do
190. \( t_\psi.x.val \leftarrow t_p.x.val \)
191. \( t_\psi.x.fbd \leftarrow t_\psi.ow; \)
192. \( t_\psi.x.rs \leftarrow \emptyset \) end for
193. \( t_p.unlock\_parent\_locks(t_p.pls); \)
194. return (commit);

Preview of the protocol

Unlike SimpSTM, at each node \( t \) in the transaction tree the access to a local copy of an object, say \( t.x \), is protected by a lock. The descendants of \( t \) have to lock \( t.x \) before reading it. If the read operation fails, the lock on \( t.x \) is released. However, if the read is successful, then the lock on \( t.x \) is retained until the subtree containing that descendant completes (as discussed in Section 5.2).

For example, referring to Figure 5.3, say \( t_{1111} \) wants to read \( x \), and the value for \( x \) is available with \( t_1 \), and not with any of the intermediate ancestors, \( t_{11} \) and \( t_{111} \). As described later in the protocol, the method for reading an object \( x \) from the parent’s local space is \( search\_parent_{t_p}(x, ...) \), where \( t_p \) is the id of the parent. Then, \( t_{1111} \) calls \( search\_parent_{t_{111}}(x, ...) \), and obtains the lock on \( t_{111}.x \) in the process. Since \( t_{111} \) does not have a local copy of \( x \) with itself, \( t_{111} \) calls \( search\_parent_{t_{11}}(x, ...) \) and obtains a lock on \( t_{11}.x \). Similarly, \( t_{11} \) calls \( search\_parent_{t_1}(x, ...) \) and locks \( t_1.x \). If the read operation on \( t_1.x \) is successful, then the locks on \( t_1.x, t_{11}.x \) and \( t_{111}.x \) are not released. However, if the read operation is unsuccessful, then the locks on these objects are released (to reflect the effect that unsuccessful read operation does not lead to locking of objects).
In alternate case, in which even \( t_1 \) does not have a local copy of object \( x \), the \texttt{search\_parent}(x, ...) method propagates to \( t_\psi \), and an attempt is made to read from the global object \( t_\psi \). Here we should observe the difference in how global objects are treated. If the read operation \( t_\psi.x \) succeeds, then the lock on \( t_\psi.x \) is released, but we retain the locks on the objects of the ancestors in the transaction tree, i.e., \( t_{111}.x, t_{11}.x \) and \( t_1.x \). In case the read is unsuccessful, then the locks on all of these objects are released.

Further, note the following. When a subtransaction \( t_c \) writes an object \( t_c.x \) which has not been previously read from the parent, \( t_p \), then a local write operation on \( t_c.x \) does not entail obtaining a lock on the parent’s object \( t_p.x \).

### 5.4.2 State of shared objects

**Globally shared objects:** Same as used for SimpSTM in Chapter 4. Recall that, at the global level, each object \( t_\psi.x \) is protected by a lock, and has the following fields: (1) \texttt{val} for value, (2) a set \texttt{rs} for storing ids of transactions that have read \( t_\psi.x \), and (3) a set \texttt{fbd} for storing ids of transactions forbidden to access \( t_\psi.x \). The set \( t_\psi.ow \) is used to indicate ids of transactions that read an object whose value has been overwritten later.

**Locally shared objects:** A local copy \( t.x \), available (locally) with a transaction \( t \), has only the value field, and is protected by a lock.

**Response object:** A response object, \( res_x \), is used as a data structure to communicate \( ( \text{value, level, a set containing incompatible transactions} ) \) information across levels while reading a value of object \( x \) from higher levels.
5.4.3 State of transaction

Each transaction \( t \) keeps a reference to its parent’s id using \( \text{parent} \) (also denoted by \( t_p \)). The information regarding the level of \( t \) is contained in \( \text{lvl} \). The read and write steps are logged using sets \( \text{lrs} \) (local read set) and \( \text{lws} \) (local write set) respectively. The ancestors’ objects on which \( t \) acquires locks are recorded in \( \text{pls} \) (parent’s lock set). The set \( \text{prs} \) (pessimistic read set) records, for each object read, the object id as well as the original level from which its value was obtained. The descendants of \( t \) that have visited it are tracked though set the \( \text{vts} \) (visited transaction set) kept in an atomic variable. Finally, \( t \) also maintains a wait-for graph, \( \text{wfg} \) that is used by its children and \( t \) itself to detect and resolve deadlock situation while acquiring a lock on \( t \)’s objects. An atomic object \( \text{cm} \) (consistency management) contains sets \( \text{mts} \) and \( \text{its} \) that are used to keep track of merged transactions and incompatible transactions respectively. Here also the atomic object used is an atomic register that can perform read and write together atomically, as mentioned in the Section 4.1.2.1. Further, lock \( \text{mrg} \) is used while searching for a local copy at the parent level, and during the merging of a child’s sets with those of its parent.

5.4.4 Methods common to both root as well as non-root nodes \((t_*)\)

\text{begin}_{t_*}(t_p, \text{level}): \) Each transaction \( t_* \) begins with this method, where \( t_p \) denotes the id of its parent and \( \text{level} \) denotes its level in the super tree. The set \( \text{mts} \) (merged transaction set) is initialized with \( t_* \).

\text{invoke\_child}_{t_*}(t_p): \) This method is used by transaction \( t_* \) to invoke a new child
transaction $t_c$.

$unlock\_parent\_locks_{t_\ast}(s)$: This method is used by transaction $t_\ast$ to release the locks on the parent objects in set $s$.

$check\_compatibility(cm_{t_\ast}, cm_{t_c})$: It is used to check the compatibility between two transactions using their respective consistency management objects.

$update\_cm(cm, s_m, s_i)$: It is used to perform the update of an atomic object $cm$, whereby contents of $s_m$ and $s_i$ are respectively appended to corresponding sets $mts$ and $its$ of $cm$. Owing to the atomic nature of $cm$, this update occurs in a single atomic step.

$get\_local\_lock_{t_\ast}(x, t_1)$: This method is used to obtain a lock on object $t_\ast.x$ for transaction $t_1$. In case of failure to lock $t_\ast.x$, transaction $t_1$ is aborted.

$get\_locks_{t_\ast}(s, t_c)$: This method is invoked by $t_\ast$’s child $t_c$ to obtain locks on $t_\ast$’s objects in set $s$. If an attempt to lock $t_\ast.x$ using the secure_lock method is successful, then $x$ is added to $t_c.pls$ (and $prs$) to indicate that $t_c$ is in possession of a lock on its parent’s object $t_\ast.x$.

$write_{t_\ast}(x, v)$: self-explanatory.

$abort_{t_\ast}()$: Upon aborting, $t_\ast$ releases all the ancestral locks acquired through its own operations or those of its descendants. Next, as the children of the aborted transaction have to be aborted as well, $t_\ast$ invokes the $abort\_active\_desc$ method of all of its active children.

$abort\_active\_desc_{t_\ast}()$: This method is used by the parent $t_\ast$ to force the abort of its active descendants. This is issued as a consequence of an abort of $t_\ast$.

$force\_abort_{t_\ast}()$: This method is used to abort a descendant such that it immediately returns $abort$, without having the need to release the parent locks in its poses-
This invoked by an aborting ancestor.

\textit{abort\_incompat\_desc}_{t_{*}}(s_{i}): This method is used to abort the incompatible descendants of \( t_{*} \). The set \( s_{i} \) contains the ids of subtransactions that are incompatible with \( t_{*} \). This method propagates down the subtree rooted at \( t_{*} \) in cascading manner. At each of the descendants \( t_{d} \) we check if \( t_{d} \) is incompatible with \( t_{*} \). If \( t_{d} \) is found incompatible, then we abort (discard) the subtree rooted at \( t_{d} \). Otherwise, we check each of the active children of \( t_{d} \) iteratively.

\subsection{5.4.5 Methods specific to non-root nodes (\( t_{\pi} \))}

\textit{read}_{t_{\pi}}(x): To read an object \( x \), transaction \( t_{\pi} \) locks its local object \( t_{\pi}.x \). If \( t_{\pi}.x.val \) is not null, then the value of \( t_{\pi}.x \) is returned after unlocking \( t_{\pi}.x \). In case \( t_{\pi}.x \) is null-valued, then \( t_{\pi} \) tries to read the value from its ancestors, using the method \textit{search\_parent}. Before \( t_{\pi} \) invokes \textit{search\_parent} method, it locks \( t_{\pi}.mrg \) to ensure that no incompatible child merges with it while it tries to read the value of \( x \) from its ancestor. Now, if the read is not successful, then \( t_{\pi} \) aborts. Otherwise, the value of \( t_{\pi}.x \) is updated. Further, \( x \) is added to \( t_{\pi}.pls \) to indicate the ownership of parent lock, and \( prs \) is updated to record the external read. The set \( t_{\pi}.cm.its \) is updated with any incompatible descendant using \( res_{x}.s_{its} \). The lock \( t_{\pi}.mrg \) is unlocked and incompatible descendants (if any) of \( t_{\pi} \) are aborted. Object \( x \) is added to \( t_{\pi}.lrs \). Finally, \( t_{\pi}.x \) is unlocked and the value of \( t_{\pi}.x \) is returned.

\textit{search\_parent}_{t_{\pi}}(x,t_{c},t_{o},cm_{d}): This method propagates recursively in a bottom-to-top manner. This method of \( t_{\pi} \) is invoked by its child transaction, \( t_{c} \), to search for a local copy of object \( x \) available with \( t_{\pi} \). The descendant originally trying to read
the value of \( t_\pi.x \) is \( t_o \). First, a lock on \( t_\pi.x \) is obtained. In case the value of \( t_\pi.x \) is non-null, the compatibility of \( t_o \) and its ancestors up to \( t_c \) with \( t_\pi \) is checked. If compatible, then a response object \( res_x \), containing \( \langle t_\pi.x.val, t_\pi.lvl, \emptyset \rangle \), is returned. Otherwise, \( null \) is returned to indicate an unsuccessful read. Alternatively, if a local (non-null valued) copy of \( x \) is not available with \( t_\pi \), then \( t_\pi \) forwards the search for a local copy of \( x \) to its parent node. If the \( res_x \) object obtained from the parent is null, then \( t_\pi.x \) is unlocked before returning \( null \). Otherwise, \( \langle x, res_x.lvl \rangle \) is recorded in \( t_\pi.prs \) and \( t_\pi.pls \) is updated before returning \( res_x \).

\[ try\_to\_merge_{t_\pi}(t_c): \] This method is invoked by the child transaction \( t_c \). The steps are self-explanatory. However, it should be noted here that the order of steps in line 118 (obtaining \( mrg \) lock) and line 120 (locking of parent’s objects) is important in order to avoid a deadlock situation between the child and the parent.

Observe that parent \( t_\pi \), during it read step, obtains the lock \( t_\pi.mrg \) (line 79) first and then locks its local object \( t_\pi.x \) (line 80) before reading from its ancestor. Here, during the merging phase, the child transaction \( t_c \) also tries to first obtain the lock on \( t_\pi.mrg \) (line 118) before object \( t_\pi.x \) (in case it does, line 120). Only the transaction successful in obtaining the lock on \( t_\pi.mrg \) proceeds to lock \( t_\pi.x \). Note that locking of both \( t_\pi.mrg \) and \( t_\pi.x \) is done through \textit{secure\_lock} method to ensure any cyclic dependency is captured in the wait-for graph \( t_\pi.wfg \). In case a deadlock is detected, the transaction will be aborted. After the merge is complete, locks are released up to the parent level.

\[ try\_to\_commit_{t_\pi}(): \] self-explanatory.
5.4.6 Methods specific to root-node \((t_\rho)\)

\texttt{search\_parent}_{t_\rho}(x,t_c,t_o,cm_o): The behaviour of \texttt{search\_parent} method associated with the root transaction \((t_\rho)\) is somewhat different from the one associated with non-root transactions. This is due to that fact that, if a local copy of the desired object is not available with the root transaction, then it tries to read directly from the globally shared copy of the object.

First, it secures a lock on its local object \(t_\rho.x\). If \(t_\rho.x.val\) is non-null and \(t_o\) along with \(t_o\)'s ancestors up to \(t_c\) are compatible with \(t_\rho\), then a response object \(res_x\) containing \(\langle t_\rho.x.val,t_\rho lvl,\emptyset \rangle\), is returned. If not compatible, then the lock on \(t_\rho.x\) is released and null is returned to indicate failure.

If \(t_\rho.x\) is null-valued, then an attempt is made to read \(t_\psi.x\). At this point, \(s_m\) is updated with the \(mts\) of all the nodes in the path from \(t_o\) to \(t_\rho\) (including \(t_\rho\)). Subtransaction \(t_o\) is added to \(t_\rho.vts\). Now, after locking \(t_\psi.x\), it is checked if any transaction in \(s_m\) is forbidden to access \(t_\psi.x\) (i.e., present in \(t_\psi.x.fbd\)). If yes, then \(t_\psi.x.rs\) is unlocked and null is returned to indicate failure. Otherwise, \(t_o\) is added to \(t_\psi.x.rs\) before unlocking \(t_\psi.x.rs\), and a response object containing \(\langle t_\psi.x.val,t_\rho lvl,s \rangle\) (\(s\) contains ids of descendants of \(t_o\), if any, forbidden to read \(t_\psi.x\)) is returned.

\texttt{read}_{t_\psi}(x): If a local copy of an object \(x\) is not available, then a non-root node calls the \texttt{search\_parent} method of its parent. In contrast, a root node, having no parent, tries to read directly from the globally shared copy in that case. First, it locks the merge lock \(t_\rho.mrg\) and local copy \(t_\rho.x\) in order. If \(t_\rho.x\) is null-valued, then it locks \(t_\psi.x\). Next, it checks the consistency of its step by ensuring that no transaction in \(t_\rho.cm.mts\) belongs to \(t_\psi.x.fbd\). If the check fails, then \(t_\psi.x\) is unlocked and \(t_\rho\) aborts.
Otherwise, \( t_\rho \) is added to \( t_\psi.x.rs \), \( t_\rho.x.val \) is updated using \( t_\psi.x.val \), and \( t_\rho.cm.its \) is updated using \( t_\psi.x.fbd \) and \( t_\rho.vts \), before unlocking \( t_\psi.x \) and \( t_\rho.mrg \). Incompatible descendants (if any) of \( t_\rho \) are aborted and \( x \) is added to \( t_\rho.lrs \). Finally the value of \( t_\rho.x \) is returned after unlocking it.

\[
\text{try\_to\_commit}_{t_\rho}(): \text{self-explanatory.}
\]

5.4.7 Regarding abort of a transaction and its descendants

When a transaction \( t \) in a transaction tree aborts, the execution of the entire subtree rooted at \( t \), i.e., \( \text{subTree}(t) \), has to be discarded and hence all the transactions in the \( \text{subTree}(t) \) are aborted. When a subtransaction aborts, the key thing is to release the locks on objects of \( t \)’s ancestors acquired by transactions in \( \text{subTree}(t) \).

By construction of ParSTM, whenever \( t \) or any of its descendants obtains a lock on an object of \( t \)’s ancestor, it is duly recorded at \( t \)’s level (lines 113-114, 89-90 during external read; 49 during merging). Thus, when \( t \) aborts, it can act on behalf of the entire transactions in \( \text{subTree}(t) \) and releases the locks of its ancestors’ objects acquired by transactions in \( \text{subTree}(T) \). Subsequently, when the descendants of \( t \) are forced to abort due to the abort of their ancestor \( t \), they do not need to worry about releasing any locks in their possession (lines 58, 62, 66). This is owing to the fact that (i) an abort of \( t \) means that execution at the subtree level \( t \) is suspended anyway, and (ii) locks on objects of \( t \)’s ancestors are already released by \( t \) (line 56-57).

The same strategy is followed for an abort of a transaction and its descendants in subsequent Chapters 6 and 7.
5.4.8 Optimization: abort of incompatible descendants

By construction of ParSTM, observe that during an external read or upon merging of a child, an ancestor $t_a$ can detect early that its descendant, say $t_d$, is incompatible with it (due to line 88 or 130). As an incompatible child is not allowed to merge with its parent (due to line 121), it follows that $t_d$ is bound to abort eventually when an attempt will be made to merge its execution with that of its ancestor $t_a$. To this end, we optimize by forcing the abort of incompatible descendants, such as $t_d$, the moment the ancestor $t_a$ identifies them as incompatible (lines 95 or 135). This policy of preemptive abort of incompatible descendants by an ancestor has been followed in the protocols discussed in forthcoming Chapters 6 and 7 as well.

5.5 Consistency checking and linearization points

at level $t$

5.5.1 Consistency checking during external read operation

When a descendant $t_d$ tries to read a value from its ancestor $t$, the consistency checking (line 102 or 142) at $t$ involves use of its local sets $t.cm$ which are not protected by any lock during this operation. During this check, it is possible that the contents of $t.cm$ may change, either due to concurrent external read of $t$ (line 88) or merging of its child (line 130).
5.5.2 Linearization points of events in a level-wise history

The level wise event history $\widehat{H}_t$ consists of the following events: (i) local read/write operations of $t$, (ii) external reads of $t$’s descendants w.r.t. $t$, (iii) external reads of $t$ itself, (iv) write operations due to merging of $t$’s children $t_c$, and (v) commits of $t$’s children.

Let $\ell_{op}$ denote the linearization point of an event. Then, the linearization points of the various events in the history are defined as follows:

i Local read/write operation of $t$

(a) $\text{read}_t(t.x) : \ell_{op}$ corresponds to the time when it unlocks $t.x$ (line 94 or 177)

(b) $\text{write}_t(t.x) : \ell_{op}$ corresponds to the time when it unlocks $t.x$ (line 53)

ii External read operation of $t$

(a) $\text{read}_t(t_a.x) : \ell_{op}$ corresponds to the time just after $t.cm$ is updated (line 88 or 171)

iii External read of a descendant $t_d$ on $t$’s object or that of $t$’s ancestor $t_a$:

(a) $\text{read}_{t_d}(t.x) : \ell_{op}$ corresponds to the time just after $t_d$ reads $t.cm$ for consistency checking (line 100 or 140)

(b) $\text{read}_{t_d}(t_a.x) : \ell_{op}$ corresponds to the time just after $t_d$ reads $t.cm$ for consistency checking (line 100 or 140)

iv Write due to merging of child $t_c$

(a) $\text{write}_{t_c}(t.x) : \ell_{op}$ corresponds to the time just after $t_c$ updates $t.x$ (line 125 or 127)

v Commit of child $t_c$
(a) $C_t \cdot \ell_{op}$ corresponds to the time just after $t.cm$ is updated (line 130)

5.5.3 Ordering of external read/search at $t$ with overlapping local operations of $t$

Let $t_d$ be a descendant of $t$ that performs the read operation $read_{td}(t.x)$ on object $t.x$.

Let the events of reading the atomic object $t_2.cm$ by $t_1$ be denoted by $R_{t_1}(t_2.cm)$ and its local update by $W_{t_2}(t_2.cm)$. Then, the ordering of $read_{td}(t.x)$ during the following concurrent operation can be specified as follows:

i External read operation $read_{t}(t_a.y)$:

As update of $t.cm$ is atomic in nature, events $R_{td}(t.cm)$ (line 100 or 140) and $W_{t}(t.cm)$ (line 88 or 171) are linearizable, i.e., either $R_{td}(t.cm) < W_{t}(t.cm)$ or $R_{td}(t.cm) > W_{t}(t.cm)$. Consequently, the ordering of $read_{td}(t.x)$ and $read_{t}(t_a.y)$ follows the same order as $R_{td}(t.cm)$ and $W_{t}(t.cm)$.

ii Local merge/commit operation try_to_merge$_t(t_c)$:

Similarly, here events $R_{td}(t.cm)$ (line 100 or 140) and $W_{t}(t.cm)$ (line 130) are linearizable. We have either $R_{td}(t.cm) < W_{t}(t.cm)$ or $R_{td}(t.cm) > W_{t}(t.cm)$. Accordingly, $read_{td}(t.x)$ and $C_{tc}$ are ordered as well.

Two concurrent read operations $read_{td1}(t.x)$ and $read_{td2}(t.y)$ can be ordered arbitrarily w.r.t. to each other.

5.5.4 Linearization point of nested transaction

Definition of linearization point $\ell_t$ of a transaction $t$:

Case I: $t$ is a non-root transaction ($t_\pi$)
1. If \( t \) commits, its linearization point, \( \ell_t \), lies at the time just after it updates the parent’s \( cm \) (consistency management) object (line 130).

2. If \( t \) aborts, \( \ell_t \) lies at the time it accessed \( t_x.cm \) for the consistency check of its last successful read operation (lines 100 or 140).

Case II: \( t \) is a root transaction \((t_\rho)\)

3. If \( t \) is a read only transaction that commits, \( \ell_t \) is placed at the earliest of (1) the time of the test during its last read operation (line 150 or 166) and (2) the time just before \( \hat{t} \) (any id in \( t.cm.mts \)) is added to \( t_\psi.ow \), if it ever is (line 188).

4. If an update transaction \( t \) commits, \( \ell_t \) is placed just after the execution of line 188 by \( t \) (update of \( t_\psi.ow \)).

5. If transaction \( t \) aborts, \( \ell_t \) is placed just before \( \hat{t} \) is added to the set \( t_\psi.ow \) (line 188 of the \texttt{try\_to\_commit}_t() operation that entails its abort).

5.6 Proof

Here, the set of proofs is divided into three parts, dealing with: (1) (optimistic part) the history of committed transactions at the global level \((\Pi(\hat{H}_t_\psi))\), (2) (pessimistic part) the level-wise history of committed transactions produced at the nodes \((\Pi(\hat{H}_t_\pi))\) of a transaction tree, and (3) aborted transactions.

First, considering the history of committed transactions, we present the proof for the optimistic part, using the history produced at the global level \((\hat{H}_t_\psi))\), and the
proof for the pessimistic part using the history produced at the nodes ($H_{t_n}$) in the transaction tree.

*Note:* All the line numbers under this section refer to Protocol 5.2 and 5.3 (Section 5.5).

### 5.6.1 Proof for committed transactions

In this section, we only consider the histories restricted to committed transactions.

**Part I: History ($\hat{H}_{t_\psi}$) produced at the global level ($t_\psi$)**

$\hat{H}_{t_\psi}$ deals with the linearizability of root transactions only. Thus, in effect, it is as good as dealing with non-nested transactions. The case of a root transaction here is similar to the one under SimpSTM. Let us observe the similarities for a root transaction $t_\rho$ under SimpSTM and ParSTM.

- Set $t_\rho.cm.mts$ contains the ids of $t_\rho$ and its descendants that have successfully merged with it.

- Sets $t_\rho.lrs$ and $t_\rho.lws$ are used to record read and write operations respectively by $t_\rho$ or its descendants in $t_\rho.cm.mts$

- $t_\rho.cm.mts$ is used for consistency checking during the read operation on a global object $t_\psi.x$ ($t_\psi.x.fbd \cap t_\rho.cm.mts = \emptyset$) as well as during commit process ($t_\psi.ow \cap t_\rho.cm.mts = \emptyset$).

The correctness for the root transactions can be proved in the same way as done in Chapter 4. The only extra requirement here is the proof for the incompatible
transactions. As discussed earlier, the merging of incompatible subtransactions with
the parent would render the parent non-linearizable at the higher level. Hence, we
need to show that incompatible transactions will not be merged together.

**Lemma 5.1.** Let $\hat{H}_t$ be a level-wise history. Let $t_1$ and $t_2$ be any two distinct transac-
tions in $\{t \cup \text{children}(t)\}$. If \(r_{\hat{t}_2}(t_{\psi}x) : \beta(\hat{t}_1,t_{\psi}x.fbd,AL_{\hat{t}_2}(t_{\psi}x,read_{\hat{t}_2}(t_{\psi}x)))\), then we
have (1) if $t_1, t_2 \neq t$, then $\neg(t_1 \in \Pi(\hat{H}_t) \land t_2 \in \Pi(\hat{H}_t))$ or (2) $t_1 = t \land t_2 \notin \Pi(\hat{H}_t)$
or (3) $t_2 = t \land t_1 \notin \Pi(\hat{H}_t)$.

**Proof.** Here, incompatibility of transactions comes into the picture when subtransac-
tions read from the globally shared objects.

Since $t_1$ and $t_2$ are distinct (incompatible) transactions, either both the transac-
tions are children of $t$, or one of the two is $t$, while the other one is a child of $t$. First,
we show that

\[ r_{\hat{t}_2}(t_{\psi}x) : \beta(\hat{t}_1,t_{\psi}x.fbd,AL_{\hat{t}_2}(t_{\psi}x,read_{\hat{t}_2}(t_{\psi}x))) \Rightarrow t_1 \in t_2.cm.its. \]

\[ r_{\hat{t}_2}(t_{\psi}x) : \beta(\hat{t}_1,t_{\psi}x.fbd,AL_{\hat{t}_2}(t_{\psi}x,read_{\hat{t}_2}(t_{\psi}x))) \]
means that when subtrans-
action $\hat{t}_2$ acquired lock on $t_{\psi}x$ to perform a read operation, $\hat{t}_1 \in t_{\psi}x.fbd$. Observe
that, by construction, before releasing the lock on $t_{\psi}x$, transaction $\hat{t}_2$ adds $\hat{t}_1$ to
$\hat{t}_2.cm.its$ (lines 152-155, 88; 171) using intersection of $t_{\rho}.vts$ and $t_{\psi}.x.fbd$ (due to line
148). Later, $\hat{t}_2$ releases the lock on its local copy $\hat{t}_2.x$ (lines 94, 177). Thus, we have
following implications:

\[ r_{\hat{t}_2}(t_{\psi}x) : \beta(\hat{t}_1,t_{\psi}x.fbd,AL_{\hat{t}_2}(t_{\psi}x,read_{\hat{t}_2}(t_{\psi}x))) \Rightarrow \beta(\hat{t}_1,\hat{t}_2.cm.its,RL_{\hat{t}_2}(t_{\psi}x,read_{\hat{t}_2}(x))) \]
\Rightarrow $\hat{t}_1 \in \hat{t}_2.cm.its$ (due to line 153-155, 88; 170-171).

Recall that $\hat{t}_2$ denotes a (sub)transaction in $t_2.cm.mts$. If $\hat{t}_2 \neq t_2$, then it means
$\hat{t}_2$ is $t_2$’s descendant that merged with $t_2$, and in the process merged its its with that
of $t_2$ (line 130).

Therefore, $\hat{t}_1 \in \hat{t}_2.cm.its \land \hat{t}_2 \in t_2.mts \Rightarrow \hat{t}_1 \in t_2.cm.its$. 

$\Rightarrow$ $t_1$ and $t_2$ are incompatible transactions.

Now, we want to show that the two incompatible transactions, $t_1$ and $t_2$, cannot be merged together. Let us consider the first case in which $t_1, t_2$ are children of $t$. We have to show $t_1 \in \Pi(\widehat{H}_t) \Rightarrow t_2 \notin \Pi(\widehat{H}_t)$ and vice versa.

**Case I: $t_1, t_2 \in children(t)$.**

Observe that, for merging with its parent, each child transaction $t_c$ has to first obtain its parent’s ($t_p$) lock $t_p.mrg$ (line 118). This ensures that only one child of $t_p$ merges with it at a time. Now, we have the following two subcases to consider:

**Case I(a):** $AL_{t_1}(t.mrg, ttc) <_{H_t} AL_{t_2}(t.mrg, ttc)$ (assuming $t_1 \in \Pi(\widehat{H}_t)$).

$AL_{t_1}(t.mrg, ttc) <_{H_t} AL_{t_2}(t.mrg, ttc)$

$\Rightarrow RL_{t_1}(t.mrg, ttc) <_{H_t} AL_{t_2}(t.mrg, ttc) \Rightarrow \beta(\hat{t}_1, t.cm.mts, RL_{t_1}(t.mrg, ttc))$ (due to line 130)

$\Rightarrow \beta(\hat{t}_1, t.cm.mts, AL_{t_2}(t.mrg, ttc))$

This means, when $t_2$ locks $t.mrg$ to merge with $t$, it will discover that it is incompatible with $t$ as $t.cm.mts$ already contains $\hat{t}_1$, i.e., $\hat{t}_1 \in t.cm.mts \cap t_2.cm.its$. Consequently $t_2$ aborts (due to line 121-123). Thus, $t_1 \in \Pi(\widehat{H}_t) \Rightarrow t_2 \notin \Pi(\widehat{H}_t)$.

**Case I(b):** $AL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc)$ (assuming $t_2 \in \Pi(\widehat{H}_t)$).

The proof is symmetric to **Case I(a).** $AL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc)$

$\Rightarrow RL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc) \Rightarrow \beta(\hat{t}_1, t.cm.its, RL_{t_2}(t.mrg, ttc))$ (due to
In this case, \( t_1 \) aborts later on (due to lines 121-123) as \( t.cm.its \) already contains \( \hat{t}_1 \), i.e., \( t.cm.its \cap t_1.cm.mts = \hat{t}_1 \neq \emptyset \). We have \( t_2 \in \Pi(\mathcal{H}_t) \Rightarrow t_1 \notin \Pi(\mathcal{H}_t) \).

Case II: Either \( t_1 = t \), or \( t_2 = t \).

Case II(a): \( t_1 = t \).

\( t_1 = t \Rightarrow t_1 \) is the parent of \( t_2 \)

\( t_1 \) is an ancestor of each \( \hat{t}_2 \in t_2.cm.mts \).

\( \Rightarrow \) read operation \( r_{\hat{t}_2}(t_\psi.x) : \beta(\hat{t}_1, t_\psi.x.fbd, AL_{\hat{t}_1}(t_\psi.x, read_{\hat{t}_2}(t_\psi.x))) \) is not possible (failure due to lines 107, 155, 150; 159, 84-85).

Case II(b): \( t_2 = t \).

Similar to Case I(b). \( t_2 \), being the ancestor \( t \), adds \( \hat{t}_1 \) to \( t.cm.its \) anyway, leading to abort of \( t_1 \) right away (due to lines 153-155, 88; 170-171; 102-104; 153-155; 83-85).

Further, observe that \( t \) might be involved in reading from its ancestor a value that is incompatible with its child that is trying to merge with it. This situation is avoided by ensuring that, while \( t \) is reading (from its ancestors), its child cannot merge with it. This is due to the fact that, during its read operation, \( t \) locks \( t.mrg \) which is required to be locked by its child transaction in order to merge with \( t \).

Thus, the conjunction of the cases I and II proves the lemma.

\( \square \)
Lemma 5.1 shows that the read set of a transaction cannot contain incompatible read operations, thereby ensuring its linearizability at ancestor levels.

**Theorem 5.1.** The level-wise history $\Pi(\mathcal{H}_{t_\pi})$ of committed transactions, produced at the global level, satisfies level-wise opacity.

*Proof.* The proof follows from the combination of the definition of linearization points (2 and 3) for root transactions, Lemma 5.1, and on the basis of the set of proofs (Lemmas 4.3, 4.4 and 4.5) outlined for root transactions in Chapter 4.

Part II: History ($\mathcal{H}_{t_\pi}$) produced at a node $t_\pi$ of transaction tree

We shall prove that the execution of the children of $t_\pi$ follows the 2PL protocol for nested transactions, i.e., the children of $t_\pi$ hold the lock on the parent’s ($t_\pi$) objects in a pessimistic manner. They retain the lock on the parent’s object until they complete (commit/abort). In other words, we shall prove that any two children of $t_\pi$, say $t_1$ and $t_2$, can execute concurrently only if they do not operate on a common object $t_\pi.x$. We show that 2PL policy is followed by the child transaction for locking its parent’s objects through the following two points.

Let node $t_c$ be a child of node $t$ in a transaction tree.

1. *If $t_c$ successfully acquires a lock on $t$’s local object $t.x$, then the lock on $t.x$ is not released until all the transactions in the subtree rooted at $t_c$ complete (commit or abort).*

2. *When $t_c$ commits or aborts, it releases all the locks in its possession at its parent ($t$) level.*

103
Next, we provide the set of proofs to show that the history produced by ParSTM at the nodes of a transaction tree satisfies the above conditions.

**Lemma 5.2.** Let $t$ be a child of $t_\pi$. If $t$ acquires a lock on $t_\pi$’s object, $t_\pi.x$, then $t$ releases the lock on $t_\pi.x$ only upon its completion.

**Proof.** By construction, $t$ acquires a lock on $t_\pi.x$ either during its read operation (lines 98, 138) or while trying to merge with the parent (line 120 during `try_to_merge`). If $t.x$ is null-valued, then $t$ tries to obtain a value for $t.x$ by invoking the method `search_parent_{t_\pi}` (line 109) of the parent. Note that, in the definition of the method `search_parent_{t_\pi}`, there is locking of object $t_\pi.x$ (lines 98, 138), but no unlocking of $t_\pi.x$ in case of a successful read operation. Unlocking is done only if the read is unsuccessful (lines 102-103, 110-111, 142-143). The only other case when $t$ locks $t_\pi$’s objects is during the validation phase (line 120, `try_to_merge` method) towards the end of $t$’s execution. Again by construction of the protocol, the lock on $t_\pi.x$ is released upon commit (successful validation; lines 132) or on abort (line 123, 55-57). Thus, $t$ retains the lock on $t_\pi.x$ until its completion. \qed

**Lemma 5.3.** If $t$ has read from $t_\pi.x$, then no other transaction, $t_\pi$ or any other child of $t_\pi$, can modify $t_\pi.x$ until $t$ completes its execution.

**Proof.** If $t$ has read $t_\pi.x$, then it means that $t$ currently holds the (exclusive) lock on $t_\pi.x$ (lines 98, 138). By construction, any read (lines 80, 98, 138, 163) or write (lines 51, 119) operation is controlled using a lock associated with $t_\pi.x$. That means, to operate on $t_\pi.x$, $t_\pi$ or its children must first acquire a lock on $t_\pi.x$. The lock on $t_\pi.x$ can be acquired by another transaction ($t_\pi$ or any of its children) only after $t$ releases the lock on $t_\pi.x$ first. Since $t$ releases the lock on $t_\pi.x$ only upon its completion
(by Lemma 5.2), it means $t_x.x$ cannot be modified by another transaction until $t$ completes.

**Lemma 5.4.** Let $t_1 \xrightarrow{t_x.x}_\alpha t_2$ be the relation defined as: $t_2$ accesses (the parent’s object) $t_x.x$ after it was accessed by $t_1$ for its read/write operation. Then, $t_1 \xrightarrow{t_x.x}_\alpha t_2 \Rightarrow t_1 \rightarrow_{H_t} t_2$.

**Proof.** We have $t_1 \xrightarrow{t_x.x}_\alpha t_2 \Rightarrow AL_{t_1}(t_x.x, r/w) <_{H_{t_x}} AL_{t_2}(t_x.x, r/w)$

$\Rightarrow AL_{t_1}(t_x.x, r/w) <_{H_{t_x}} RL_{t_1}(t_x.x, ttc) <_{H_{t_x}} AL_{t_2}(t_x.x, r/w) <_{H_{t_x}} RL_{t_2}(t_x.x, ttc)$

(using Lemma 5.2).

$\Rightarrow RL_{t_1}(t_x.x, ttc) <_{H_{t_x}} RL_{t_2}(t_x.x, ttc)$

$\Rightarrow \ell_{t_1} <_{H_{t_x}} \ell_{t_2}$

$\Rightarrow t_1 \rightarrow_{H_{t}} t_2$.

**Lemma 5.5.** The level wise event history $\hat{H}_t$ at node $t$ is linearizable.

**Proof.** The proof follows from the definition of the linearization points of the events for a level wise history, in Section 5.2.2.

While a descendant $t_d$ reads or checks for a value at an ancestor $t$, several other operations (external reads of other transactions, $t$’s local or external read operation, or commit of $t$’s child) could be happening concurrently at $t$. Descendant $t_d$ can become incompatible with $t$ either due to external read of $t$ or commit of a $t$’s child. We shall show that the consistency checking on the value read by $t_d$ from $t$ is guaranteed to be correct.

Now, similar to the definition of $\beta(t_1, t.ow, \tau)$, let us define $\gamma_{\text{inc}}(t_1, t, \tau)$ to indicate that transaction $t_1$ becomes incompatible with transaction $t$ at time $\tau$. In other words,
at time \( \tau \), \( \text{check\_compatibility}(t\_cm, t.cm) = false \). Using this definition, we shall prove the next lemma.

**Lemma 5.6.** Let \( t_d \) be a descendant of \( t \) such that history \( \text{op}_t, \text{read}_{t_d}(t.x) \in \widehat{H}_t \), where \( \text{op}_t \) denotes an external read operation or a commit of \( t \)'s child at time \( \tau \). Then, we show that (i) \( \text{op}_t, \text{read}_{t_d}(t.x) \) can be ordered, and (ii) \( \text{op}_t < \text{read}_{t_d}(t.x) \Rightarrow \neg \gamma_{\text{inc}}(t_d, t, \tau) \), i.e., \( t \) and \( t_d \) are not inconsistent before time \( \tau \).

**Proof.** By definition of the linearization points for \( \text{op}_t \) and \( \text{read}_{t_d}(t.x) \) in \( \widehat{H}_t \) w.r.t. access to \( t.cm \), and \( t.cm \) being an atomic variable, it follows \( \text{op}_t, \text{read}_{t_d}(t.x) \in \widehat{H}_t \) are linearizable, i.e., \( \text{op}_t < \text{read}_{t_d}(t.x) \) in \( \widehat{H}_t \) or \( \text{op}_t > \text{read}_{t_d}(t.x) \) in \( \widehat{H}_t \).

To prove part (ii), let us assume by contrast that \( \gamma_{\text{inc}}(t_d, t, \tau) \), i.e., at the completion of \( \text{op}_t \) at time \( \tau \), \( \text{check\_compatibility}(t.cm, t_d.cm) = false \).

Now, for \( \text{op}_t < \text{read}_{t_d}(t.x) \), we have the following two cases:

**Case I: \( \text{op}_t \) is an external read operation of \( t \)**

By definition of linearization points for events in a level wise history (Section 5.5.3), we have

\[
\ell_{\text{op}_t} < \ell_{\text{read}_{t_d}(t.x)}
\]

\( \Rightarrow \) update of \( t.cm \) (line 88 or 171) such that \( \text{check\_compatibility}(t.cm, t_d.cm) = false \) occurs before \( t.cm \) is read (line 100 or 140) for consistency check of \( t_d \). (Recall that \( t.cm \) is kept in an atomic variable.)

\[
\gamma_{\text{inc}}(t_d, t, \tau) < \ell_{\text{read}_{t_d}(t.x)}
\]

\( \Rightarrow \text{check\_compatibility}(t.cm, t_d.cm) = false \) during consistency check for \( \text{read}_{t_d}(t.x) \) (line 102 or 142)

\( \Rightarrow \text{read}_{t_d}(t.x) \) fails (due to lines 102-104 or 142-144), i.e., \( \text{op}_t < \text{read}_{t_d}(t.x) \) is not
possible in this case.

Case II: \( \text{op} \) is commit of \( t \)'s child

By definition of the linearization points for events in a level wise history (Section 5.5.3), we have

\[
\ell_{\text{op}} < \ell_{\text{read}_{\text{td}}(t.x)}
\]

\[\Rightarrow \text{update of } t.cm \text{ (line 130) such that } \text{check\_compatibility}(t.cm, t_{d}.cm) = false \text{ occurs before } t.cm \text{ is read (line 100 or 140) for consistency check of } \text{read}_{\text{td}}(t.x). \text{ (Recall that } t.cm \text{ is kept in an atomic variable.)}
\]

\[
\gamma_{\text{inc}}(t_{d}, t, \tau) < \ell_{\text{read}_{\text{td}}(t.x)}
\]

\[\Rightarrow \text{check\_compatibility}(t.cm, t_{d}.cm) = false \text{ during consistency check for } \text{read}_{\text{td}}(t.x) \text{ (line 102 or 142)}
\]

\[\Rightarrow \text{read}_{\text{td}}(t.x) \text{ fails (due to lines 102-103 or 142-144), i.e., } \text{op} < \text{read}_{\text{td}}(t.x) \text{ is not possible in this case also.}
\]

Hence, after analysis of the both the cases, we conclude that \( \text{op} < \text{read}_{\text{td}}(t.x) \Rightarrow \text{check\_compatibility}(t.cm, t_{d}.cm) = true \) upon completion of \( \text{op} \).

\[\square\]

**Theorem 5.2.** Level wise transaction history of committed transactions \( \Pi(\hat{H}_{t_{\pi}}) \) produced at a node \( t_{\pi} \) follows 2PL for nested transactions, and satisfies level-wise opacity.

**Proof.** The proof follows from lemmas 5.2 to 5.6 that show that the execution of subtransactions in the transaction tree is based on the 2PL for nested transactions, and is linearizable (using the definition of linearization points). Thus, it satisfies level-wise opacity.
5.6.2 Proof for aborted transactions

Before diving into showing the correctness of aborted transactions, we prove that in case of an abort of a subtransaction, the objects locked by it are indeed released up to the respective ancestors from which they were read. To complete the scenario, we shall also show that the objects are released only up to the parent level in case of commit.

**Lemma 5.7.** When a subtransaction $t$ commits, it releases the locks only up to its parent. However, in case of its abort, it releases the locks on ancestors’ objects held by $t$ and its descendants all the way up to the respective level from which they were read.

**Proof.** The locks obtained by a transaction $t$ at an ancestor level, either through `search_parent` method or its `commit`, are recorded in $t.prs$ (lines 90, 120; 49). Proving that the locks are released only up to the parent level in case of commit is straightforward. In case of a commit, locks are released through `unlock_parent_locks` method (line 132). Observe that this method only unlocks objects at $t$’s parent level (line 37). It does not propagate to higher levels. Thus, in case of a commit, the locks are released only up to the parent level.

In case of an abort of $t$, it is $t$’s responsibility to release all the locks on its ancestors’ objects held by $t$ or its descendants. Such locks obtained by $t$ itself are tracked during `search_parent` (line 113-114) or during `try_to_merge` (lines 120, 49). The locks obtained on behalf of $t$’s descendants are logged during `search_parent` (lines...
113-114). Thus, $t$ is able to track all the locks held by it or its descendants on $t$’s ancestors’ objects.

Now, in case of $t$’s abort, *unlock_to_ancestors* method is invoked (line 56) to release the locks for objects in $t.prs$ at higher levels. Observe that *unlock_to_ancestors* is cascading in nature. Using the level information associated with objects in $t.prs$ (*pessimistic read set*), first lock at parent level is released (line 40), and then at higher level ancestors in a cascading manner (line 42) until the required level is reached (line 39, 41). This way, locks are released all the way up to the respective level from which an object was originally read in the chain, and not just up to the parent level.

\[ \square \]

To show the correctness of aborted transactions, we consider one aborted transaction $t_a$ at a time in the transaction tree and obtain its closure. The level-wise histories at different levels are obtained in the same fashion as done in Chapter 3 (refer to Section 3.7). However, we should show that the closure does not contain incompatible read operations.

Owing to the concurrent execution of transactions at the nested levels, it is quite possible that two active transactions in the transaction tree are mutually incompatible. We need to show that these two transactions are not part of $t_a$ or any of its ancestors in the closure for $t_a$.

**Lemma 5.8.** Let $\mathcal{H}$ denote the execution of the entire transaction tree in which $t_a$ is an aborted subtransaction whose last operation occurs at time $\tau^{t_a}$. Then, $t_a$ is compatible with other transactions in the closure $\mathcal{H}^{C_{t_a}}$ for $t_a$.

**Proof.** Following the definition of the closure $\mathcal{H}^{C_{t_a}}$ for $t_a$, let $\text{read}_{t_a}(t,x)$ be the last
operation of \( t_a \). Clearly, \( \text{read}_{t_a}(t.x) \) is the last operation in \( \mathcal{H}_{t_a} \).

Assume that \( t_1 \) and \( t_2 \) are two incompatible transactions belonging to \( \mathcal{H}_{t_a} \). Let \( P \) denote the set containing \( t_a \) as well as its ancestors, and \( S \) denote all the transactions whose steps are represented in \( \mathcal{H}_{t_a} \). Clearly, \( t_1, t_2 \in S \). In other words, \( t_1, t_2 \) either belong to \( P \) or have successfully merged with some transactions in \( P \). Also, now we have two cases.

Case I: \( t_1 \) and \( t_2 \) are two committed descendants that have merged with a node \( t \) in \( P \). This is not possible as at any level, two incompatible subtransactions are not allowed to merge with the parent (follows from Lemma 5.1). This implies that both \( t_1 \) and \( t_2 \) cannot be part of \( S \) (i.e., \( \neg(t_1 \in S \land t_2 \in S) \)). Hence, the contradiction.

Case II: \( t_1 \) is part of \( t_i \) and \( t_2 \) is part of \( t_j \), where \( t_i, t_j \) are two distinct transactions in \( P \).

Clearly, \( t_i \) and \( t_j \) have an ancestor-descendant relationship. For simplicity, let \( t_i \) be an ancestor of \( t_j \).

Now, if \( t_1 = t_i \), then \( t_1 \) can become incompatible with \( t_2 \) by performing an external read operation, say \( \text{r}_{t_1}(t.p.x) \), that is incompatible with the previous read steps of \( t_2 \). By construction of ParSTM, before completion of \( \text{r}_{t_1}(t.p.x) \), the incompatible descendant \( t_j \) (as it contains \( t_2 \)) is aborted (line 95). Now \( t_a \) clearly being a descendant of \( t_i \), it means the last operation of \( t_a \) (at time \( \tau_{t_a} \)) occurred before \( \text{r}_{t_1}(t.p.x) \). This in turn contradicts the fact that \( \text{r}_{t_1}(t.p.x) \) is part of \( \mathcal{H}_{t_a} \), as all the steps after \( \tau_{t_a} \) are discarded in \( \mathcal{H}_{t_a} \) (Section 3.7.4, step (i)).

Alternatively, \( t_i \) can become incompatible with \( t_j \) (containing \( t_2 \)) when \( \hat{t}_1 \) commits and merges with \( t_i \). Even in this case, by construction of ParSTM, \( t_j \) will be aborted before \( \hat{t}_1 \) commits (due to line 135). That means \( t_1 \notin S \). Hence, the contradiction.
Note that an incompatible descendant \( t_j \) (containing \( t_2 \)) cannot exist after becoming incompatible with its ancestor \( t_i \) (due to lines 85 or 135). That means all the steps of \( t_2 \) were consistent and completed before it became incompatible with \( t_i \).

Thus, we conclude that all the steps in \( H^{C_{ta}} \) are compatible.

\[ \square \]

**Theorem 5.3.** The history \( \Pi(H^{C_{ta}}) \) for an aborted transaction \( t_a \) satisfies level-wise opacity.

**Proof.** Using Lemma 5.5, it has been proved that \( H^{C_{ta}} \) consists of compatible steps only. Moreover, recall that by construction of a closure, transaction \( t_a \) and its active ancestors are transformed into committed transactions in \( H^{C_{ta}} \). That means, \( \Pi(H^{C_{ta}}) \) represents a history of committed transactions. Now the set of proofs for a history of committed transactions in Section 5.5.1 can be applied directly to \( \Pi(H^{C_{ta}}) \). This implies that \( \Pi(H^{C_{ta}}) \) too satisfies level-wise opacity. \[ \square \]
Chapter 6

HParSTM

6.1 Overview of HParSTM

In the previous chapter, we discussed ParSTM that exercised pessimistic concurrency control at the nested level. In this chapter we discuss HParSTM that leverages full concurrency by allowing all transactions in a transaction tree to execute concurrently. In simplest terms, this is achieved by replicating the optimistic concurrency control mechanism discussed for non-nested transactions in Section 2.1 at every level (node) of a transaction tree.

In other words, a control variable $ow$ (overwritten set) and lock-based local copy of objects, using sets $rs$ (read set) and $fbd$ (forbidden set) as control variables, are used at each level. Moreover, similar semantics are associated with the operations on these control variables. A (sub)transaction $t$’s local copy of object $x$, $t.x$, is accessible to $t$ as well as its descendants. When the descendants of $t$ access $t.x$, they add their ids to $t.x.rs$. Later, if $t.x$ is modified by $t$ or its children, then the ids of descendants present
in \( t.x.rs \) are added to \( t.ow \), followed by updating \( x.fbd \) using \( t.ow \), and clearing \( t.x.rs \) (similar to Protocol given Section 2.1: lines 17-18 in Table 2.1).

### 6.2 Discussion of contention management

#### 6.2.1 Standard cases

Most of the scenarios and approaches presented in this section have already been discussed in previous chapters. However, to set the context for HParSTM, we shall discuss the various cases in light of the protocol in this chapter.

#### 6.2.1.1 Consistency checking at the time of a read operation

Each transaction \( t \) uses \( mts \) (merged transaction set) to keep track of the descendants that have merged with \( t \). When a subtransaction \( t \) tries to read from an object \( t'.x \)
from its ancestor \( t' \)'s local space, it should ensure that none of the transactions in 
\( mts \) of \( t \) or any of its intermediate ancestors belongs to \( t'.x.fbd \).

**Example:**

\[
\mathcal{H}_1 = r_{t_{121}}(t_{12}.x), w_{t_{122}}(t_{12}.x), w_{t_{122}}(t_{12}.y)c_{122}, r_{t_{1212}}(t_{12}.y)\]

Referring to Figure 6.1a, \( t_{121} \) (\( t_{1212}'s \) ancestor) reads from \( t_{12}.x \), therefore \( t_{121} \in t_{12}.x.rs \). Later, \( t_{122} \) modifies \( t_{12}.x \) and \( t_{12}.y \), while merging with \( t_{12} \), resulting in \( t_{121} \in t_{12}.x.fbd, t_{12}.y.fbd \) and \( t_{12}.ow \). Later, \( t_{1212} \) tries to read \( t_{12}.y \). Now, although \( t_{1212} \not\in t_{12}.y.fbd \), it is prevented from reading \( t_{12}.y \) (which is \( t_{122}.y \)) as its intermediate ancestor \( t_{121} \) belongs to \( t_{12}.y.fbd \).

\[
\mathcal{H}_2 = r_{t_{1211}}(t_{12}.x)c_{1211}, w_{t_{122}}(t_{12}.x), w_{t_{122}}(t_{12}.y)c_{122}, r_{t_{1212}}(t_{12}.y)\]

Consider another example using Figure 6.1b. In \( \mathcal{H}_2 \), instead of \( t_{121} \) reading from \( t_{12}.x \), its child \( t_{1211} \) reads from \( t_{12}.x \) and merges with \( t_{121} \). In this case, when \( t_{1212} \) tries to read \( t_{12}.y \), it will notice \( t_{1211} \in t_{12}.y.fbd \). Now, as \( t_{1211} \) has already become part of \( t_{121} \), it is illegal for \( t_{1212} \) to access \( t_{12}.y \) and is consequently prevented.

### 6.2.1.2 Avoiding cyclic conflict through transitivity across levels

In the nesting of transactions, there is a possibility of cyclic conflict between transactions not only at the same level (between sibling transactions) but also between transactions at different levels (ancestors and descendants). The cyclic conflict between sibling subtransactions is automatically taken care of by the control variables at their parent’s level. The cyclic conflict between transactions at different levels is
handled by aborting a subtransaction as soon as it becomes incompatible with its ancestor.

**Example:**

\[ \mathcal{H}_3 = r_{1211}(t_{12}.x), w_{t_{12}}(t_{12}.x), w_{t_{12}}(t_{12}.y), r_{121}(t_{12}.y) ? \]

Refer to Figure 6.2a for history \( \mathcal{H}_3 \). Here, first \( t_{1211} \) reads from \( t_{12}.x \) \( \Rightarrow t_{1211} \in t_{12}.x.rs \). Later, \( t_{12} \) modifies \( t_{12}.x \) and \( t_{12}.y \) \( \Rightarrow t_{1211} \in t_{12}.x.fbd \) and \( t_{12}.y.fbd \). Next, \( t_{121} \) successfully reads from \( t_{12}.y \) (as \( t_{121} \notin t_{12}.y.fbd \)) and creates its own local copy \( t_{121}.y \). Observe that \( r_{t_{1211}}(t_{12}.x) \) is incompatible with \( r_{t_{121}}(t_{12}.y) \). Therefore, \( t_{121} \) immediately aborts \( t_{1211} \).
6.2.1.3 Keeping track of incompatible read operations

In case of ParSTM, the incompatible read operations are introduced when subtransactions read from the global objects which are operated through optimistic concurrency control. In case of HParSTM, this may happen while reading from any ancestor in the super tree. This is due to the fact that the objects at each level in HParSTM are operated through optimistic concurrency control. As discussed in Section 5.3, here also each transaction keeps track of its incompatible transactions using set $its$ (incompatible transaction set).

Example:

$\mathcal{H}_4 = r_{t_{1211}}(t_{12}.x), w_{t_{12}}(t_{12}.x), w_{t_{12}}(t_{12}.y), r_{t_{1212}}(t_{12}.y), (c_{1211}, c_{1212})$?

Consider the history $\mathcal{H}_4$ using Figure 6.1b. Here, $t_{1211} \in t_{12}.x.rs$. Next, $t_{12}$, on $w_{t_{12}}(t_{12}.x)$ and $w_{t_{12}}(t_{12}.y)$, adds $t_{1211}$ to $t_{12}.x.fbd, t_{12}.y.fbd$ and $t_{12}.ow$. Now, $t_{1212}$, during $r_{t_{1212}}(t_{12}.y))$, finds $t_{1211} \in t_{12}.y.fbd$.

Here $t_{1212}$ is incompatible with $t_{1211}$. Both $t_{1211}$ and $t_{1212}$ share the common ancestor $t_{121}$ under $t_{12}$, and the conflicting write operations of $t_{12}$ are sandwiched between the respective read operations of $t_{1211}$ and $t_{1212}$ on $t_{12}$’s objects. Therefore, $t_{1212}$ adds $t_{1211}$ to its $its$.

6.2.2 Special cases

In this section, we discuss the special cases presented by HParSTM as well as the solutions to address them.
6.2.2.1 Tracking overwrite at intermediate ancestor level

In ParSTM, when a subtransaction \( t \) reads an object \( x \) from its ancestor \( t_a \), the pessimistic locking scheme ensures that the value of \( x \) is not changed at each of its intermediate ancestors, including \( t_a \). The optimistic concurrency control in HParSTM offers no such guarantee. This poses a challenge for maintaining consistency. To illustrate the scenario, consider the following case.

Example:

\[ H_5 = r_{t_{1212}}(t_{12}.x), w_{t_{1211}}(t_{121}.x), w_{t_{1211}}(t_{121}.y)c_{1211}, r_{t_{1212}}(t_{121}.y)? \]

In history \( H_5 \), \( t_{1212} \) first reads \( t_{12}.x \). Later, \( t_{1211} \) modifies objects \( x \) and \( y \) in \( t_{121} \)'s (\( t_{1212} \)’s ancestor) local space, thus \( t_{1211} \) modifies the value of the object \( x \) previously read by \( t_{1212} \). Now, if \( t_{1212} \) is allowed to read the value of \( y \) written by \( t_{1211} \) in \( t_{121} \)'s local space, then a cyclic conflict is established between \( t_{1211} \) and \( t_{1212} \). As \( t_{1212} \) has read \( t_{121}.y \) written by \( t_{1211} \), the serial order should be \( t_{1211}, t_{1212} \). This means that \( t_{1212} \) should have read a value of object \( t_{121}.x \) written by \( t_{1211} \), instead of the value of \( t_{12}.x \), which is not true. Hence, the cycle.

Solution. When a subtransaction \( t \) reads an object \( t'.x \) from its ancestor \( t' \), besides adding its id to \( t'.x.rs \), it adds its id to read set \( t''.x.rs \) of the null-valued local object \( t''.x \) of each of the intermediate ancestors \( t'' \) as well. Now, since \( t_{1212} \) accessed \( t_{12}.x \) before \( t_{1211} \) merges with their common ancestor \( t_{121} \), it means \( t_{121}.x.rs \) already contained \( t_{1212} \) before \( t_{121}.x \) is modified. Hence, after modification of \( t_{121}.x \) and \( t_{121}.y \), we have \( t_{1212} \in t_{121}.fbd, t_{121}.y.fbd \) and \( t_{121}.ow \). This ensues that \( t_{1212} \) is
forbidden to access \( t_{121}.y \) or any other object \( t_{121}.z \) written along/after the write of \( t_{121}.x \).

### 6.2.2.2 Significance of vts

\( H_6 = r_{122}(t_{12}.x), c_{122}, r_{1211}(t_{12}.x)w_{12}(t_{12}.x), w_{12}(t_{12}.y), r_{1212}(t_{12}.y), (c_{121}, c_{1212}) \).  

Consider \( H_6 \) in which we extend \( H_4 \) by adding \( r_{122}(t_{12}.x), c_{122} \). Transaction \( t_{122} \) is a committed child of \( t_{12} \) and has read \( t_{12}.x \). Subtransaction \( t_{1211} \) also has read \( t_{12}.x \). Later, when \( t_{12} \) writes \( t_{12}.x \) and \( t_{12}.y \), both \( t_{122} \) and \( t_{1211} \) are added to \( t_{12}.x.fbd \) and \( t_{12}.y.fbd \). Next, \( t_{1212} \) reads \( t_{12}.y \). Here, \( t_{122} \) and \( t_{1212} \) are compatible, but not \( t_{1211} \) and \( t_{1212} \). Observe that during \( r_{1212}(t_{12}.y) \), \( t_{12}.y.fbd \) contains both \( t_{1211} \) and \( t_{122} \), but \( t_{1212} \) should add only \( t_{1211} \) (not \( t_{121} \)) to its its. Otherwise, \( t_{1212} \) will definitely be not able to merge with \( t_{12} \), even under a valid scenario.

This is where set vts comes handy. Note that \( t_{122} \) and \( t_{1211} \) belong to different subtrees under \( t_{12} \). Differently, set \( t_{121}.vts \) contains \( t_{1211} \) but not \( t_{122} \) for the same reason. Hence, \( t_{1212} \) uses \( t_{121}.vts \) during \( r_{1212}(t_{12}.y) \) to filter out \( t_{122} \) and add only \( t_{1211} \) to \( t_{1212}.cm.its \).
6.3 Protocol

Protocol 6.1: HParSTM

1. State of base object \( x \):
2. \( \text{val} \in V \)
3. \( \text{rs} \) and \( \text{fb}: \subset T \)

State of response object:
4. \( \text{res}_x(\text{value}, \text{level}, s) \):
5. \( \text{val} \in V \), set to value
6. \( \text{lvl} \in L \), set to level
7. \( s_{\text{its}} \subset T \), set to \( s \)

Helper methods:
8. Operation \( \text{check\_compatibility}(\text{cm}_{t_a}, \text{cm}_{t_d}) \):
9. return \((\text{cm}_{t_a}.\text{mts} \cap \text{cm}_{t_d}.\text{its} = \emptyset) \land (
\text{cm}_{t_a}.\text{its} \cap \text{cm}_{t_d}.\text{mts} = \emptyset)\);}
10. Operation \( \text{update\_cm}(\text{cm}, \text{s}_m, \text{s}_l) \):
11. \( \text{cm}.\text{mts} \leftarrow \text{cm}.\text{mts} \cup \text{s}_m \),
12. \( \text{cm}.\text{its} \leftarrow \text{cm}.\text{its} \cup \text{s}_l \),
13. State of local atomic object \( t_s.\text{cm} \):
14. \( \text{mts} \subset T \)
15. \( \text{its} \subset T \)
16. State of transaction \( t \):
17. \( \text{parent} \in T \), parent’s id \( (t_p) \)
18. \( \text{lvl} \in L \).
19. \( \text{lws}, \text{lws} \subset X \)
20. \( \text{lrs}, \text{lws} \subset X \)
21. \( \text{mrg} : \text{locks} \)
22. \( \text{mts}, \text{its}, \text{vts} \) and \( \text{ow} \subset T \)
23. Operation \( \text{begin}(t_p, \text{level}) \):
24. \( \text{t.parent} \leftarrow t_p \),
25. \( \text{t lvl} \leftarrow \text{level} \),
26. \( \text{t mts} \leftarrow \{t\} \),
27. Operation \( \text{invoke\_child}(t_c) \):
28. \( t_c.\text{begin}(t, t.lvl - 1) \),
29. Operation \( \text{unlock\_parent\_locks}(s) \):
30. for each \( x \in (s \cap t.p\text{l} s) \) do
31. \( t.p\text{l} s \leftarrow t.p\text{l} s \setminus \{x\} \),
32. unlock \( t.p.x \); end for
33. Operation \( \text{abort}(t) \):
34. unlock\_parent\_locks(t.p\text{l} s);
35. \( t.\text{abort}(\text{active\_desc}) \);
36. return \( \text{abort} \);
37. Operation \( \text{abort\_incompat\_desc}(s_i) \):
38. \( s_a \leftarrow s_i \cap t.vts \),
39. if \( (s_a = \emptyset) \) then return; end if
40. lock \( t.mrg \);
41. for each \( (t_c \in \text{active\_children}(t)) \) do
42. if \( (t_c.\text{cm}.\text{mts} \cap s_a \neq \emptyset) \) then
43. \( t_c.\text{abort} \);
44. else
45. \( t_c.\text{abort\_incompat\_desc}(s_a) \); end if
46. end for
47. unlock \( t.mrg \);
48. \( t.vts \leftarrow t.vts \setminus s_a \);
49. Operation \( \text{abort\_active\_desc}(c) \):
50. for each \( l \in \text{active\_children}(t) \) do
51. \( t.\text{force\_abort}() \); end for
52. Operation \( \text{force\_abort}(t) \):
53. for each \( l \in \text{active\_children}(t) \) do
54. \( t.\text{force\_abort}() \); end for
55. return \( \text{abort} \);
56. Operation \( \text{get\_local\_lock}(x) \):
57. lock \( t.x \);
58. Operation \( \text{get\_locks}(s, t_c) \):
59. for each \( x \in s \) do
60. \( \text{get\_local\_lock}(x, t_c) \);
61. \( t_c.p\text{l} s \leftarrow t_c.p\text{l} s \cup \{x\} \); end for
62. Operation \( \text{write}(x, v) \):
63. lock \( t.x \);
64. \( t.x.val \leftarrow v \);
65. \( t.o\text{w} \leftarrow t.o\text{w} \cup t.x.r\text{s} \);
66. \( t.x.f\text{bd} \leftarrow L.\text{ow} \);
67. \( t.x.r\text{s} \leftarrow \emptyset \);
68. unlock \( t.x \);
69. \( t.l\text{ws} \leftarrow t.l\text{ws} \cup \{x\} \);
70. Operation \( \text{search\_parent}(x, t_c, t_o, cm_d) \):
71. lock \( t.x \);
72. \( \text{cm} \leftarrow t.cm \);
73. \( t.x.r\text{s} \leftarrow t.x.r\text{s} \cup \{t_o\} \);
74. \( \text{if} (t.x.val \neq \text{null}) \) then
75. \( \text{if}(\neg \text{check\_compatibility}(\text{cm}, \text{cm}_d) \lor
\text{((t.x.f\text{bd} \cap \text{cm}_d.mts \neq \emptyset)) \) then
76. unlock \( t.x \);
77. return \( \text{null} \); end if
78. \( s_i \leftarrow t.x.f\text{bd} \cap t.vts \);
79. \( \text{res}_x \leftarrow (t.x.val, t.l\text{vl}, s_i) \);
80. unlock \( t.x \);
82. else
83. \( \text{update}\_\text{cm}(cm, cm_d.mts, cm_d.\text{its}) \);
84. \( t.vts \leftarrow t.vts \cup \{ o \} \);
85. \( res_x \leftarrow \text{search\_parent}_{t_p}(x, t.o, cm) \);
86. \( \text{unlock } t.x; \)
87. \( \text{end if} \)
88. \( \text{return } res_x; \)
89. Operation \( \text{read}_t(x) \):
90. \( \text{lock } t.mrg; \)
91. \( \text{lock } t.x; \)
92. \( res_x \leftarrow \emptyset; \)
93. \( \text{if } (t_x,x.val = \text{null}) \text{ then} \)
94. \( res_x \leftarrow \text{search\_parent}_{t_p}(x, t.t, t.cm) ; \)
95. \( \text{if } (res_x = \text{null}) \text{ then} \)
96. \( \text{abort}_t(); \text{ end if} \)
97. \( t.x.val \leftarrow res_x.val; \)
98. \( t.lrs \leftarrow t.lrs \cup \{ x \}; \)
99. \( \text{update\_cm}(t.cm, \emptyset, res_x.sits); \)
100. \( \text{end if} \)
101. \( \text{unlock } t.mrg; \)
102. \( v \leftarrow t.x.val; \)
103. \( \text{unlock } t.x; \)
104. \( \text{abort\_incompat\_desc}(res_x.sits); \)
105. \( \text{return } v; \)
106. Operation \( \text{try\_to\_merge}_t(t_e) : \)
107. \( \text{lock } t.mrg; \)
108. \( t.get\_locks(t_e.lrs \cup t_e.lws, t_e); \)
109. \( \text{if } (\neg \text{check\_compatibility}(t.cm, t_e.cm)) \lor \)
110. \( (t_e.lws \neq \emptyset \land (t_e.cm.mts \cup t.ow \neq \emptyset)) \text{ then} \)
111. \( \text{unlock } t.mrg; \)
112. \( t.e.abort(); \text{ end if} \)
113. \( \text{for each } x \in t_e.lws \text{ do} \)
114. \( t.x.val \leftarrow t_e.x.val; \text{ end for} \)
115. \( \text{for each } x \in t_e.lrs ; t.x.val = \text{null} \text{ do} \)
116. \( t.x.val \leftarrow t_e.x.val; \text{ end for} \)
117. \( t.ow \leftarrow t.ow \cup (\bigcup_{x \in t_e.lws} t.x.rs) ; \)
118. \( \text{for each } x \in t_e.lws \text{ do} \)
119. \( t.x.fbd \leftarrow t.ow; \)
120. \( t.x.rs \leftarrow \emptyset; \text{ end for} \)
121. \( t.lrs \leftarrow t.lrs \cup t_e.lrs; \)
122. \( t.lws \leftarrow t.lws \cup t_e.lws; \)
123. \( \text{update\_cm}(t.cm, t_e.cm.mts, t_e.cm.its); \)
124. \( \text{unlock } t.mrg; \)
125. \( t.e.un\_lock\_parent\_locks(t_e.pls); \)
126. \( \text{Operation } \text{try\_to\_commit}_t(); \)
127. \( t_e.\text{try\_to\_merge}(t); \)
128. \( t_e.\text{abort\_incompat\_desc}(t.cm.its); \)
129. \( \text{return } (\text{commit}); \)

**Protocol 6.2: HParSTM (Special case of root node, } t_p) **

130. Operation \( \text{search\_parent}_{t_p}(x, t_c, t_o, cm_d) : \)
131. \( \text{lock } t_p.x; \)
132. \( cm \leftarrow t_p.cm; \)
133. \( t.x.rs \leftarrow t.x.rs \cup \{ t_o \}; \)
134. \( \text{if } (t_p.x.val \neq \text{null}) \text{ then} \)
135. \( \text{if } (\neg \text{check\_compatibility}(cm, cm_d)) \lor \)
136. \( (t_p.x.fbd \cap cm_d.mts \neq \emptyset)) \text{ then} \)
137. \( \text{unlock } t_p.x; \)
138. \( \text{return } \text{null}; \text{ end if} \)
139. \( res_x \leftarrow \{ t_p.x.val, t_p.lvl, \emptyset \}; \)
140. \( \text{unlock } t_p.x; \)
141. \( \text{else} \)
142. \( s_m \leftarrow cm.mts \cap cm_d.mts; \)
143. \( t_p.vts \leftarrow t_p.vts \cup \{ t_o \}; \)
144. \( \text{lock } t_p.x; \)
145. \( \text{if } (t_p.x.fbd \cap s_m = \emptyset) \text{ then} \)
146. \( t_p.x.rs \leftarrow t_p.x.rs \cup \{ t_o \}; \)
147. \( s_i \leftarrow t_p.x.fbd \cap t_p.vts; \)
148. \( res_x \leftarrow \{ t_p.x.val, t_p.lvl, s_i \}; \)
149. \( \text{else} \)
150. \( res_x \leftarrow \text{null}; \text{ end if} \)
151. \( \text{unlock } t_p.x; \)
152. \( \text{unlock } t_p.x; \)
153. \( \text{end if} \)
154. \( \text{return } res_x; \)

The following methods are same as in Protocol 5.3.
6.3.1 Transaction state

A transaction $t$ begins with $\text{begin}(t_p, \text{level})$, where $t_p$ denotes the id of the parent of $t$. The $t_p$ is $t_\psi$ for a root-level transaction. The set $\text{lrs}$ (local read set) is used to record the objects read from the ancestors, whereas set $\text{lws}$ (local write set) is used to record its write steps. The access to each copy of a base object is protected by a lock. The set $\text{ow}$ is used to store the ids of those descendants of $t$ that have read a value from its locally shared objects whose values have been modified since the reading. The local variable $\text{parent}$ stores the reference to $t$’s parent $t_p$.

Each transaction maintains a consistency management object, $\text{cm}$, which consists of sets $\text{mts}$ and $\text{its}$. Set $\text{mts}$ (merged transaction set) contains the ids of $t$ as well as those descendants of $t$ whose results have been propagated to (merged with) $t$. A descendant $t'$ of $t$ is included in $t.mts$ only when $t'$ and all of its intermediate ancestors up to $t$ commit. The set $\text{its}$ (incompatible transaction set) denotes a set of transactions that $t$ is incompatible with and hence cannot be merged together. Further, a set $\text{pls}$ (parent lock set) is used to keep track of the locks obtained on parent’s objects. Further, lock $\text{mrg}$ is used for merge and read operations.

Observe that a transaction in HParSTM does not maintain set $\text{prs}$ as is done in ParSTM in Chapter 5. This is owing to the fact that an external read in HParSTM
does not result in pessimistic locking of ancestors at multiple levels. A transaction only needs to abort the locks up to its parent level (obtained in try_to_merge operation), even in case of its abort.

6.3.2 Working of HParSTM:

In HParSTM, the allocation of space for local copy of an object \( x \) in the local space of a transaction is automatically done whenever required. For a transaction \( t \), this typically happens when a transaction \( t \) or its descendant) tries to obtain a lock on a local copy \( t.x \) of \( t \), and \( t.x \) does not already exist. At the time of allocation of space for a local copy of object \( x \) with transaction \( t \), the initial state of \( t.x \) is: \( t.x.val = \text{null} \), \( t.x.rs = \emptyset \) and \( t.x.fbd = \emptyset \). Further, in the text, wherever a transaction is looking for a local copy of an object to read, we mean a non-null valued copy of that object. Several steps of the protocol are self-explanatory or similar to the ones already discussed in previous chapters. We describe only the salient features. The key procedures of the protocol are discussed as follows.

\[ \begin{align*}
\text{begin}_t(t_p, \text{level}), \ & \text{invoke_child}_t(t_c): \ \text{Self-explanatory.} \\
\text{unlock_parent_locks}_t: \ & \text{Set} \ pls \ \text{contains the ids of parent’s objects on which} \ t \ \text{currently holds the locks. Locks on these objects are released using the method.} \\
\text{abort}_t(): \ & \text{This method is invoked when a transaction} \ t \ \text{has to be aborted. Before aborting, transaction} \ t \ \text{releases all the locks on parent’s objects in its possession. Finally,} \ t \ \text{calls the} \ \text{abort()} \ \text{method of its active children (if any). Observe that, in comparison to ParSTM, there is no unlock_to_ancestors needed in this case, as there is no multi-level pessimistic locking on ancestors’ objects in case of HParSTM.}
\end{align*} \]
check_compatibility_t(cm_a, cm_d): Discussed earlier.

abort_incompat_desc_t(s_i): Discussed earlier.

abort_active_desc_t(): Discussed earlier.

get_local_lock_t(x): This method is used to lock object t.x.

get_locks_t(s, t_c): This method is invoked by the child t_c to obtain locks on parent t's objects in set s.

write_t(x, v): In order to perform a write operation in its local space, transaction t locks its local copy t.x, updates the value of x.val, adds x.rs to ow before clearing x.rs, and updates x.fbd using ow. Then, t unlocks t.x, followed by addition of x in t.lws.

read_t(x): Same as the one discussed in case of ParSTM in Chapter 5, except that pls is not updated in this case as the lock on parent’s object is released upon completion of the read operation.

search_parent_t(x, t_c, t_o, cm_d): This method is similar to the search_parent method discussed in Chapter 5. This method is invoked by the child transaction t_c on behalf of a descendant t_o (original descendant requiring to read value of nearest copy of x) to search for a local copy of x at its parent t’s level. First, t.x is locked, and t_o is added to t.x.rs. If t.x is not null-valued, then we ensure that none of the transactions in mts of t_o and its intermediate ancestors up to t_c (1) belongs to t.x.fbd and (2) is incompatible with t. Otherwise, t.x is unlocked and null is returned to indicate failure. If the consistency check is valid, then incompatible descendants, s_i, of t_o are obtained using t.x.fbd and t_c.vts, and value of res_x(t.x.val, t.lvl, s_i) is returned, followed by unlocking of t.x.

In case t.x is null valued, t_o is added to t.vts. Next, t forwards the search to
its parent after updating $cm_d$ (cumulative $cm$). Finally, the value obtained from
the parent level is returned after unlocking $t.x$. Unlike ParSTM, local object $t.x$ is
unlocked not only in case of failed search but also in case of a successful search.

$try\_to\_merge(t_c)$ : This method is invoked by the child transaction $t_c$ to merge its
local sets with the parent $t$. Access control between competing children is achieved
by using a $mrg$ lock. Only one child transaction can merge at a time. For each $x \in$
$(t_c.lrs \cup t_c.lws)$ a lock is obtained on object $t.x$. Next, for consistency, compatibility
of $t_c$ with $t$ as well as its membership in $t.ow$ is checked. If the consistency check of
$t_c$ with $t$ is successful, then for each object $x \in (t_c.lrs \cup t_c.lws)$, the value of $t.x$ is
updated using that of $t_c.x$. Set $t.ow$ is updated by merging the cumulative content
of $t.x.rs$ for each $x \in t_c.lws$. For each $x \in (t_c.lws)$, $t.x.fbd$ is updated with $t.ow$ and
$t.x.rs$ is reset. The sets $t_c.lrs, t_c.lws$ and $t_c.cm$ are merged with the corresponding
sets of the parent $t$. Finally, all the locks previously obtained by $t_c$ are released.

$try\_to\_commit(t)$ : If $t$ is a nested transaction, then it tries to merge with its
parent.

In case $t$ is a root-level (non-nested) transaction, then the behaviour of the valida-
tion process for $t$ is the similar to that proposed in [9] for a non-nested transaction,
as already discussed in Chapters 4 and 5. When the root transaction commits, the
objects in its local write set are modified globally, i.e., the change is reflected in the
globally shared copy of objects available with $t_\phi$. The root transaction first checks if
it has only read steps. If it is found to be $read\_only$, then the transaction commits
immediately. Otherwise, it locks all the objects in its sets $t.lrs$ and $t.lws$. Then, it
checks if any of the subtransactions in its set $t.mts$ belongs to the $t_\phi.ow$ set. If yes,
then it means that the consistency of the root transaction has been compromised, and
the transaction releases all the locks before aborting. Otherwise it updates the values of all the global objects in its write set, followed by updating \( t_\psi.ow \) and \( t_\psi.x.fbd \) for each \( x \) it writes. Finally, the root transaction releases all the locks and commits. Observe that, during the commit of a root transaction, there is no checking of incompatibility or update operations for the sets \( cm, lrs, lws \) of its parent.

### 6.3.3 About management of sets

The local objects/sets associated with a transaction (other than \( t_\psi \)) exist only during the lifespan of that transaction. The local memory allotted to that transaction is freed as soon as it completes. Only the globally shared objects/sets, associated with the fictitious transaction \( t_\psi \), exist throughout the run of the STM system. As such the size of their content may grow very large if not managed over the run of the STM system. We can adopt the following approach for suppressing the ids of transactions that have been aborted or committed. When \( t \) commits or aborts, we subtract set \( t.vts \) from the sets \( (t_p.*.rs, t_p.*.fbd \) and \( t_p.ow \) at the parent level.

### 6.3.4 About deadlock freedom

In general, a deadlock situation may arise between two descendants, say \( t_1 \) and \( t_2 \), of a node \( t \) if each descendant requests a lock on \( t \)'s object such that it is currently held by the other descendant. In other words, say \( t_1 \) and \( t_2 \) already hold lock on \( t.x \) and \( t.y \) respectively. Now, if \( t_1 \) and \( t_2 \) try to lock \( t.y \) and \( t.x \) respectively (notice the reverse order of objects), a deadlock situation will arise where each subtransaction will be waiting indefinitely for a lock held by the other. Observe that for a deadlock
situation to possibly occur, the participating transaction should try to lock at least two shared objects. Otherwise, deadlock is not possible.

Unlike ParSTM, in HParSTM the lock on an object is released soon after the external read on it by a descendant. That means when a transaction \( t \)'s object is locked by its descendant for an external read, that lock/descendant cannot be an accomplice in a deadlock situation. If more than one descendant requests the lock for the same object, then only one of them gets the lock while others wait for their chance until the lock is released at the completion of the current external read operation. The only exception is when a descendant tries to merge with its parent. At that time, it may try to obtain and hold lock on more than one object of the parent. Thus, deadlock may occur when more than one child tries to merge with the parent at the same time. We prevent this case by restricting only one child to merge with the parent by using \( mrg \) lock.

6.4 Consistency checking and linearization points at level \( t \)

6.4.1 Linearization points of events in a level-wise history

The level wise event history \( \widehat{H}_t \) comprises of the following events: (i) local read/write operations of \( t \), (ii) external reads of \( t \)'s descendants w.r.t. \( t \), (iii) external reads of \( t \) itself, (iv) write operations due to merging of \( t \)'s child \( t_c \), and (v) commits of \( t \)'s children.

Let \( \ell_{op} \) denote the linearization point of an event. Then, the linearization points
of the various events in the history are defined as follows:

i Local read/write operation of $t$

(a) $read_t(t.x) : \ell_{op}$ corresponds to the time when it unlocks $t.x$ (line 103 or 172)

(b) $write_t(t.x) : \ell_{op}$ corresponds to the time when $t$ updates $t.ow$ (line 68)

ii External read operation of $t$

(a) $read_t(t_a.x) : \ell_{op}$ corresponds to the time when $t.cm$ is updated (line 99)

iii External read of a descendant $t_d$ on $t$’s object or that of $t$’s ancestor $t_a$:

(a) $read_{t_d}(t.x) : \ell_{op}$ corresponds to the time when $t_d$ reads $t.cm$ for consistency checking (line 72 or 132)

(b) $read_{t_a}(t.a.x) : \ell_{op}$ corresponds to the time when $t_d$ reads $t.cm$ for consistency checking (line 72 or 132)

iv Write due to commit of child $t_c$

(a) $write_{t_c}(t.x) : \ell_{op}$ corresponds to the time when $t_c$ updates $t.x$ (line 114)

v Commit of child $t_c$

(a) $C_{t_c} : \ell_{op}$ corresponds to the time when $t.cm$ is updated (line 123)

6.4.2 Definition of linearization point of a transaction

The definition of linearization points of root nodes in HParSTM is similar to what is defined in Chapters 4 and 5. A comprehensive set of proofs has been furnished for the same as well in those chapters. In this chapter, we shall focus on the definition of linearization points for non-root nodes in HParSTM and provide a set of proofs
for the same. With this end in view, the linearization points for non-root nodes are defined as follows.

1. If $t$ is an update transaction that commits, its linearization point, $\ell_t$, lies at the time just after it updates the parent’s $t_p.ow$ and $t_p.cm$ (consistency management) object (line 123).

2. If $t$ is a read only committed transaction, then $\ell_t$ is placed at the earliest of (i) the time it reads $t_p.cm$ for its last successful external read operation (lines 72, 132), and (ii) the time just before $\hat{t}$ (any id in $t.mts$) is added to $t_p.ow$ (if it ever is) (lines 65, 117).

3. If a transaction $t$ aborts, $\ell_t$ is determined as if it were a read only transaction, i.e., $\ell_t$ lies at the earliest of (i) the time it reads $t_p.cm$ for its last successful external read operation (lines 72, 132), and (ii) the time just before $\hat{t}$ (any id in $t.mts$) is added to $t_p.ow$ (if it ever is) (lines 65, 117).

Observe that in HParSTM, the linearization point of each transaction has been defined w.r.t. access/update of consistency management set $t.cm$ and $t.ow$ instead of set $t.ow$ alone as was done in [9]. This is so due to the fact that in HParSTM the consistency checking is based on combination of checking the membership in sets $t.cm$ and $t.ow$. Checking $t.ow$ alone is not sufficient.
6.5 Proof

Owing to the construction of HParSTM, the local objects available with a transaction \( t \) are operated on the same way (by \( t \)’s children) as the global copies of objects. In Chapter 4, we have already outlined the proofs for the correctness for the history produced at the global level (\( \mathcal{H}_t^{\psi} \)). The same set of proofs can be applied here for the level-wise history produced at any node of the super tree. In other words, to draw a parallelism, at each level \( t \) the local objects available with node \( t \) can be treated as global copies of objects, and \( t \)’s children as root-level transactions. The additional element we need to account for in the proofs for HParSTM is the compatibility of transactions.

6.5.1 Proof for committed transactions

Recall that, in ParSTM, the consistency at each level was achieved using locks associated with objects in a pessimistic manner. HParSTM differs in the sense that the objects at each node are accessed in an optimistic manner. Further, Unlike ParSTM where sets \( fbd \) and \( ow \) have been used only at the global level \( \mathcal{H}_t^{\psi} \), HParSTM uses these sets at each level. For this reason, the proofs for \( HParSTM \) are slightly different as well.

For a given level-wise history \( \mathcal{H}_t \) of committed transactions produced by HParSTM, we need to prove:

1. \( \rightarrow_{\mathcal{H}_t^{\sigma}} \) is total order.

2. \( \rightarrow_{\mathcal{H}_t^{\sigma}} \subseteq \rightarrow_{\mathcal{H}_t^{\sigma}} \).
3. If \( t_1, t_2 \) are two incompatible transactions, then they cannot merge together.

4. \( t_w \xrightarrow{t \cdot x} r_f t_r \Rightarrow \# t_w \) such that \( (t_w \rightarrow H^o_t, t_w' \rightarrow H^o_t \land (w_{t'}(t.x) \in H_t)) \).

5. \( t_w \xrightarrow{t \cdot x} r_f t_r \Rightarrow t_w \rightarrow H^o_t \).

Proofs of (a) \( \rightarrow H^o_t \) is total order, and (b) \( \rightarrow H^o_t \subseteq \rightarrow H^o_t \) follows directly from the definition of linearization points discussed above.

Next, we prove the remaining parts.

**Lemma 6.1.** Let \( \widehat{H}_t \) be a level-wise history of HParSTM. Let \( t_1 \) and \( t_2 \) be any two distinct transactions \( \in \{t \cup \text{children}(t)\} \), and \( t' \) be an ancestor of \( t \). If \( r_{t_2}(t'.x) : \beta(\widehat{t}_1, t'.x.fbd, AL_{t_2}(t'.x, read_{t_2}(t'.x))) \), then we have (1) if \( t_1, t_2 \neq t \), then \( \neg(t_1 \in \Pi(\widehat{H}_t) \land t_2 \in \Pi(\widehat{H}_t)) \), or (2) \( t_1 = t \land t_2 \notin \Pi(\widehat{H}_t) \) or (3) \( t_2 = t \land t_1 \notin \Pi(\widehat{H}_t) \).

**Proof.** Since \( t_1 \) and \( t_2 \) are distinct transactions, either both the transactions are children of \( t \), or one of the two is \( t \), while the other one is a child of \( t \). First, we show that

\[
r_{t_2}(t'.x) : \beta(\widehat{t}_1, t'.x.fbd, AL_{t_2}(t'.x, read_{t_2}(t'.x))) \Rightarrow \widehat{t}_1 \in t_2.cm.its.
\]

\[
r_{t_2}(t'.x) : \beta(\widehat{t}_1, t'.x.fbd, AL_{t_2}(t'.x, read_{t_2}(t'.x))) \Rightarrow \beta(\widehat{t}_1, res_x.its, RL_{t_2}(t'.x, read_{t_2}(x)))
\]

By construction \( res_x.its \) is used to update \( t_2.cm.its \) before unlocking \( t_2.x \) (lines 79-80, 103).
Thus, we have

\[ RL_{\hat{t}_2}(t'x, read_{\hat{t}_2}(x)) <_{H_1} RL_{\hat{t}_2}(t_2x, read_{\hat{t}_2}(x)) \Rightarrow \beta(\hat{t}_1, \hat{t}_2.cm.its, RL_{\hat{t}_2}(\hat{t}_2,x, read_{\hat{t}_2}(x))) \]

(due to line 79-80, 147-148).

Finally, \( \hat{t}_1 \in \hat{t}_2.cm.its \land \hat{t}_2 \in t_2.cm.mts \Rightarrow \hat{t}_1 \in t_2.cm.its \) (due to merging, \( \hat{t}_2 \) is a descendant of \( t_2 \)).

Now, let us consider the first case in which \( t_1, t_2 \) are children of \( t \). We have to show \( t_1 \in \Pi(\hat{H}_t) \Rightarrow t_2 \notin \Pi(\hat{H}_t) \) and vice versa.

**Case I: \( t_1, t_2 \in \text{children}(t) \).**

The \textit{try.to.merge}(t_c) method is invoked by the child to merge its local sets with those of its parent. Synchronization of concurrent requests from the children is achieved by means of a special lock, called merge lock (denoted, \( t.mrg \)). That means only one child transaction can merge with \( t \) at a time. Now, we have following two subcases to consider:

**Case I(a): \( AL_{t_1}(t.mrg, ttc) <_{H_1} AL_{t_2}(t.mrg, ttc) \) (assuming \( t_1 \in \Pi(\hat{H}_t) \)).**

\[ AL_{t_1}(t.mrg, ttc) <_{H_1} AL_{t_2}(t.mrg, ttc) \Rightarrow RL_{t_1}(t.mrg, ttc) <_{H_1} AL_{t_2}(t.mrg, ttc) \Rightarrow \beta(\hat{t}_1, t.cm.mts, RL_{t_1}(t.mrg, ttc)) \] (due to line 123)

\[ \Rightarrow \beta(\hat{t}_1, t.cm.mts, AL_{t_2}(t.mrg, ttc)) \]

This means, when \( t_2 \) tries to merge with \( t \), it will discover that it is incompatible with \( t \) and consequently abort (due to line 109-112). Thus, \( t_1 \in \Pi(\hat{H}_t) \Rightarrow t_2 \notin \Pi(\hat{H}_t) \).
Case I(b): $AL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc)$ (assuming $t_2 \in \Pi(H_t)$).

$AL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc)$

$\Rightarrow RL_{t_2}(t.mrg, ttc) <_{H_t} AL_{t_1}(t.mrg, ttc) \Rightarrow \beta(\widehat{t_1}, t.cm.its, RL_{t_1}(t.mrg, ttc))$ (due to line 123).

$\Rightarrow \beta(\widehat{t_1}, t.cm.its, AL_{t_2}(t.mrg, ttc))$

In this case, $t_1$ aborts later on (due to line 109-112) while trying to merge with $t$.

We have $t_2 \in \Pi(H_t) \Rightarrow t_1 \notin \Pi(H_t)$.

Case II: Either $t_1 = t$, or $t_2 = t$.

Case II(a): $t_1 = t$.

$t_1 = t \Rightarrow t_1$ is the parent of $t_2$

$\Rightarrow t_1$ is an ancestor of each $\widehat{t_2} \in t_2.cm.mts$.

$\Rightarrow$ read operation $r_{\widehat{t_2}}(t'.x): \beta(\widehat{t_1}, t'.x.fbd, AL_{\widehat{t_1}}(t'.x, read_{\widehat{t_1}}(t'.x)))$ is not possible (failure due to lines 123, 75-78, 135-138, 95-96).

Case II(b): $t_2 = t$.

Similar to Case I(b). $t_2$, being the ancestor $t$, adds $\widehat{t_1}$ to $t.its$ anyway, leading to abort of $t_1$ (due to lines 79-80, 88; 147-148, 99, 104).

Thus, the conjunction of the cases I and II proves the lemma. \qed
Lemma 6.2. Given a level-wise history $\hat{H}_t$, let $t_1 \in \Pi(\hat{H}_t)$. Then,
$$\beta(t_1, t.ow, \tau) \Rightarrow \ell_{t_1} <_{H_t} \tau.$$ 

Proof. We have to show that the linearization point for a transaction cannot lie after
the time at which its id has been added to $t.ow$. There are two cases:

- If $t_1$ is a read only committed transaction, then $\ell_{t_1}$ is placed at the earliest of (i)
the time it reads $t.cm$ for its last successful external read operation, and (ii) the time
just before $\hat{t}_1$ (any id in $t_1.mts$) is added to $t.ow$ (if it ever is), which proves the lemma.

- If $t_1$ writes and commits, its linearization point $\ell_{t_1}$ is placed during $\text{try\_to\_commit}()$, 
while $t_1$ holds the locks of every object of $t$ that it has read. If $\hat{t}_1$ was in $t.ow$ before
it acquired all the locks, it would not commit (due to lines 110-112, 181-182). Let
us notice that $\hat{t}_1$ can be added to $t.ow$ only by $t$ or an update child transaction of $t$
holding a lock on a base object previously read by $\hat{t}_1$. As $t_1$ releases the locks just
before committing (lines 123, 125; 183, 188), it follows that $\ell_{t_1}$ occurs before the time
at which $\hat{t}_1$ is added to $t.ow$ (if it ever is), which again proves the lemma.

Similarly, we shall prove the next lemma. Recall that $\gamma_{inc}(t_1, t, \tau)$ has been defined
in Chapter 5 to mean that transaction $t_1$ becomes incompatible with transaction $t$ at
time $\tau$.

Lemma 6.3. Given a level-wise history $\hat{H}_t$, let $t_1 \in \Pi(\hat{H}_t)$. Then,
$$\gamma_{inc}(t_1, t, \tau) \Rightarrow \ell_{t_1} <_{H_t} \tau.$$ 

Proof. The proof is similar to that of the previous lemma.
We have to show that the linearization point for a transaction $t_1$ cannot lie after the time at which $t_1$ has become incompatible with $t$. There are two cases:

- If $t_1$ is a read only committed transaction, then $\ell_{t_1}$ is placed at the earliest of (i) the time it reads $t.cm$ for its last successful external read operation, and (ii) the time just before $\widehat{t}_1$ (any id in $t_1.mts$) is added to $t.ow$ (if it ever is). For a successful read operation $t_1$ must read $t.cm$ (line 72 or 132) before time $\tau$ at which we have $\gamma_{inc}(t_1, t, \tau)$. Otherwise, $t_1$ will fail the subsequent consistency check at line 75 or 135 and consequently abort, leading to an unsuccessful read operation. Thus, $\ell_{t_1}$ occurs before $\tau$, which proves the lemma.

- If $t_1$ writes and commits, its linearization point $\ell_{t_1}$ is placed during $try\_to\_commit()$, while $t_1$ holds the lock on $t.mrg$ lock. If $\widehat{t}_1.cm$ was incompatible with $t.cm$ before it acquired lock on $t.mrg$ (line 107), it would not commit (due to failure to pass the consistency check at line 109). Let us notice that $t.cm$ is updated either during an external read of $t$ (line 99 or 167) or by merging of its committed child (line 123). Further, while $t.mrg$ is locked by a child, $t$ cannot perform an external read operation as it must lock $t.mrg$ as well first (line 90). As $t_1$ releases the lock on $t.mrg$ just before committing (lines 123, 124), it follows that $\ell_{t_1}$ occurs before the time $\tau$ at which $\gamma_{inc}(t_1, t, \tau)$, which again proves the lemma.

\begin{align*}
\text{Lemma 6.4. } t_w \xrightarrow{t.x} r_f r_r \Rightarrow \not \exists t'_w \text{ such that } (t_w \rightarrow H^w_t t'_w \rightarrow H^w_t r_r) \land (w_{t'_w}(t.x) \in H_t).
\end{align*}

\textbf{Proof.} By contradiction, let us assume that there are transactions $t_w, t'_w$ and $t_r$ and an object $t.x$ such that:
\(-t_w \xrightarrow{t.x} r_f t_r\)

\(-w_{t'_w}(t.x)v' \in \mathcal{H}_t\)

\(-t_w \rightarrow_{\mathcal{H}_t} t'_w \rightarrow_{\mathcal{H}_t} t_r.\)

As both \(t_w\) and \(t'_w\) write \(t.x\) in shared memory, they have necessarily committed (a write in shared memory occurs only at lines 114, 185 (and 64 in case of fictitious child transaction) during the execution of \(\text{try\_to\_commit}\), i.e., \(t_w, t'_w \in \Pi(\mathcal{H}_t)\)). Moreover, their linearization points \(\ell_{t_w}\) and \(\ell_{t'_w}\) occur while they hold the lock on \(t.x\) (before committing), from which we have the following implications:

\[ t_w \rightarrow_{\mathcal{H}_t} t'_w \iff \ell_{t_w} <_{\mathcal{H}_t} \ell_{t'_w}, \]

\[ \ell_{t_w} <_{\mathcal{H}_t} \ell_{t'_w} \Rightarrow RL_{t_w}(t.x, ttc) <_{\mathcal{H}_t} AL_{t'_w}(t.x, ttc) \]

\[ \Rightarrow w_{t_w}(t.x)v <_{\mathcal{H}_t} w_{t'_w}(t.x)v', \]

\[ (t_w \xrightarrow{t.x} r_f t_r) \land (w_{t_w}(t.x)v <_{\mathcal{H}_t} w_{t'_w}(t.x)v') \Rightarrow w_{t_w}(t.x)v <_{\mathcal{H}_t} r_{t_r}(t.x)v <_{\mathcal{H}_t} w_{t'_w}(t.x)v'. \]

When a subtransaction in \(\widehat{t}_1\) (i.e., \(t_1\) or any of its descendants that merged with \(t_1\)) reads an object \(t.x\), it always adds its id to \(t.x.rs\) and to null-valued \(t'_x.rs\) of each of its intermediate ancestors \(t'\) (if any) upon acquiring a lock on \(t.x\) (lines 73, 133). Therefore, the predicate \(\beta(\widehat{t}_1, t.x.rs, RL_{\widehat{t}_1}(t.x, read(x)))\) is true \((t.x.rs\) is set to \(\emptyset\) only after being added to the set \(t.ow\)). Using this observation, we have the following:

\[ r_{t_r}(t.x)v <_{\mathcal{H}_t} w_{t'_w}(t.x)v' \land \beta(\widehat{t}_r, t.x.rs, RL_{\widehat{t}_r}(t.x, read(x))) \]

\[ \Rightarrow \beta(\widehat{t}_r, t.x.rs, AL_{t'_w}(t.x, ttc)), \]

\[ \beta(\widehat{t}_r, t.x.rs, AL_{t'_w}(t.x, ttc)) \land (w_{t'_w}(t.x)v' \in \mathcal{H}_t) \Rightarrow \beta(\widehat{t}_r, t.ow, \ell_{t'_w}) \Rightarrow \ell_{t_r} <_{\mathcal{H}_t} \ell_{t'_w} \iff t_r \rightarrow_{\mathcal{H}_t} t'_w. \]

which proves that, contrary to the initial assumption, \(t'_w\) cannot precede \(t_r\) in the sequential transaction history \(\mathcal{H}_t\).
Similar to ParSTM discussed in Chapter 5, we also need to show that in HParSTM the value read by a descendant from its ancestor is consistent at the time of reading. Recall that a descendant \( t_d \) can become incompatible with an ancestor \( t \) either due to an external read of \( t \) or a commit of a \( t \)'s child. We shall show that the consistency checking of the value read by \( t_d \) from \( t \) is guaranteed to be correct.

**Lemma 6.5.** Let \( t_d \) be a descendant of \( t \) such that history \( op_t, read_{t_d}(t.x) \in \hat{\mathcal{H}}_t \), where \( op_t \) denotes an external read operation or a commit of \( t \)'s child at time \( \tau \). Then, we show that (i) \( op_t, read_{t_d}(t.x) \) can be ordered, and (ii) \( op_t < read_{t_d}(t.x) \Rightarrow \neg \gamma_{inc}(t_d, t, \tau) \), i.e., \( t \) and \( t_d \) are not inconsistent before time \( \tau \).

**Proof.** By definition of linearization points for \( op_t \) and \( read_{t_d}(t.x) \) in \( \hat{\mathcal{H}}_t \) w.r.t. access to \( t.cm \), and \( t.cm \) being an atomic variable, it follows \( op_t, read_{t_d}(t.x) \) in \( \hat{\mathcal{H}}_t \) are ordered according to their linearization points.

To prove part (ii), let us assume by contrast that \( \gamma_{inc}(t_d, t, \tau) \), i.e., at the completion of \( op_t \) at time \( \tau \), \( check\_compatibility(t.cm, t_d.cm) = false \).

Now, for \( op_t < read_{t_d}(t.x) \), we have the following two cases:

*Case I: \( op_t \) is an external read operation of \( t \)

By definition of linearization points for events in a level wise history (Section 6.1), we have

\[ l_{op_t} < l_{read_{t_d}(t.x)} \]

⇒ update of \( t.cm \) (line 99) such that \( check\_compatibility(t.cm, t_d.cm) = false \) occurs before \( t.cm \) is read (line 72 or 132) for consistency check of \( t_d \). (Recall that \( t.cm \) is kept in an atomic variable.)

\[ \tau < l_{read_{t_d}(t.x)} \] such that \( \gamma_{inc}(t_d, t, \tau) = true \)
⇒ check\_compatibility(t.cm, t.d.cm) = false during the consistency check for read_{t.d}(t.x) (line 75 or 135)

⇒ read_{t.d}(t.x) fails (due to lines 75-78, 135-138, 95-96), i.e., op_t < read_{t.d}(t.x) is not possible in this case.

**Case II: op_t is a commit of t’s child**

By definition of linearization points for events in a level wise history (Section 5.5.3), we have

\[ \ell_{op_t} < \ell_{read_{t.d}(t.x)} \]

⇒ update of t.cm (line 123) such that check\_compatibility(t.cm, t.d.cm) = false occurs before t.cm is read (line 72 or 132) for consistency check of read_{t.d}(t.x) (Recall that t.cm is kept in an atomic variable).

\[ \tau < \ell_{read_{t.d}(t.x)} \] such that \(\gamma_{inc}(t.d, t, \tau) = true\)

⇒ check\_compatibility(t.cm, t.d.cm) = false during consistency check for read_{t.d}(t.x) (line 75 or 135)

⇒ read_{t.d}(t.x) fails (due to lines 75-78, 135-138, 95-96), i.e., op_t < read_{t.d}(t.x) is not possible in this case also.

Hence, after analysis of both the cases, we conclude that \(\neg \gamma_{inc}(t.d, t, \tau)\) upon completion of op_t.

\[ \square \]

**Lemma 6.6.** \( t_w \xrightarrow{t.x} rf t_r \Rightarrow t_w \rightarrow_{H^o_t} t_r. \)

**Proof.** The proof is made up of two parts. First it is shown that \( t_w \xrightarrow{t.x} rf t_r \Rightarrow \neg \beta(\hat{t}_r, t.ow, \ell_{t.w}) \), and then it is shown that \( \neg \beta(\hat{t}_r, t.ow, \ell_{t.w}) \land t_w \xrightarrow{t.x} rf t_r \Rightarrow t_w \rightarrow_{H^o_t} t_r. \)
Proof of $t_w \xrightarrow{t.x} r_f \ t_r \Rightarrow \neg \beta(t_r, t.ow, \ell_{t_w})$. Let us assume by contradiction that the predicate $\beta(\widehat{t}_r, t.ow, \ell_{t_w})$ is true. Due to lines 66, 119 (or 186) we have $\beta(\widehat{t}_r, t.ow, \ell_{t_w}) \Rightarrow \beta(\widehat{t}_r, t.x.fbd, RL_{t_w}(t.x, ttc))$

If the read of $t.x$ from shared memory by $t_r$ is before the write by $t_w$, we cannot have $t_w \xrightarrow{t.x} r_f t_r$. So, in the following we consider that the read of $t.x$ from shared memory by $t_r$ is after its write by $t_w$. We have then $RL_{t_w}(t.x, ttc) <_{H_t} AL_{t_r}(t.x, read(x))$, and consequently $\beta(\widehat{t}_r, t.x.fbd, RL_{t_w}(t.x, ttc)) \Rightarrow \beta(\widehat{t}_r, t.x.fbd, AL_{t_r}(t.x, ttc))$.

As $\widehat{t}_r \in t.x.fbd$ when it locks $t.x$, it follows that the read operation fails at line 75-76, 135-136 (or 145) and consequently we cannot have $t_w \xrightarrow{t.x} r_f t_r$. Summarizing the previous reasoning we have $\beta(\widehat{t}_r, t.ow, \ell_{t_w}) \Rightarrow \neg (t_w \xrightarrow{t.x} r_f t_r)$, and taking the contrapositive we finally obtain $t_w \xrightarrow{t.x} r_f t_r \Rightarrow \neg \beta(\widehat{t}_r, t.ow, \ell_{t_w})$

Proof of $\neg \beta(\widehat{t}_r, t.ow, \ell_{t_w}) \land t_w \xrightarrow{t.x} r_f t_r \Rightarrow t_w \rightarrow_{H_t} t_r$. As defined earlier, the linearization point $\ell_{t_r}$ depends on whether $t_r$ is a read only or an update transaction. The proof considers the two possible cases.

- If $t_r$ is an update transaction that commits, its linearization point $\ell_{t_r}$ (that is defined at line 123 after it updates the set $t.ow$ and $t.cm$) occurs while merging (try_to_commit()). Due to this observation, the fact that $t_w$ releases its locks after its linearization point, and $t_w \xrightarrow{t.x} r_f t_r$, we have $\ell_{t_w} <_{H_t} \ell_{t_r}$, i.e., $t_w \rightarrow_{H_t} t_r$.

- If $t_r$ is a read only transaction that commits, its linearization point $\ell_{t_r}$ is placed just before the earliest time at which it is added to $t.ow$ (lines 65, 117), or at the time it accesses $t.cm$ for its read operation (lines 72 or 132). In the latter case, we have $w_{t_w}(t.x)v <_{H_t} \ell_{t_w} <_{H_t} RL_{t_w}(t.x, ttc) <_{H_t} AL_{t_r}(t.x, read(x)) <_{H_t} r_{t_r}(t.x)v <_{H_t} \ell_{t_r}$ from which we have $\ell_{t_w} <_{H_t} \ell_{t_r}$, i.e., $t_w \rightarrow_{H_t} t_r$. Hence, in all cases, we have $t_w \xrightarrow{t.x} r_f$
Theorem 6.1. Every level-wise history of committed transactions, \( \Pi(\widehat{H}^\sigma_t) \), produced by HParSTM satisfies the level-wise opacity consistency criterion.

Proof. The proof follows from the definition of linearization points, and Lemmas 6.1 through 6.4.

6.5.2 Proof for aborted transactions

The correctness of aborted transaction is established in the same way as done in Chapter 5. For each aborted transaction \( t_a \) we consider its closure \( \widehat{H}^{c_{t_a}} \) (Section 3.7.4). Observe that \( \widehat{H}^{c_{t_a}} \) represents a history of committed transactions. Next, using \( \widehat{H}^{c_{t_a}} \), we construct the level history at each of the ancestors of \( t_a \) and prove the consistency at each level, as done for the history of committed transactions.
Chapter 7

MxSTM

7.1 The main idea

The idea here is to have nodes employing different (optimistic/ pessimistic) concurrency control mechanism in the transaction tree (super tree) for STM. There are two types of nodes: \textit{p-nodes} and \textit{o-nodes}. The local objects of a \textit{p-node} are operated on in a pessimistic manner. On the other hand, the local objects of an \textit{o-node} are operated on in an optimistic manner. Thus, on the basis of the management of the locks, we have two types of objects in the picture.

7.1.1 About nesting of transactions

In the nesting of transactions, a node can be a \textit{p-node} or an \textit{o-node}. Depending upon the type of node (\textit{p-type} or \textit{o-type}), we get different behaviours (pessimistic or optimistic) at that node. Also, depending upon the various combinations of \textit{p-type} and \textit{o-type} nodes, we can obtain different degrees of concurrency in a subtree. For
example, consider node $t_{13}$ in Figure 7.1. Node $t_{13}$ is an o-node. Two of its children, $t_{131}$ and $t_{133}$, are p-nodes, whereas the other child, $t_{132}$, is an o-node. Also consider the path from $t_1$ to $t_{1121}$. Nodes $t_1$ and $t_{1121}$ are o-nodes, whereas the intermediate nodes, $t_{11}$ and $t_{112}$, are p-nodes.

### 7.1.1.1 Behaviour of a p-node

As stated earlier, a p-node denotes a transaction which employs a pessimistic approach to concurrency control. More precisely, when an object $x$ (denoted as $t_p.x$) of a p-node, $t_p$, is locked by its child transaction $t_c$, then $t_c$ retains the ownership of that lock until $t_c$ completes its execution (commit/abort). That means, $t_p$ or any other child of $t_p$ wanting a lock on $t_p.x$ now has to wait until $t_c$ releases the lock on $t_p.x$ upon its completion. However, it should be noted that if it is a local operation of $t_p$, then it releases the lock on its local object after its read/write operations. This way, the
pessimistic approach offers limited concurrency. Since a child may try to lock more than one objects at a time during its normal operation, there is a possibility of a deadlock situation between the children. Hence, a p-node maintains a wait-for graph, whose access is controlled by a lock, to detect and resolve this deadlock situation between its children.

7.1.1.2 Behaviour of an o-node

An o-node denotes a transaction that applies optimistic concurrency control mechanism. Let \( t_p \) be an o-node, and \( t_c \) be one of its children. Here, when \( t_c \) acquires a lock on \( t_p.x \) for its read operation, it does not retain the lock throughout its lifetime, rather it releases the lock immediately after the termination of its operation. Thus, \( t_p.x \) becomes available for use by \( t_p \) or its children. Hence, there is greater degree of concurrency offered by an o-node.

7.2 Design challenge

In previous chapters, we already designed the protocols for emulating the behaviours of p-node and o-node. We shall denote an o-node by \( t_\omega \), and a p-node by \( t_\pi \). Each of the nodes in HParSTM is an o-node, and the (non-root) nodes in ParSTM is a p-node. Hence, the job here is to integrate the two protocols, ParSTM and HParSTM, to obtain the desired result in a correct fashion.

As discussed before, the set of ancestors of a node \( t \) in the transaction tree can be a combination of o-nodes and p-nodes. This introduces a challenge of ensuring that the locks of p-nodes and o-nodes are managed in the right fashion, especially while
(a) Duplicate reads from same level

(b) Duplicate reads from different levels

Figure 7.2: Duplicate reads (SP: search parent)

reading from the ancestors.

### 7.2.1 Handling special cases for MxSTM

#### 7.2.1.1 Issue of duplicate request at a p-node

In MxSTM, the interleaving of o-nodes and p-nodes introduces a unique case (challenge), not faced in any of the previous protocols. If a p-node, \( t_{\pi 1} \), has an o-node,
Figure 7.3: Handling release of locks in case of abort of a transaction with duplicate reads (SP: search parent; UTA: unlock to ancestors)
$t_{\omega 11}$, as one of its children, then $t_{\pi 1}$ may get more than one request for reading same object through $t_{\omega 11}$ on behalf of $t_{\omega 11}$’s descendants.

To explain the case, let us refer to Figure 7.2. In Figure 7.2a, $t_{\pi 1}$ and $t_{\pi 111}$ are p-nodes whereas rest of the nodes are o-nodes. Let us say that the value of an object $x$ is only available with $t_{\pi 1}$.

Step 1: Now, $t_{\pi 111}$ reads object $t_{\pi 1}.x$ through propagation of search_parent request through its ancestors $t_{\omega 11}$ and $t_{\pi 1}$ in order. Recall that $t_{\pi 1}$ being a p-node, its objects are accessed in pessimistic manner, i.e., $t_{\omega 11}$ retains the lock on its parent’s object $t_{\pi 1}.x$. However, the lock on the object of the o-node $t_{\omega 11}.x$ is released after the operation.

Step 2: Now, suppose $t_{\omega 112}$ wants to read $x$ and invokes search_parent procedure. The request propagates up to $t_{\omega 11}$ (because $t_{\omega 11}.x$ was unlocked after completion of external read by $t_{\pi 111}$ and is available for locking again). Observe that $t_{\omega 11}$ already holds the lock on $t_{\pi 1}.x$ owing to previous external read operation on $t_{\pi 1}.x$. At this point the question is how should the second search_parent request for $x$ from $t_{\omega 11}$ be handled at $t_{\pi 1}$?

As $t_{\omega 11}$ already holds the lock on $t_{\pi 1}.x$, it is logical to argue that $t_{\omega 11}$ should forward the request search_parent to $t_{\pi 1}$ and should return the value of $t_{\pi 1}.x$, without trying to lock it again. However, this behavior is something new and not witnessed in ParSTM. In ParSTM, the object of a p-node could be read by only one transaction at a time. In this case, the value of object $t_{\pi 1}.x$ is read by $t_{\pi 111}$ and $t_{\omega 112}$ at the same time. In other words, $t_{\pi 1}.x$ has been read more than once by its descendant $t_{\omega 11}$. In Figure 7.2a,
both \( t_{\pi 111} \) and \( t_{\omega 112} \) read from the same level \( t_{\pi 1} \). However, it is possible that the two duplicate reads obtain values from different levels, as shown in Figure 7.2b. This can happen when the value becomes available later at an intermediate o-node ancestor, owing to update of \( x \) due to its local write or merging of its child.

Now, let us refer to Figure 7.3 and take the case of an abort of one of the transactions that performed the duplicate reads earlier. In Figure 7.3a transactions \( t_{\pi 1111} \) and \( t_{\omega 1112} \) perform duplicate reads on \( t_{\omega 1}.x \) through \( t_{\omega 111} \). Next, in Figure 7.3b, one of the two transactions, say \( t_{\pi 1111} \), aborts. In ParSTM, when a subtransaction aborts, it releases the lock up to the original ancestor from which it read the value. This behavior creates a problem here. Transaction \( t_{\omega 111} \) should not release the lock on \( t_{\pi 11}.x \) as it is read by another descendant \( t_{\omega 1112} \) which is still active. The lock on \( t_{\pi 11}.x \) should be released only in one of the following three cases:

a. Both \( t_{\pi 1111} \) and \( t_{\omega 1112} \) abort.

b. Transaction \( t_{\omega 111} \) commits.

c. Transaction \( t_{\omega 111} \) aborts.

### 7.2.1.2 Solution

Each node (o-node as well as p-node) maintains a data structure called \( prs \) (pessimistic read set) containing objects \( x_{pr} \) that has a field \( trc \) (total read count) to keep track of total number of pessimistic reads, and a dictionary \( lcs \) (level count set) of \( \langle lvl, rc \rangle \) key-value pairs, where \( lvl \) is the level of the farthest pessimistic node through which object \( x \) was read, and \( rc \) is the number of reads of \( x \) from that level. For the sake of simplicity, we shall denote an entry \( \langle lvl, rc \rangle \) in \( t.prs.x_{pr}.lcs \) corresponding to level \( l \) such that \( l = lvl \) as \( t.prs.x_{pr}(l) \).
Further, each object $x_{pr}$ in $prs$ is individually lockable. Thus, the locks on two different objects, say $x_{pr}$ and $y_{pr}$, can be concurrently obtained.

**Working:**

While reading a value of $x$ by a descendant $t_d$ from an ancestor $t_a$ if there happens to be a $p$-node (if any), $t_\pi$, in the path from $t_a$ to $t_d$ (top to bottom order), then $t.prs.x_{pr}.trc$ is incremented by 1 at each descendant $t$ of $t_\pi$ in the path from $t_\pi$ to $t_d$ (excluding $t_\pi$). Conversely, when $t_d$ aborts, then $t.prs.x_{pr}.trc$ decremented by 1 at each intermediate node in the path from $t_d$ to $t_\pi$ (excluding $t_\pi$) and if $t.prs.x_{pr}.trc$ equals 0 and $t$ holds the lock on its parent’s object $t_p.x$, then $t$ releases the lock on $t_p.x$.

### 7.2.2 Comparing MxSTM with ParSTM and HParSTM

The methods in HParSTM and ParSTM were carefully designed so that they can be integrated to form MxSTM, with the smallest number of changes in the individual methods. The idea was to keep the code modular, easy to understand and reusable. A comparative study of MxSTM against HParSTM and ParSTM reveals that most of the code from HParSTM and ParSTM has been reused as is. However, a few adjustments had to be made in order to handle the special cases unique to MxSTM.

#### 7.2.2.1 Changes w.r.t. both ParSTM and HParSTM

$res_x$: Object $res_x$ has been updated to include two additional boolean fields, namely $is\_pread$ (Is Pessimistic Read) and $is\_plocked$ (Is Parent object Locked). Thus, $res_x$ has following structure:
\[ \text{res}_x: \{ \]
\begin{align*}
&\text{i} \quad v & \text{val} \in V \\
&\text{ii} \quad p & \text{lvl} \in L \\
&\text{iii} \quad p & \text{its} \in T \\
&\text{iv} \quad is\_pread \in \text{boolean} \\
&\text{v} \quad is\_plocked \in \text{boolean} \\
\} \\
\]

\text{read}_t(x): \text{Read method has been updated to include tracking of external read through a pessimistic ancestor (if any).}

\text{prs: In ParSTM (Chapter 5), prs was defined as a set containing \langle x, \text{level} \rangle pairs. Here, prs is extended to contain more complex objects } x_{pr}, \text{ as discussed in Section 7.2.1.2.}

\text{unlock\_to\_ancestors: This method too has been updated to take into consideration the external ‘pessimistic read count’ of an object before unlocking it at the parent level (if needed).}

\textbf{7.2.2.2 Specific changes w.r.t. HParSTM}

\text{search\_parent: This method has been updated to handle duplicate requests of its descendants that propagate to its ancestor that is a } p\text{-node, by tracking their pessimistic read counts at its level. Other methods are the same as in HParSTM.}

\text{unlock\_to\_ancestors: Unlike HParSTM, an } o\text{-node } t_\omega \text{ here has to participate in the release of locks up to higher level ancestors, in case some transaction in } subTree(t_\omega), \text{ through their external read operations, obtained pessimistic locks on objects of } t_\omega\text{'s pessimistic ancestors.}

\text{abort: Similarly, abort involves releasing the locks not only to the parent level, but also up to higher level ancestors, if needed.}
7.2.2.3 Specific changes w.r.t. ParSTM

**search_parent**: This method has been updated similarly as done for HParSTM. Other methods are the same as in ParSTM.

**vts**: In MxSTM, not only the root node but also the intermediate ancestors can employ optimistic concurrency control (*o-nodes*). As such, *vts* of a child *p-node* can be used by its parent (an *o-node*) to update its.

### 7.2.2.4 New methods

We introduce the following methods in MxSTM to manage the external read count on ancestors object.

**trc_incre (Total read count Increment)**: To increment the total read count of external pessimistic read using a lock.

**trc_decre (Total read count Decrement)**: To decrement the total read count of external pessimistic read using a lock.

**prc_incre (Pessimistic read count Increment)**: To increment the external pessimistic read count of an object from a level.

**prc_decre (Pessimistic read count Decrement)**: To decrement the external pessimistic read count of an object at a level and release the lock on the parent’s object accordingly.
7.3 Protocol

7.3.1 Pseudocode

Protocol 7.1: MxSTM

Common to both o-node and p-node:

t denotes tπ/tω.

1. State of pessimistic read object xpr:
2. trc ∈ N, default 0
3. lcs ∈ L × N:
4. lvl ∈ L
5. rc ∈ N, default 0
6. Operation diff_lrc(x1pr, x2pr, l2):
7. for each ((l, n) ∈ ∪i≥2x2pr,lcs) do
8. x1pr(l).rc ← x1pr(l).rc − n;
9. x1pr.trc ← x1pr.trc − n; end for
10. x1pr.lcs ← x1pr.lcs \ {∪rc=0x1pr.lcs};

11. State of transaction t∗:
12. parent ∈ T, parent’s id (tπ)
13. type ∈ { o-node, p-node }
14. lrslws C X
15. cmmts, lsts and vs C T
16. plsls C X
17. mrg : lock
18. prs C Xpr : individual items lockable

19. Operation begin∗(tp, level, type):
20. tπ.parent ← tπ;
21. tπ.lvl ← level;
22. tπ.cm.mts ← {tπ};
23. tπ.type ← type;
24. Operation invoke_child∗(tπ, type):
25. begin∗(tπ, tπ.lvl − 1, type);
26. Operation unlock_parent_locks∗(s):
27. for each x ∈ (s ∩ tπ.pls) do
28. unlock tπ.x;
29. tπ.pls ← tπ.pls \ {x}; end for
30. Operation unlock_to_ancestors∗(s):
31. sanc ← ∪xpr.trc>0∧MaxLevel(xpr)>tp.lvl8
32. tπ.prccore(s);
33. if (sanc ≠ ϕ) then
34. tπ.unlock_to_ancestors(sanc); end if
35. Operation trc_incre∗(x):
36. lock tπ.prs.xpr;
37. tπ.prs.xpr.trc ← tπ.prs.xpr.trc + 1;
38. unlock tπ.prs.xpr;
39. Operation trc_decre∗(x):
40. lock tπ.prs.xpr;
41. tπ.prs.xpr.trc ← tπ.prs.xpr.trc − 1;
42. if (tπ.prs.xpr.trc = 0)
43. ∧x ∈ tπ.pls) then
44. unlock tπ.x;
45. tπ.pls ← tπ.pls \ {x}; end if
46. unlock tπ.prs.xpr;
47. Operation prc_incre∗(x, l):
48. lock tπ.prs.xpr;
49. tπ.prs.xpr(l).rc ← tπ.prs.xpr(l).rc + 1;
50. unlock tπ.prs.xpr;
51. Operation prc_decre∗(sprs):
52. for each (xpr ∈ sprs) do
53. lock tπ.prs.xpr;
54. diff_lrc(tπ.prs.xpr, xpr, tπ.lvl);
55. if (tπ.prs.xpr.trc = 0)
56. ∧x ∈ tπ.pls) then
57. unlock tπ.x;
58. tπ.pls ← tπ.pls \ {x}; end if
59. unlock tπ.prs.xpr;
60. end for
61. Operation get_lock∗(s, tπ):
62. for each x ∈ s do
63. tπ.get_local_lock(x, tπ);
64. tπ.pls ← tπ.pls \ {x}; end for
65. Operation abort∗(tπ):
66. tπ.abort_active_desc();
67. tπ.unlock_to_ancestors(tπ.prs);
68. tπ.unlock_parent_locks(tπ.pls);
69. return (abort);
70. Operation abort_active_desc∗():
71. for each t′ ∈ activeChildren(tπ) do
72. t′.force_abort(); end for
73. Operation force_abort∗():
74. for each t′ ∈ activeChildren(tπ) do
75. return `force_abort();` end for
76. return (abort);
77. Operation `abort_incompat_desc(s_i);`:
78. `s_a ← s_i ∩ t_v.t_s;`
79. if (s_a = ∅) then return; end if
80. for each (t_e ∈ activeChildren(t_s)) do
81. if (t_e.concat = `cm.mts ∩ s_a = ∅`) then
82. t_e.abort();
83. else
84. t_e.abort_incompat_desc(s_a);
85. end if
86. end for
87. t_v.t_s ← t_v.t_s \ s_a;
88. Operation `read_a(x);`:
89. t_v.get_local_lock(mrg, t_v);
90. t_v.get_local_lock(t_v, t_x);
91. res_a ← ∅;
92. if (t_x.val = null) then
93. ∗ t.i.incre(x);
94. else
95. res_a ← search_parent_{t}(x, t_s, t_x, t_x.concat, t_v); return if (res_a = null) then
96. ∗ t.i.decre(x);
97. if (res_a is locked) then
98. abort_t(); end if
99. t_x.val ← res_a.val;
100. t_x.lrs ← t_x.lrs ∪ {x};
101. update_cm(t_x.cm, ∅, res_a.s_its);
102. if (¬f ∧ res_a.is_locked) then
103. t_x.pls ← t_x.pls ∪ {x}; end if
104. if (res_a is pread) then
105. t_x.i.decre(x)
106. else
107. ∗ t.x.pls.incre(x, res_a.lvl); end if
108. ∗ v ← t_x.val;
109. unlock t_x;
110. unlock t_mrg;
111. unlock t_mrg;
112. unlock t_mrg;
113. return v;
114. Operation `try_to_commit();`: try_to_merge_{t}(t_i);
115. return (commit);

Specific to p-node (t_p):

117. State of object x:
118. val ∈ V
119. State of transaction t_x:
120. w/y ∈ X × T × T: lockable
121. Operation `get_local_lock(x, t_i);`:
122. if (¬secure_lock_{x}(x, t_i)) then
123. t_i.abort(); end if
124. Operation `release_lock(x, isLocked);`:
125. if (isLocked) then
126. unlock t_x; end if
127. Operation `write(x, v);`:
128. lock t_x;
129. t_x.val ← v;
130. unlock t_x;
131. t_x.lws ← t_x.lws ∪ {x};
132. Operation `search_parent(t_x, t_c, t_o, cm.d, f);`:
133. if (¬f) then
134. if (¬secure_lock_{x}(x, t_x)) then
135. return null; end if
136. end if
137. cm ← t_x.cm;
138. // Check if value is locally available
139. if (t_x.val = null) then
140. release_lock_t(x, ¬f);
141. return null; end if
142. res_x ← (t_x.val, t_x.lvl, ∅, true, true);
143. return res_x;
144. end if
145. // Otherwise, try to read from its parent
146. update_cm(cm, cm.d, mts, cm.d.its);
147. t_x.vts ← t_x.vts ∪ {o};
148. t_x.i.decre(x);
149. if (res_x = null) then
150. release_lock_t(x, ¬f);
151. t_x.i.decre(x);
152. return null; end if
153. if (res_x is pread) then
154. t_x.i.decre(x);
155. else
156. t_x.i.decre(x, res_x.lvl); end if
157. if (res_x is locked) then
158. t_x.pls ← t_x.pls ∪ {x}; end if
159. unlock t_x;
160. unlock t_mrg;
161. unlock t_mrg;
162. return res_x;
163. Operation `try_to_merge(t_c)`:
164. t_c.get_local_lock(mrg, t_c);
165. s ← ∪{x : x ∈ t_x.pls}
166. t_c.get_locks(t_x.lws \ s, t_c);
167. if (check_compatibility(t_x.cm, t_c.cm)) then
168. unlock t_mrg;
169. t_x.abort(); end if
170. for each x ∈ t_c.lws do
171. t_x.val ← t_x.val; end for
172. for each x ∈ t_x.lrs : t_x.val = null do
173. t_x.val ← t_x.val; end for
174. t_x.lws ← t_x.lws ∪ t_x.lws;
175. t_x.lrs ← t_x.lrs ∪ t_x.lrs;
176. update_cm(t_x.cm, t_c.cm.its, t_c.cm.its);
177. t_c.unlock_parent_locks(t_x.pls);
178. t_x.abort_incompat_desc(res_x.s_its);
179. unlock t_x.mrg;
Specific to o-node ($t_o$):

180. **State of base object $x$:**
181. $val \in V$
182. $rs$ and $fbd \subset T$
183. **State of transaction $t_o$:**
184. $ow \subset T$
185. Operation **get_local_lock**($t_o$, $x_1$):
186. lock $t_o.x$;
187. Operation **write**($t_o$, $x$, $v$)
188. lock $t_o.x$;
189. $t_o.x.val \leftarrow v$;
190. $t_o.ow \leftarrow t_o.ow \cup t_o.x.rs$;
191. $t_o.x.fbd \leftarrow t_o.ow$;
192. $t_o.x.rs \leftarrow \emptyset$;
193. unlock $t_o.x$;
194. $t_o.lws \leftarrow t_o.lws \cup \{x\}$;
195. Operation **search_parent**($t_o$, $x$, $t_c$, $t_o$, $cm_d$, $f$):
196. lock $t_o.x$;
197. $cm \leftarrow t_o.cm$;
198. $t_o.x.rs \leftarrow t_o.x.rs \cup \{t_o\}$;
199. **if** ($t_o.x.val \neq \text{null}$) **then**
200. **if** ($\neg \text{check_compatibility}(cm, cm_d) \lor (t_o.x.fbd \cap cm_d.mts \neq \emptyset)$) **then**
201. unlock $t_o.x$;
202. return $null$; **end if**
203. $s_1 \leftarrow t_o.x.fbd \cap t_c.x.its$;
204. $res_x \leftarrow \langle t_o.x.val, t_o.lvl, s_1, false, false \rangle$;
205. unlock $t_o.x$;
206. return $res_x$; **end if**
207. // Otherwise, try to read from its parent
208. update_cm($cm$, $cm_d.mts$, $cm_d.its$);
209. $t_o.x.its \leftarrow t_o.x.its \cup \{t_o\}$;
210. $t_o.trc.incre(x)$;
211. $f2 \leftarrow x \in t_o.pls$;
212. $res_x \leftarrow \text{search_parent}_t_p(x, t_o, cm, f2)$;
213. 214. **if** ($res_x = \text{null})$ **then**
215. $t_o.trc.decre(x)$;
216. unlock $t_o.x$;
217. return $null$; **end if**
218. **if** ($\neg res_x.is.pread$) **then**
219. $t_o.trc.decre(x)$;
220. **else**
221. $t_o.prc.incre(x, res_xlvl)$
222. **end if**
223. **if** ($res_x.is.plocked \land \neg f2$) **then**
224. $t_o.pls \leftarrow t_o.pls \cup \{x\}$; **end if**
225. $res_x.is.plocked \leftarrow false$;
226. unlock $t_o.x$;
227. return $res_x$;
228. Operation **try_to_merge**($t_c$):
229. $t_o.get_local_lock(mrg, t_c)$;
230. $t_o.get_locks(t_c.lrs, t_c.lws, t_c)$;
231. **if** ($\neg \text{check_compatibility}(t_o.cm, t_c.cm) \lor (t_c.lws \neq \emptyset \lor (t_c.cm.mts \cap t_o.ow \neq \emptyset))$) **then**
232. unlock $t_o.mrg$;
233. $t_c.abort();$ **end if**
234. **for each** $x \in t_c.lws$ **do**
235. $t_o.x.val \leftarrow t_c.x.val;$ **end for**
236. **for each** $x \in t_c.lrs$ **:** $t_o.x.val = \text{null}$ **do**
237. $t_o.x.val \leftarrow t_c.x.val;$ **end for**
238. $t_o.ow \leftarrow t_o.ow \cup (\cup_{x \in t_c.lws} t_o.x.rs)$;
239. **for each** $x \in t_c.lws$ **do**
240. $t_o.x.fbd \leftarrow t_o.ow$;
241. $t_o.x.rs \leftarrow \emptyset$; **end for**
242. $t_o.lrs \leftarrow t_o.lrs \cup t_c.lrs$;
243. $t_o.lws \leftarrow t_o.lws \cup t_c.lws$;
244. update_cm($t_o.cm$, $t_c.cm.mts$, $t_c.cm.its$);
245. $t_c.unlock_parent_locks(t_c.pls)$;
246. $t_o.abort.incomp.lws(res_x.sits)$;
247. unlock $t_o.mrg$;

Special case of root node, $t_p$:

if ($t_p$ is a p-node): use Protocol 5.3
if ($t_p$ is a o-node): use Protocol 6.2
7.3.2 State of pessimistic read object, \( x_{pr} \), and helper methods

\( x_{pr} \): The pessimistic read object, \( x_{pr} \) consists of \( trc \) (total read count), and \( lcs \) (level count set) which is a hashtable of \( \langle lvl, rc \rangle \) key-value pairs, where \( lvl \) denotes the level of the highest level \( p\)-node ancestor through which the value of \( x \) was obtained, and \( rc \) (read count) is the number of times the value was read through that level. An entry in \( lcs \) corresponding to a level \( l \) is denoted by \( x_{pr}(l) \). Thus, \( x_{pr}(l).lvl \) and \( x_{pr}(l).rc \) denote the corresponding \( lvl \) and \( rc \) of that entry respectively.

\( diff\_lrc(x_{1pr}, x_{2pr}, l2) \): This helper method is used to decrement the level-wise read count of \( x_{1pr} \) by the read count of the corresponding level-wise entry in \( x_{2pr} \). The parameter \( l2 \) is used to filter out entries in \( x_{2pr}.lcs \) whose level is lower than the level to which \( x_{1pr} \) belongs.

7.3.3 Methods common to \( o\)-node and \( p\)-node \( (t*) \)

The common methods have already been discussed in previous chapters. Therefore, we discuss only the main methods and leave out the discussion of other (self-explanatory) methods.

\( begin\_t\_*(t_p, level, ntype) \): Each transaction \( t_* \) begins with this method, where \( t_p \) denotes the id of its parent, \( level \) is the level of the node in the super tree, and \( ntype \) is the type of the transaction, \( p\)-type or \( o\)-type. Here, \( ntype \) has been introduced for enabling easy identification of the type of a transaction.

\( unlock\_to\_ancestors\_t\_*(s) \): In \( MxSTM \), the \( unlock\_to\_ancestors \) method uses the method \( prc\_decr \) to decrement the level-wise read count at each level, and release
the pessimistic lock on the parent’s objects accordingly.

\( trc\_incre\_\pi(x) \): As discussed before.

\( trc\_decre\_\pi(x) \): As discussed before.

\( prc\_incre\_\pi(x,l) \): As discussed before.

\( prc\_decre\_\pi(s) \): As mentioned before, this method is used to decrement the level-wise read count of object \( t_\pi.prs.x\_pr \) for each object \( x\_pr \) in \( s\_prs \) using an individual lock on \( t_\pi.prs.x\_pr \). Upon decrementing, if the resulting \( t_\pi.prs.x\_pr.trc \) is 0 then the lock on parent’s object \( t_p.x \) is released. This method is used by an aborting subtransaction to release the pessimistic lock on its ancestor’s objects.

\( abort\_\pi() \): Before aborting, transaction \( t_\pi \) releases the locks on its ancestor’s objects obtained by \( t_\pi \) or its descendants, followed by forcefully aborting its active descendants.

\( abort\_incompat\_desc\_\pi() \): This method is used by an ancestor to abort its incompatible descendants.

\( read\_\pi(x) \): To read from its local copy of its object, \( t_\pi.x \), transaction \( t_\pi \) locks \( t_\pi.x \). If \( t_\pi.x \) is null-valued, then \( t_\pi \) invokes the method \( search\_parent(x) \) to get the value from its parent’s local copy of \( x \).

Before invoking the \( search\_parent \) method of its parent, it first increments \( t_\pi.prs.x\_pr.trc \). This ensures that, if it does already possess a lock on the parent’s object, then that lock is not released in the meantime due to the abort of a descendant. This is important as in this case \( t_\pi \) assumes that it will continue to hold the lock on \( t_p.x \) and informs its parent not to lock \( t_p.x \) while invoking \( t_p.search\_parent \). If \( res_x \) returned from \( t_p.search\_parent \) is \textit{null}, then \( t_\pi.prs.x\_pr.trc \) is decremented to reset it to its previous value. Alternatively, if \( res_x.is\_plocked \) is \textit{true}, then the lock on the par-
ent’s object was retained in the process and therefore \( t_\pi \) adds \( x \) to \( t_\pi.pls \). Further, if \( res_x.is.pread \) is true, then \( t_\pi.prs.x_{pr}(res_x lvl) \) is incremented. The objects read by \( t_* \) are recorded in its local read set, \( t_*.lrs \). The compatibility set \( cm \) is updated accordingly. At the end of the operation, the lock on \( t_*x \) is released.

\[ try\_to\_commit_\*(\): A (nested) transaction \( t_* \) merges its local read/write sets with those of its parent, and updates the value of the local objects of its parent.\]

### 7.3.4 State of local objects and methods associated with p-node \((t_\pi)\)

**State of local object \( x \):** In the case of a p-node, local object \( x \) has only the value field, and is protected by a lock.

**State of a p-node \((t_\pi)\):** The id of the parent of a transaction \( t_\pi \) is denoted by \( parent \). The sets, \( lrs \) (local read set) and \( lws \) (local write set) record the objects read and written respectively by \( t_\pi \). A p-node also maintains a wait-for graph, \( wfg \), for its children, to detect and resolve deadlock situation among them. The set \( prs \) is used to keep track of the level-wise external pessimistic read count.

\[ write_{t_*}(x,v): \text{self-explanatory.}\]

\[ search\_parent_{t_\pi}(x,t_c,...): \text{The call to this method is cascaded in nature, and is invoked by the child node } t_c \text{ to obtain a lock on object } t_\pi.x \text{ of its parent, } t_\pi. \text{ In case of a duplicate request where } t_c \text{ already holds a lock on } t_\pi.x, t_\pi.x \text{ need not be locked. Now, if } t_\pi \text{ does not have a local value for } t_\pi.x, \text{ then it tries to obtain the value from its own parent } (t_\pi.parent), \text{ before returning that value to its child } t_c. \text{ Like the read operation, this method also, in case of forwarding the request to its parent, updates}\]

155
its \( \text{prs}.x_{pr} \) to ensure correct checking and recording of ownership on its parent’s lock. If \( \text{res}_x.\text{is_plocked} \) returned from its parent is \( \text{true} \), then it indicates that the lock on the parent’s object was retained in the process and therefore \( t_\pi \) adds \( x \) to \( t_\pi.pls \). Finally, if the \( \text{res}_x \) needs to be passed further down, the \( t_\pi \) sets \( \text{res}_x.\text{is_pread} \) and \( \text{res}_x.\text{is_plocked} \) to \( \text{true} \) to indicate to its child that the value was read through a \text{p-node} (i.e., \( t_\pi \)) and a pessimistic lock on its parent was retained in the process.

\textit{try_to_merge}_{t_c}(t_c): \) This method is invoked by the child node to merge its results (values of objects, read set, write set, etc.) with the parent. Please note that, in case the child transaction writes a local object \( x \) without having previously read its value from the parent, then it does not possess the lock on that object of the parent. Hence, at the time of merging, locks on such objects of the parent are obtained. Next, the value of all the objects, belonging to \( lws \) and \( lrs \) of the child node, is updated using the local copy of the child node. Finally, the compatibility object \( \text{cm} \) and sets \( lrs \) and \( lws \) of the child node are merged with those of its parent, before releasing the locks.

### 7.3.5 State of local objects and methods associated with o-node \((t_\omega)\)

\textit{State of local object \( x \):} Here, a local object \( x \) has three components, namely value field \((\text{val})\), read set \((\text{rs})\) and forbidden set \((\text{fbd})\). Each local copy is protected by a lock.

\textit{State of an o-node \((t_\omega)\):} Here, a transaction uses the \textit{parent} field to store the id of its parent. Further, sets, \( lrs \) and \( lws \), are used to record the objects read and written
respectively by \( t_\omega \). The set \( ow \) (overwritten set) denotes the of those children of \( t_\omega \), that read an object \( t_\omega.x \) which has been modified (overwritten) later. An o-node also maintains the set \( prs \) to keep track of the level-wise external pessimistic read count.

\texttt{write}_t_\omega(x, v): \textit{First,} \( t_\omega.x \) \textit{is locked. The value of} \( t_\omega.x \) \textit{is updated. The ids of all the children that previously read} \( t_\omega.x \) \textit{are added to} \( t_\omega.ow \), \textit{followed by updating} \( t_\omega.x.fbd \) \textit{using} \( t_\omega.ow \), \textit{and clearing} \( t_\omega.x.rs \). \textit{Finally,} \( t_\omega.x \) \textit{is unlocked, and} \( x \) \textit{is added to} \( t_\omega.lws \).

\texttt{search\_parent}_t_\omega(x, t_c, \ldots): \textit{Similar to the} \texttt{search\_parent}(x) \textit{method of a} p\textit{-node, this method is invoked by the child node. If the value of the requested object is null, then the value is obtained from the parent by invoking the} \texttt{search\_parent}(x) \textit{of its own parent (recursive call). Unlike in case of a} p\textit{-node, before returning the value of the object to the child node (}t_c\textit{), it is checked if it is legal for} t_c \textit{to read} t_\omega.x \textit{by checking for the membership of} t_c \textit{in} t_\omega.x.fbd \textit{as well as doing a compatibility check. If the validation is not successful, then} t_c \textit{is aborted. Otherwise,} t_c \textit{is added to} t_\omega.x.rs \textit{before unlocking} t_\omega.x \textit{and returning the value. Similar to the operation for} p\textit{-nodes,} t_\omega.prs.xpr \textit{is updated accordingly. Note that in the case of an o\textit{-node, it only sets} res_x.is\_plocked \textit{to} false \textit{to inform its child that a pessimistic lock was not obtained at its parent level in the process.}

\texttt{try\_to\_merge}_t_\omega(t_c): \textit{This method is invoked by the child node,} t_c. \textit{First, all the objects belonging to} t_c.lrs \cup t_c.lws \textit{in the parent’s} (t_\omega’s) \textit{local space are locked, followed by locking} t_\omega.mrg. \textit{The validation for} t_c \textit{consists of ensuring} t_c.lws \neq \emptyset \textit{and} t_c \notin t_\omega.ow \textit{as well as checking for compatibility. If the validation is not successful, then} t_c \textit{is aborted. Otherwise, the value of the parent’s objects that are present in} t_c.lrs \textit{and} t_c.lws \textit{are updated using} t_c’s \textit{local copy of those objects. Next, the ids of transactions in} t_\omega.x.rs \textit{of each} t_\omega.x \textit{that is modified are added to} t_\omega.ow \textit{. The set} t_\omega.ow \textit{is used to}
update \( t_\omega.x.fbd \) of each \( t_\omega.x \) that is modified, followed by clearing \( t_\omega.x.rs \). If \( t_\omega \) is not the super transaction, then \( t_c.lrs \) and \( t_c.lws \) are merged with the corresponding sets of the parent, \( t_\omega \).

### 7.3.6 About deadlock freedom

To show that MxSTM is deadlock-free, we shall examine \( p\)-nodes and \( o\)-nodes. Observe that, by construction (also, refer to discussion in Section 5.1.2.2 and Section 6.3.4), any access to local objects of a \( p\)-node is duly recorded in its wait-for-graph, \( wfg \), to prevent any deadlock situation. Thus, no deadlock is possible involving objects of \( p\)-nodes. In other words, locks on such objects cannot be held for an indefinite period of time.

In case of an \( o\)-node, say \( t_\omega \), locks on its objects are released soon after the completion of the external read on them. However, when the object \( t_\omega.x \) is locked as part of the \texttt{search\_parent} operation that escalates up to a \( t_\omega \)'s ancestor, say \( t_\pi1 \), that is a \( p\)-node, then \( t_\pi1 \)'s local object \( t_\pi1.x \) may not be immediately available. In that case, the lock on \( t_\omega.x \) will be held until \( t_\pi1.x \) becomes available. As the lock on \( t_\pi1.x \) cannot be held indefinitely (based on the discussion earlier about \( p\)-nodes), it follows that the lock on \( t_\omega.x \) cannot be held forever. In the meantime, while \( t_\omega.x \) is locked, any other request for \( t_\omega.x \) (for a read or commit of a child node) will ensue in a wait that is deadlock-free. Thus, locking of \( t_\omega \)'s objects for external reads on them is free from deadlocks.

During commit, deadlock freedom is ensured by allowing only one child to merge with the parent, by using the parent’s \texttt{mrg} lock. In the case of a \( p\)-node, the wait-
for-graph is used in addition for this purpose.

7.4 Correctness

7.4.1 Definition of linearization points of events

7.4.1.1 At a \( p \)-node \( t_\pi \):

Let \( \ell_{op} \) denote the linearization point of an event. Then, the linearization points of the various events in the history are defined as follows:

i Local read/write operation of \( t_\pi \)

(a) \( read_{t_\pi}(t_\pi.x) : \ell_{op} \) corresponds to the time when it unlocks \( t_\pi.x \) (line 110)

(b) \( write_{t_\pi}(t_\pi.x) : \ell_{op} \) corresponds to the time when it unlocks \( t_\pi.x \) (line 130)

ii External read operation of \( t_\pi \)

(a) \( read_{t_\pi}(t_a.x) : \ell_{op} \) corresponds to the time just after \( t_\pi.cm \) is updated (line 101)

iii External read of a descendant \( t_d \) on \( t_\pi \)'s object or that of \( t_\pi \)'s ancestor \( t_a \):

(a) \( read_{t_d}(t_\pi.x) : \ell_{op} \) corresponds to the time just after \( t_d \) reads \( t_\pi.cm \) for consistency checking (line 137)

(b) \( read_{t_d}(t_a.x) : \ell_{op} \) corresponds to the time just after \( t_d \) reads \( t_\pi.cm \) for consistency checking (line 137)

iv Write due to commit of child \( t_c \)

(a) \( write_{t_c}(t_\pi.x) : \ell_{op} \) corresponds to the time just after \( t_c \) updates \( t_\pi.x \) (line 171)
v Commit of child $t_c$

(a) $C_{t_c} : \ell_{op}$ corresponds to the time just after $t_w.cm$ is updated (line 245).

### 7.4.1.2 At an o-node $t_w$:

i Local read/write operation of $t_w$

(a) $\text{read}_{t_w}(t_w.x) : \ell_{op}$ corresponds to the time when it unlocks $t_w.x$ (line 110)

(b) $\text{write}_{t_w}(t_w.x) : \ell_{op}$ corresponds to the time when $t_w$ updates $t_w.ow$ (line 190)

ii External read operation of $t_w$

(a) $\text{read}_{t_w}(t_w.x) : \ell_{op}$ corresponds to the time when $t_w.cm$ is updated (line 101)

iii External read of a descendant $t_d$ on $t_w$’s object or that of $t_w$’s ancestor $t_a$:

(a) $\text{read}_{t_d}(t_w.x) : \ell_{op}$ corresponds to the time when $t_d$ reads $t_w.cm$ for consistency checking (line 197)

(b) $\text{read}_{t_d}(t_a.x) : \ell_{op}$ corresponds to the time when $t_d$ reads $t_w.cm$ for consistency checking (line 197)

iv Write due to commit of child $t_c$

(a) $\text{write}_{t_c}(t_w.x) : \ell_{op}$ corresponds to the time when $t_c$ updates $t_w.x$ (line 236)

v Commit of child $t_c$

(a) $C_{t_c} : \ell_{op}$ corresponds to the time just after $t_w.cm$ is updated by $t_c$ (line 245).
7.4.2 Definition of linearization point of a transaction $t$

7.4.2.1 At $p$-node (i.e., parent $t_p$ of $t$ is a $p$-node):

1. If $t$ commits, its linearization point, $\ell_t$, lies at the time just after it updates the parent’s $cm$ (consistency management) object (line 176).

2. If $t$ aborts, $\ell_t$ coincides with the last time when $t$ reads $t_p.cm$ for its successful external read operation at $t_p$’s level (line 137).

7.4.2.2 At $o$-node (i.e., parent $t_p$ of $t$ is an $o$-node):

1. If $t$ is an update transaction that commits, its linearization point, $\ell_t$, lies at the time just after it updates the parent’s $ow$ and $cm$ (consistency management) object (line 245).

2. If $t$ is a read only committed transaction, then $\ell_t$ is placed at the earliest of (i) the time it reads $t_p.cm$ for its last successful external read operation (line 197), and (ii) the time just before $\hat{t}$ (any id in $t.cm.mts$) is added to $t_p.ow$ (if it ever is) (lines 190, 239).

3. If a transaction $t$ aborts, $\ell_t$ is determined as if it were a read only transaction, i.e., $\ell_t$ lies at the earliest of (i) the time it reads $t_p.cm$ for its last successful external read operation (line 197), and (ii) the time just before $\hat{t}$ (any id in $t.cm.mts$) is added to $t_p.ow$ (if it ever is). (lines 190, 239).
This definition of linearization points should be used when we consider the various level-wise histories of MxSTM to establish the correctness.

### 7.4.3 Proof

To show that the protocols presented in ParSTM and HParSTM have been integrated in a correct fashion to obtain MxSTM, we show that the following two points indeed hold true: (1) at o-nodes, the lock management is done in an optimistic fashion, whereas (2) at p-nodes, the locks are managed in a pessimistic manner (2PL for nested transactions).

**Lemma 7.1.** In MxSTM, the locks associated with an o-node ($t_\omega$) are managed in an optimistic manner, whereas those associated with a p-node ($t_\pi$) are operated in a pessimistic manner.

**Proof.** Let us take the case of an o-node, $t_\omega$, first. Consider an object $t_\omega.x$. We need to show that lock on $t_\omega.x$ is retained only for the duration of the read/write operation, not for the entire lifespan of the transaction. We consider all the methods (cases) in which $t_\omega.x$ is locked, and show that $t_\omega.x$ in unlocked at end of each of these methods.

- **read$_{t_\omega}(t$, $x$):** $t_\omega.x$ locked at line 90, and released at line 110.

- **search$_{parent}(t_\omega(x, t_\text{c}, t_\text{o}, cm_d, f):** $t_\omega.x$ locked at line 196, and released at line 202, 206, 216 or 226.

- **write$_{t_\omega}(x, v):** $t_\omega.x$ locked at line 188, and released at line 193.

- **try$_{to\_merge(t_\omega.x):** $t_\omega.x$ locked at line 230, and released at line 246 (or 234 due to abort).
Next, we show that when the lock on $t_\pi.x$ is obtained by $t_\pi$’s child $t_c$ in a successful read/write operation, it is released by $t_c$ only upon its completion (commit/abort). The $read_{t_\pi}(x)$ and $write_{t_\pi}(x,v)$ methods are used for local read/write operations of $t_\pi$. As such, $t_\pi$ releases the lock immediately after its read/write operation on its local object $t_\pi.x$.

Let us now consider the corresponding methods of $t_\pi$ in which $t_\pi.x$ is locked by its children.

- $search\_parent_{t_\pi}(x, t_c, t_o, cm, f)$: $t_\pi.x$ locked at line 134, but the object is not unlocked in case of successful read. Unlocking is done only if the read step is unsuccessful (lines 140 or 151).

- $try\_to\_merge_{t_\pi}(t_c)$: $t_\pi.x$ locked at line 166. This operation is performed by a transaction during its commit phase which marks the end of the lifespan of the transaction. The child transaction $t_c$ releases all the locks on its parent’s objects only upon successful merging/committing (line 177) or upon aborting (line 169).

Observe that, when a subtransaction performs an external read on an object $t_\pi.x$ of $p$-node $t_\pi$, it holds the lock on that object until it terminates. Thus, the lock obtained during an external read is held for the lifespan of the descendant transaction. On the other hand, in case of an $o$-node $t_\omega$, the lock on $t_\omega.x$ is released upon completion of the external read operation. Here, such a lock is held only until the external read operation completes.

Next, we show that MxSTM handles incompatible transactions correctly.
Lemma 7.2. MxSTM ensures correct checking of incompatible transactions at each level.

Proof. Checking for incompatibility of transactions is done using the sets, $mts$ and $its$. The incompatibility of transactions is introduced due to reading from and update of objects at higher level $o$-nodes. Hence, we need to show that (1) when a subtransaction $t$ reads an object $t_\omega.x$ from an ancestor $t_\omega$, $t$ is added to $t_\omega.x.rs$, irrespective of the type of $t$ or its intermediate ancestors, and (2) the sets, $cm.mts$ and $cm.its$, are updated at every level.

The recursive method $\text{search\_parent}_{t_o}(x, t_c, t_o, cm_d, f)$ that is used by a subtransaction to read from ancestors, has the same signature for both types of nodes. As such, the id of the subtransaction, $t_o$, originally initiating the method is propagated upwards. If the value is successfully returned from an $o$-node (ancestor, $t_\omega$), then $t_o$ is added to $t_\omega.x.rs$ (line 198). When $t_\omega.x$ is updated, $t_o$ is added to $t_\omega.x.fbd$ and $t_\omega.ow$. This ensures that later, when some other subtransaction $t_d$ that is incompatible with $t_o$, reads from $t_\omega$, the set $t_d.cm.its$ is updated correctly.

Further, observe that every node $t_\pi$ ($o$-node as well as $p$-node) maintains the sets, $t_\pi.cm.its$ and $t_\pi.cm.mts$. The set $t_\pi.cm.mts$ is updated at the time of merge operation (lines 176, 245), whereas $t_\pi.cm.its$ is updated while reading from an ancestor (line 101) and during the merge process (lines 176, 245).

Next, we shall show that the protocol handles duplicate requests at a $p$-node correctly. First, we shall show that at any level (node), for a given object $x$, only one request can come from the same child. Then, we shall show that at a $p$-node, $t_\pi$, any
subsequent duplicate request for accessing $t_{\pi}.x$ from a child $t_c$ that already retains
lock $t_{\pi}.x$ is handled correctly. Correctness here means (i) as $t_c$ already holds the lock
for $t_{\pi}.x$, it should be allowed to access $t_{\pi}.x$ directly, and (ii) in case the subsequent call
fails for some reason, $t_{\pi}.x$ should not be unlocked in the current operation context,
to preserve the state of previous successful call.

**Lemma 7.3.** At any level (node) $t$, only one request for external read on an object
$t.x$ can come from the same child $t_c$ at a time.

*Proof.* As per the statement of the lemma, note that the types of $t$ and $t_c$ do not
matter and the lemma is applicable in general. Further, we know that the external
read is performed through the `search_parent` method.

Observe that before invoking $t.search_parent$, the lock on $t_c.x$ needs to be obtained
(line 90 for $t_c$’s own read; line 134 in case of `p-node`; line 196 in case of `o-node`). In
case $t_c$ is a `p-node`, then looking at line 133 one can argue that $t_c.x$ is not locked at
line 133 in all cases. However, it should be noted that the exception at line 133 is only
applied for a descendant $t_d$ of $t_c$ that already has a lock on $t_c.x$. Thus, the statement
that only one transaction holds the lock on $t_c.x$ at a time is true.

Now, one may question if two or more concurrent invocations for reading $x$ can
come from $t_d$? The answer is ‘no’. To put the answer in perspective, let us look at the
behaviour of concurrent requests for external reads. The invocation of `search_parent`
is initiated by the `read` operation. Let us consider two descendants, $t_{d1}$ and $t_{d2}$ of $t_d$
such that $t_d$ is their least common ancestor. Both the descendants want to read $x$.
For the sake of the argument, let us also assume that requests of $t_{d1}$ and $t_{d2}$ reach $t_d$
concurrently. Now, based on the earlier discussion, only one of the two descendants
can possibly obtain the lock on $t_d.x$ first. That means the request of only one of the descendants will be propagated forward beyond $t_d$ while the other request has to wait. That means only one request for external read on an object $x$ can come from the same descendant.

Thus, following this line of argument, we conclude that only one request for an external read on $t.x$ can come from a child $t_c$ at a time. 

**Lemma 7.4.** At a p-node, duplicate request by a child is allowed in a non-blocking manner.

*Proof.* Let $t_\pi$ be a p-node and $t_c$ be its child that has already read $t_\pi.x$ through the $\text{search\_parent}$ method. We shall show that the subsequent request by $t_c$ to read $t_\pi.x$ is allowed, given that $t_c$ is consistent with $t_\pi$. Observe that $t_c$ may be required to make the duplicate read at $t_\pi$ for its own purpose or on behalf of its descendant.

Let $R_o$ and $R_d$ denote the original and duplicate requests respectively. Based on Lemma 7.3, we have $R_o < R_d$, i.e., the duplicate request occurs after the original request has completed. Now, given that $R_o < R_d$, we have

$x \in t_c.pls$ during $R_d$ (due to lines 142, 158-159) 

$\Rightarrow t_c$ does not have to wait for lock during $R_d$ (due to lines 94, 133) 

Now, if $t_c$ is compatible with $t_\pi$ (line 139) during $R_d$, then $t_c$ can read $t_\pi.x$ (line 142)

Thus, $t_c$ can make duplicate requests at its parent level without having to wait for a lock on $t_\pi.x$.

Further observe that, as no new request for locking $t_\pi$’s object is made, the duplicate request does not alter the existing wait-for dependency for locks at $t_\pi$’s level. Hence, a deadlock scenario does not arise in this case.
Lemma 7.5. Sets pls and prs are managed correctly during an external read.

Proof. Set pls (parent lock set) is used to denote whether a subtransaction currently holds a lock on an object of its parent, where as prs (pessimistic read set) is used to track the number of read count and the level of the highest level pessimistic node whose object was locked during an external read involving this transaction.

Let us consider a subtransaction $t$ with its parent $t_p$.

Case pls:

First, for pls, let us show that an object $x$ is added to $t.pls$ at a transaction $t$ only if $t$ retains the lock on its parent’s object $t_p.x$ upon a successful external read. By construction of the Protocol, $x$ is added to $t.pls$ only if the response object $res_x$ from its parent $t_p$ has its property $is\_plocked$ (Is Parent Locked) set to $true$ (lines 102-103; 158-159; 223-224).

Now, if $t_p$ is an $p$-node, then $res_x.is\_plocked$ is set to $true$ only in case of a successful external read operation (line 142, 161) at $t_p$. Otherwise, $t_p$ returns $null$ which does not lead to adding $x$ to $t.pls$ (lines 96-98; 150-153; 214-217).

On the other hand, if $t_p$ is an $o$-node, $res_x.is\_plocked$ is set to $false$ even in case of a successful external read. Consequently, $t_c$ does not add $x$ to $t_c.pls$ while reading from an $o$-node.

Case prs:

Let $t_d$ be a descendant that reads $x$ from its ancestor $t_a$. Let $t_{\pi}$ be the highest level $p$-node in the path from $t_a$ to $t_d$. Then, we show that:

(a) In case of a successful read, $t.prs$ is updated (increment $t.prs.x_{pr}(t_{\pi}.lvl).rc$ by 1)
at each descendant \( t \) of \( t_\pi \) in the path from \( t_\pi \) to \( t_d \).

(b) Conversely, when \( t_d \) aborts, \( t.prs.x_pr(t_\pi.lvl).rc \) is decremented by 1 at each intermediate ancestor \( t \) in the path from \( t_d \) to \( t_\pi \), excluding \( t_\pi \).

**Proof for part(a):**

Before invoking \texttt{search\_parent} of the parent \( t_p \), the invoking child transaction \( t \) optimistically increments its \( t.prs.x_pr.trc \) by 1 (lines 93, 147, 211). By construction of \texttt{search\_parent} in \textit{MxSTM}, observe that if the response object \( res_x \) passes through a \( p\)-node, \( t_\pi \), then \( res_x.is\_pread \) is set to \texttt{true} (line 160). Thereafter, all the subsequent descendants of \( t_\pi \) in the path from \( t_d \) increment \( prs.x_pr(t_\pi.lvl).rc \) by 1 (line 107, 157, 221). If \( res_x \) is \texttt{null} or \( res_x.is\_pread \) is \texttt{false}, then \( t.prs.x_pr.trc \) is decremented by 1 to compensate for the optimistic increment made to \( t.prs.x_pr.trc \) at the beginning of the invocation (line 97, 105, 152, 155, 215, 219). Thus, we see that \( prs \) is correctly updated during external read operation.

**Proof for part(b):**

Now, we shall show that when \( t_d \) aborts, \( t.prs.x_pr \) is decremented at each of the ancestors in the path from \( t_d \) to \( t_\pi \), excluding \( t_\pi \). Observe that the entries in \( t_d.prs.x_pr \) contains the \textit{read count} as well as level of \( t_\pi \). Upon aborting, \( t_d \) invokes \texttt{unlock\_to\_ancestors} with its \( prs \). Observe that the recursive operation \texttt{unlock\_to\_ancestors} propagates upward up to \( t_\pi \)'s child in the path from \( t_d \) to \( t_\pi \) (lines 31, 34). At each level \( t \), the entries in \( t.prs.x_pr \) are decremented by an amount equal to the corresponding level-wise entries in \( t_d.prs.x_pr \) (line 32, 54, 6-10).

\[\square\]

**Lemma 7.6.** \textit{MxSTM guarantees safety during duplicate external read and unlock\_to\_ancestors}
occurring at the same time at a level.

Proof. Given parent $t_p$ of node $t_c$ is a $p$-node, if $x \in t_c.pls$ holds true at the time of invocation of $\text{search}_{parent_{t_p}}$, then invocation of $\text{unlock}_{to\_ancestor}$ at $t_c$ does not alter the relation $x \in t_c.pls$ while $\text{search}_{parent_{t_p}}$ is executing.

In other words, if a descendant $t_d$ of $t_c$ previously read $x$ through $t_p$ then the lock on $t_p.x$ is retained by $t_c$ and $x$ is added to $t_c.pls$. In that case, $x \in t_c.pls$ evaluates to true at the time of the subsequent invocation of $t_p.search_{parent}(x)$. Now, we have to show that invocation of $t_p.unlock_{to\_ancestors}(x, ...)$ due to abort of $t_d$, after the invocation of $t_p.search_{parent}(x)$ by $t_c$, does not unlock $t_p.x$. This is important because this invocation $t_p.search_{parent}(x)$ assumes that $t_c$ already holds the lock on $t_p.x$ and does not try to lock $t_p.x$ for $t_c$.

By the design of the Protocol, following Lemma 7.5, if $x \in t_c.pls$ holds true during a $\text{search}_{parent}$ invocation, then it also means $t_c.prs.x.pr.trc > 0$. Let $t_c.prs.x.pr.trc = n_1$. Then, we have $n_1 > 0$. Observe that before invoking $t_p.search_{parent}(x)$, $t_c$ invokes $t_c.trc.incre(x)$ to increment $t_c.prs.x.pr.trc$ by 1. Conversely, the execution of $t_c.unlock_{to\_ancestors}(x, ...)$ invokes $t_c.prc.decre(\langle x, l, n \rangle)$ to decrement $t_c.prs.x.prc.trc$ by a count $\leq n_1$. Thus, given $t_c.trc.incre(x) < t_c.prc.decre(s)$, we have:

Before execution of $t_c.trc.incre(x)$, $t_c.prs.x.pr.trc = n_1$

After execution of $t_c.trc.incre(x)$, $t_c.prs.x.pr.trc = n_1 + 1 = n_2$

Before execution of $t_c.prc.decre(\langle x, l, n \rangle)$, $t_c.prs.x.rc = n_2$

After execution of $t_c.prc.decre(\langle x, l, n \rangle)$, $t_c.prs.x.rc = n_3$

In the worst case scenario, $n$ could be as large as $n_1$, but even then we have:
\[ n_3 = n_2 - n \geq n_2 - n_1 \geq 1 > 0 \]

Since the criterion for unlocking \( t_p.x \) is that \( t_c.prs.x_{pr}.trc = 0, n_3 \geq 1 \neq 0 \Rightarrow t_p.x \) is not unlocked during the execution of \( t_c.unlock_to_ancestors(x,...) \), i.e., \( x \in t_c.pls \) still holds true.

\[ \square \]

**Lemma 7.7.** Duplicate external read operations at a level obtaining values from different ancestral levels are logged correctly.

**Proof.** Let \( R_1 \) be the first external read operation by a descendant \( t_{d1} \) of an o-node, \( t_\omega \), on its ancestor \( t_\pi \). Observe that the intermediate ancestors of \( t_\omega \) up to \( t_\pi \) could be a combination of o-nodes and p-nodes. Let \( t_\omega' \) be such an intermediate ancestor such that \( t_\omega'.x \) becomes non-null after \( R_1 \) has completed. This can happen either due to a local write by \( t_\omega' \) or a commit of its child. Let \( R_2 \) be the subsequent external read operation by \( t_\omega' \)’s another descendant, \( t_{d2} \), such that it obtains the value from \( t_\omega'.x \).

Further, let \( t_{\pi'} \) be the highest level p-node in the path from \( t_\omega' \) to \( t_{d2} \). Then we have to show that \( t_\omega.prs.x_{pr}.lcs \) contains \( \langle t_\pi.lvl, 1 \rangle \) as well as \( \langle t_{\pi'}.lvl, 1 \rangle \).

By construction of MxSTM, during \( R_1 \), \( \langle t_\pi.lvl, 1 \rangle \) is added to \( t_\omega.prs.x_{pr}.lcs \) (due to lines 142 or 160, 221, 49). Similarly, when \( R_2 \) completes, \( \langle t_{\pi'}.lvl, 1 \rangle \) is added to \( t_\omega.prs.x_{pr}.lcs \) (due to lines 205, 160, 221, 49).

\[ \square \]

**Lemma 7.8.** At a level \( t_c \), if the highest level p-node \( t_{\pi'} \) registered by a duplicate external read operation \( R_2 \) is different from the p-node \( t_\pi \) registered by the original read operation \( R_1 \), then the level \( t \) from which value has been read by \( R_2 \) is an o-node.
and it lies between $t_\pi$ and $t_\pi'$. 

Proof. Here $R_1$ is the original external read operation through a subtransaction $t_c$ such that the value was read from an ancestor $t_a$ and the highest level $p$-node registered in the process is $t_\pi$. $R_2$ is the subsequent duplicate external read operation such that it registers $t_\pi'$ as the highest level $p$-node. Node $t$ is the ancestor from which $R_2$ obtains its value. Then, we show that (a) $t$ is not an ancestor of $t_a$, (b) $t$ is an $o$-node, and (c) $t$ lies between $t_\pi$ and $t_\pi'$ in the ancestral path.

Proof of part (a): As $R_1$ read from $t_a$, it means $t_a$ is non-null valued. That means that invocation of search_parent during $R_2$ cannot go beyond $t_a$. Hence, $t$ cannot be an ancestor of $t_a$.

Further, $t$ cannot be $t_a$ as in that case, the highest level $p$-node registered by $R_2$ would be same as the one registered by $R_1$. Hence, $t$ is a descendant of $t_a$.

Proof of part (b):

If $t$ is a $p$-node, then $t.x$ is locked due to the pessimistic locking of $t.x$ for the original read operation $R_1$. As such, $t.x$ cannot be updated to have a new value. Also observe that the original external read operation propagated to $t_\pi$ because $t.x$ was null-valued initially. As $t.x$ remains null-valued, $R_2$ cannot read from $t.x$. However, if $t$ is an $o$-node, $t.x$ can be updated in the mean time to have a new value. This means, $t$ is an $o$-node.

Proof of part (c):

By contrast, assume that $t$ lies between $t_a$ and $t_\pi$. Then, the $p$-node registered by $R_2$ would be $t_\pi$ itself (due to lines 205, 160, 221, 49). That means $t$ cannot be an ancestor of $t_\pi$. The only alternative then is that $t$ is a descendant of $t_\pi$. Further, for
Lemma 7.9. An abort of a subtransaction releases the lock up to the appropriate level.

Proof. Let $t_d$ be a subtransaction that has performed an external read operation through its ancestor $t_\pi$ that is a $p$-node. Observe that the lock on $t_\pi$ is retained in the process as it is a $p$-node. Now, if $\hat{t}_d$ ($t_d$ or a transaction containing $t_d$ in its mts) aborts, then the idea is that the lock on $t_\pi.x$ should be released.

Observe that upon successful external read operation, subtransaction $t_d$ records the level of the ancestor $t_a$ it read $x$ from. If there is a $p$-node ancestor $t_\pi$ involved in the path from $t_a$ to $t_d$, then level information $x_{pr} \langle t_\pi.lvl, n \rangle$ is captured in set $prs$ at each descendant of $t_\pi$ in the path from $t_\pi$ to $t_d$. Later, if any of these descendants, say $t$, aborts, then $t$ invokes $unlock.to.ancestors$ using $x_{pr} \langle t_\pi.lvl, n \rangle$ in $t.prs$. As, at intermediate ancestor level $t$, we have $t_\pi.lvl > t.lvl$, the $unlock.to.ancestors$ invocation is propagated up to immediate child of $t_\pi$ in the path from $t_\pi$ to $t_d$ (due to line 31, 33-34). During $unlock.to.ancestors$, at the intermediate level $t$, $t.prs.x_{pr} \langle t_\pi.lvl \rangle.rc$ is decremented by $n$ and if resulting $t.prs.x_{pr}.trc$ is 0 and $t$ holds the lock on its parents object $t_p.x$, then $t_p.x$ is unlocked.

Thus we see that upon the abort of a subtransaction, it releases the lock up to the appropriate level.

Lemma 7.10. A failed duplicate request at a $p$-node does not alter the state of the

\[ \text{\ } \]

172
original request.

Proof. As in Lemma 7.4, let us consider a p-node, $t_\pi$, with its child $t_c$ such that $t_c$ is an o-node and already retains the lock on $t_\pi.x$ owing to its previous invocation of $t_\pi.search.parent$ on behalf its descendant $t_{d1}$. Now, suppose $t_c$ invokes $t_\pi.search.parent(x, t_c, ...)$ again on behalf of its another descendant $t_{d2}$ such that the request fails at $t_\pi$’s level.

Typically, in the search.parent method of a p-node, the lock $t_\pi.x$ is released in case of failure of the operation (lines 140, 151). As a duplicate request does not participate in locking $t_\pi.x$ (lines 148-149, 212-213), unlocking $t_\pi.x$ in case of failure would invalidate the expectation that the lock on $t_\pi.x$ should be retained on behalf of the prior original request by $t_{d1}$. The failure of $t_{d2}$’s attempt for an external read should have no effect on the state of $t_{d1}$. This is ensured by unlocking $t_\pi.x$ only if the lock on $t_\pi.x$ was obtained in the context of the current operation (lines 94-95, 148-149, 212-213, 125-126). As the lock on $t_\pi.x$ is not obtained in the case of a duplicate request, no unlocking of $t_\pi.x$ is done in the case of failure either. Thus, the state of the original request is preserved.

\[\square\]

Lemma 7.11. Abort of descendants is non-blocking.

Proof. Abort of descendants is initiated by an ancestor in a top-to-bottom manner in a transaction tree (lines 70-72, 74-75). The methods used for this purpose are - abort, force_abort, abort_incompat_desc. Observe that no lock is obtained in any of these methods. Therefore, abort of descendants is non-blocking. \[\square\]
7.4.3.1 History \( \mathcal{H}_{t_\omega} \) produced at an \textit{o-node} \( t_\omega \)

The history \( \mathcal{H}_{t_\omega} \) produced at an \textit{o-node} \( t_\omega \) is similar to the history produced at a node \( t \) by HParSTM (Chapter 6). By construction, the objects of \( t_\omega \) in MxSTM are accessed in the same way as in HParSTM (as discussed already in Section 7.1.1.2).

Observe that, MxSTM has been designed in a way that we do not check the types of transactions involved while accessing the objects. It is automatically taken care of, by keeping the signature of the methods (e.g., \textit{search\_parent}, etc) intact but modifying their definitions, wherever required, accordingly for \textit{o-node} and \textit{p-node}. Further, recall that the linearization point of a child depends upon the type of its parent, not its own type (Section 7.4.1). Thus, when we consider the history at \( t_\omega \), the type of its children does not matter. Its child could be a \textit{p-node} or \textit{o-node}. This means that the level-wise history at an \textit{o-node} in MxSTM can be constructed in the same way as in case of a node in HParSTM.

While constructing the level-wise history at a node \( t_\omega \), in bottom-to-top manner in a transaction tree (Section 3.4 and 3.7), we concern ourselves only with the objects of \( t_\omega \), and treat the subtree rooted at a \( t_\omega \)’s child as single transaction. We do not need to care about the composition of transactions in that subtree, or the type of that child.

The set of proofs used for HParSTM can be directly applied to show the correctness for \( \mathcal{H}_{t_\omega} \).

Next, we show that the history produced at an \textit{p-node}, \( t_\pi \), in MxSTM is same as that produced at a node in ParSTM.
7.4.3.2 History ($\mathcal{H}_{t\pi}$) produced at a \textit{p-node} ($t\pi$)

Similarly, by construction (Section 7.1.1.1), we observe that the methods associated with $t\pi$ node in MxSTM are similar to the methods defined for a non-root node in ParSTM (Chapter 5). In other words, the objects associated with $t\pi$ in MxSTM are operated in the same fashion as are the objects of a non-root node in ParSTM. Therefore, the history $\mathcal{H}_{t\pi}$, produced at a node $t\pi$ by MxSTM, is similar to the history $\mathcal{H}_t$ produced locally at a (pessimistic) node $t$ by ParSTM. Hence, the set of proofs used for $\mathcal{H}_t$ can be applied for $\mathcal{H}_{t\pi}$. 
Chapter 8

Conclusion and future work

This thesis provides a comprehensive study into the complexities involved in designing STM protocols for closed nested transactions. Compared to non-nested transactions, nested transactions pose a set of new problems unique to them that need to be treated differently. To this end, we provide a formalism for closed nested transactions and extend the definition of opacity used for non-nested transactions to define level-wise opacity as a consistency criterion for nested transactions. In addition, we describe a model for mapping the execution of nested transactions to obtain level-wise histories in a transaction tree. We also provide a framework for formally proving the correctness of STM protocols for nested transactions. This framework can be used for establishing the correctness of other STM protocols for nested transactions.

Furthermore, we design a set of four STM protocols (SimpSTM, ParSTM, HParSTM, and MxSTM) for closed nested transactions. These protocols offer different modes of concurrency. Starting with SimpSTM, a simple protocol which offers no concurrency at the nested level (subtransactions are executed serially), we progress to ParSTM
which uses *pessimistic concurrency control* scheme at nested level and offers partial concurrency: two subtransactions in the transaction tree can execute concurrently as long as they do not try to access the same object; otherwise they execute sequentially. Next, we obtain full concurrency in HParSTM by employing *optimistic concurrency control* mechanism at each node of the transaction tree. Finally, we combine ParSTM and HParSTM to obtain a *hybrid* protocol, MxSTM, in which some nodes operate under optimistic concurrency control while others under pessimistic concurrency control mechanism. The protocols ParSTM and HParSTM are carefully crafted in a modular way, using shared interface, such that the two can be easily integrated to obtain MxSTM. Special cases have been duly discussed and addressed.

In future, it would be interesting to implement and test the protocols against standard benchmarks to analyze their performance, especially under varying levels of nesting. Further, MxSTM can be very useful in developing new applications where different degrees of concurrency can be employed at different levels. For example, the level where most of the child threads are read only, *o-node* (optimistic approach) can be used, and the one where the frequency of updates by children is high, *p-node* (pessimistic approach) can be employed.
Bibliography


