Concurrency Control in Transactional Systems: Spring 2023

Project 1: Efficient Optimistic Closed Nested Software Transactional Memory Systems

Team members
Souvik Sarkar(CS23MTECH02001)

Chinmay kapkar(CS22MTECH14010)

Report

Introduction:

Concurrency in traditional programming systems is usually implemented using locks, which have several drawbacks such as deadlocks and priority inversion, ultimately making it challenging to build scalable software systems. However, transactions offer a more promising approach for composing software components. With transactions, multiple state-changing operations can be grouped and executed as a single atomic operation, simplifying concurrent programming and mitigating the issues associated with locks.

Concurrent programming can be simplified using a Software Transactional Memory System (STM) by grouping multiple operations that change the state and executing them as a single atomic operation, thus solving many problems associated with traditional concurrency models. Transactions in STM are optimistic in nature, which means that multiple transactions can execute concurrently. During the completion of each transaction, it is validated and committed only if no inconsistencies are found; otherwise, it is aborted. To ensure the correctness of the STM system, well-defined rules must exist that decide whether a transaction should be committed or aborted. Opacity is a common measure of correctness in STM, meaning that a STM system is considered correct if it satisfies Opacity, This refers to the process of ensuring that a transaction appears to execute as if it is atomic or not at all.

A memory transaction is a unit of code that runs in memory. In a software transactional memory system (STM), a transaction is executed as if it is atomic, even when there are other transactions executing concurrently. The STM ensures that a transaction either executes atomically or not at all. Once a transaction is executed to completion, its effects become visible to other transactions, and it is committed. However, if a transaction fails to execute to completion, it is aborted, and its effects are not visible to other transactions. This type of execution is referred to as optimistic execution.

To facilitate optimistic execution, transactions maintain a local log that records the values they read and write during their execution. When a transaction completes, it checks the contents of its log to ensure correctness. Nested transactions are useful for modular programming, where one transaction invokes another transaction as part of its execution. The implementation of nested transactions varies among databases, but they share a common feature where changes made within a nested transaction are not visible to unrelated transactions until the outermost transaction commits.

In closed nesting, when a sub-transaction commits, its effects are only visible to its parent transaction and its siblings, but not to other transactions.

The target of this project is to build an efficient STM by using serialization Graph testing(SGT) protocol to implement concurrency between transactions and within a transaction.

Nested Transaction:

Nested transactions occur when a transaction invokes another transaction during its execution. This provides transactional guarantees for a subset of operations performed within a larger transaction, allowing for independent committing or aborting of that subset. During nesting, the parent transaction is unable to perform any operations besides creating more child transactions or attempting to commit. Each child transaction creates a local log that becomes visible to the parent transaction upon the child's commit, but is not visible to any other transaction until the parent also commits. If a child transaction aborts, the system will not experience any state changes, but the associated local log will be deleted. Child transactions follow the abort or commit of the parent transaction; if the parent aborts, the child transactions also abort, and if the parent commits, the changes made by the child transactions are also committed. There is no limit to the depth of nesting as long as the system has sufficient memory available.
Nested transactions are categorises into two types: open nesting and closed nesting.

- In closed nesting, when a sub-transaction is committed, its effects can only be seen by its parent and siblings until the parent transaction is committed, after which the effects become visible to other transactions.
- On the other hand, unlike closed nesting, open nesting enables a transaction's effects to be visible to other transactions without delay.

System model:

In the context of computing, a transaction refers to a piece of code that can do read and write operations on memory and can also call upon other sub-transactions. This creates a computation tree, where the nodes represent read and write operations & transactions. The operations of a transaction can be classified into read and write operations and transaction operations (transactions themselves). When a transaction completes successfully, it ends with a commit operation, and if it fails, it ends with an abort operation. read and write operations always commit by default.

In a closed-nested transaction, all writes are done on a local buffer, and when it commits, the contents of its local buffer are merged with the parent's buffer. If it aborts, its local write values are not merged with its parent's buffers, and the writes of an aborted transaction are never visible to other transactions.

At the beginning of the system, if we assume that there is a root transaction t0 which calls all other transactions, then there will be a child transaction called t_{init} that sets all t_0's buffers to default values. Also, there will be another child transaction called t_{fin} in that reads the contents of t_0's buffers when the computation is completed.

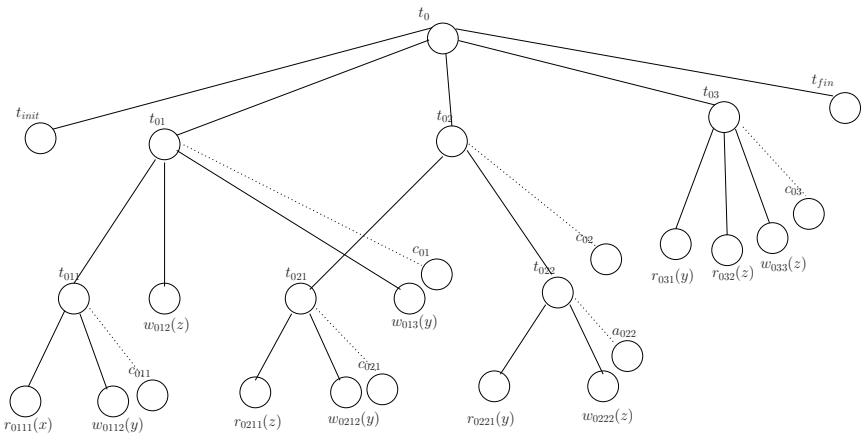

Figure 1: Hypothetical computation tree[1]

To implement an optimistic algorithm, we use several functions for each transaction. BeginTrans() is one such function that generates a new transaction whenever it is called by a process or another transaction. Another function, tryC(), is used to validate the transaction. If the consistency rules specified by the algorithm are satisfied, the transaction can write the calculated values to shared memory by acquiring appropriate locks on the data objects. If the validation succeeds, tryC() returns the value 'c', indicating that the transaction has committed. However, if the consistency requirements are not met, tryC() returns 'a', indicating that the transaction has been aborted. It's important to note that the database is not modified when transactions are aborted due to the optimistic execution approach.

Correctness criteria:

In a system utilizing STM (Software Transactional Memory), it is crucial that transactions, whether nested or not, that access the same data items are executed correctly when running concurrently. To ensure accurate execution in such situations, it is widely accepted that all transactions, including the aborted ones, should read consistent values. This means that the values obtained from any sequential execution of transactions should be consistent. To capture this requirement, Guerraoui and Kapalka introduced the concept of **"opacity"**, which demands a single equivalent serial execution of all committed and aborted transactions for any concurrent execution. In this process, only the read steps of aborted transactions are taken into account.

Opacity:

In the field of STM systems, opacity is a commonly utilized measure for validating the accuracy of the system. This is because it guarantees that all transactions, including the failed ones, read consistent values. An opaque concurrent execution schedule is one that has a comparable serial schedule that preserves the partial order of the original schedule, and for each read operation, it has the same last write operation as in some serial schedule.

Close nested opacity(CNO) :

The concept of opacity works well for straightforward transactions, but when dealing with nested transactions, we require a more advanced approach. In this project, we have implemented the concept of CNO (closed nested opacity), which is specifically designed to handle closed nested transactions.

If there is a serial schedule SS such that a given schedule S belongs to the Closed Nested Opacity (CNO) class, then the schedule S satisfies certain criteria, such that:[1]

1. The events in the schedule S and SS are equivalent. Formally (S.evts = SS.evts).
2. If in the computation tree represented by schedule S, two nodes n_Y and n_Z are peers, and n_Y occurs before n_Z in S, then in the serial schedule SS, n_Y will also occur before n_Z. This is known as the schedule-partial-order equivalence.
3. The equivalence of lastWrites in S and SS ensures that the read operations in both schedules have the same lastWrites.

Checking for opacity can be a challenging task, just like general serializability (e.g., view-serializability). However, specific classes of serializability, such as conflict-serializability, have been established. These classes are based on conflicts and have a polynomial membership test, making online scheduling more manageable. Similarly, a subclass of CNO, referred to as CP-CNO, can be defined using comparable methods.

Now in case of conflict serializability we have r-w, w-r and w-w conflicts similarly in case of closed nesting opt-conf are defined.But first we need to know about extOpsSet.
extOpsSet :A transaction's extOpsSet is made up of its external reads and commit writes.

In the context of memory operations in extOpsSets, optConf is a conflict notion that only applies between two peer nodes. Each node communicates with its peers through its extOpsSet. To illustrate, let's take two peer nodes n_A and n_B. If two memory operations, m_X and m_Y, are in the extOpsSets of n_A and n_B respectively, then S.optConf(m_X, m_Y) is true if m_X happens before m_Y in S and one of the following conditions is met:[1]

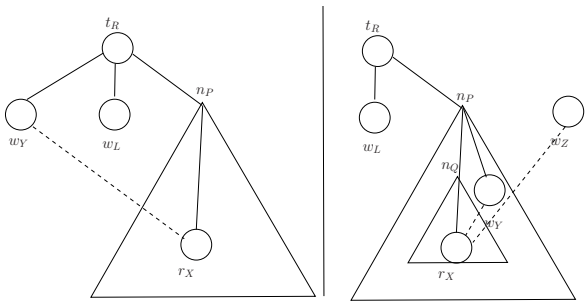

Figure 2: w-r and r-w conflicts[1]

1. W-r optConf: m_x is a conflict-write w_x of n_a and m_y is an external- read r_y of n_b.
2. R-w optConf: m_x is an external-read r_x in n_a , m_y is a commit-write w_y of n_b.
3. W-w optConf: m_x is a commit-write w_x of n_a and m_y is a commit-write w_y of n_b .

When taking into account the operation conflicts, the third requirement for opacity can be guaranteed, which states that if two memory operations in schedule S conflict optimistically (optConf), then they will also conflict optimistically in the corresponding serial schedule SS.

Modified Serialization Graph Testing Protocol :

The SGT protocol is a tool used to test serialization graphs, and it works by constantly updating a conflict graph based on the operations received by the scheduler. To ensure that the output adheres to the CSR property, the graph must always remain acyclic. Therefore, nodes and edges are inserted or deleted from the conflict graph in real-time based on the dynamic nature of the operations.
Now we modify the SGT algorithm for closed nesting as follows:[2]

1. If the operation $p_i(x)$ is the first one seen from transaction t_i, a new node is generated for t_i in the current graph G.
2. Edges of the form (t_j, t_i) are added to the graph G for each operation $q_j(x)$ that conflicts with $P_i(x)$, where $i \neq j$, and has been output before. There are two possible scenarios that can occur after this step.:
   (a) The graph G is cyclic, which means executing P_i (x) would result in a schedule that is not serializable. Therefore, P_i (x) is rejected, t_i is aborted, and the node corresponding to t_i and all its associated edges are removed from G.
   (b) If the graph G is still acyclic, then it is possible to output $p_i(x)$ by adding it to the previously output schedule. After that, the graph is tentatively updated and kept as the new current graph..
3. Each read step r(x) can read from a node in the computation tree provided it has committed and its effects can be seen. In general a read step can read a value from its parent n_p transaction or the peer transactions.
4. In the normal SGT algorithm we consider only committed transactions but in our modified SGT we consider the read operation from the Aborted transactions also.

We call this algorithm CN-SGT. We will define the data structures required and the pseudocode for the algorithm in subsequent sections let us first prove that this algorithm satisfies CP-CNO.

Correctness of modified SGT:

To show that the schedule generated by the CN-SGT algorithm always adheres to the Conflict-Serializability Property (CSR), we need to show that the precedence graph of the schedule is acyclic.
The precedence graph is defined as follows: There is a directed edge from transaction T_i to T_j if there exists an item X such that T_i performs a write operation on X before T_j performs a read or write operation on X, and T_i and T_j are not conflict-equivalent.
Let's consider the execution of the CN-SGT algorithm on a set of transactions T_1, T_2, ..., T_n. Initially, the conflict graph is empty. As each transaction Ti arrives, the algorithm adds a new node for Ti in the conflict graph if it is the first operation it sees from T_i. Then, for each previously executed transaction T_j that conflicts with T_i, the algorithm adds an edge from T_j to T_i in the conflict graph.
Now, there are two cases:

Case 1: The resulting conflict graph is acyclic. In this case, the algorithm outputs the operation $p_i(x)$ of transaction T_i, and the conflict graph remains the same for the next transaction. The precedence graph is a subset of the conflict graph and is therefore also acyclic. Hence, the resulting schedule adheres to the CSR property.

Case 2: The resulting conflict graph is cyclic. In this case, the algorithm aborts transaction T_i and removes its node and incident edges from the conflict graph. The algorithm proceeds to the next transaction with the updated conflict graph. Since the algorithm ensures that the conflict graph remains acyclic at all times, the resulting schedule adheres to the CSR property.
Therefore, we can conclude that the schedule generated by the CN-SGT algorithm always adheres to the CSR property.

The CN-SGT algorithm can be extended to handle sub-transaction calls by treating each sub-transaction as a separate transaction. When a sub-transaction call is encountered, the algorithm creates a new node in the conflict graph for the sub-transaction and proceeds to execute it as a separate transaction. Once the sub-transaction has completed, the algorithm updates the conflict graph and proceeds to the next operation in the parent transaction.
For non-layered transactions, the CN-SGT algorithm can be applied directly. Each non-layered transaction is treated as a separate transaction, and the algorithm proceeds as described in the original algorithm.
In both cases, the key to maintaining the CSR property is to ensure that the conflict graph remains acyclic at all times. The CN-SGT algorithm achieves this by constantly updating the conflict graph based on the dynamic nature of the operations and ensuring that the graph remains acyclic. By treating each sub-transaction as a separate transaction, the algorithm can handle nested transactions without violating the CSR property.

Overall, the CN-SGT algorithm can handle sub-transaction calls and non-layered transactions while maintaining the CSR property.

Datastructures and Headers :

At each step a transaction creates a list of data Items which will become visible to its sub- transactions and also the list of sub-transactions it will have.

$T_i \rightarrow$ read_list : It creates a list of transaction id's which read a particular data item 'x' from the transaction.
$T_i \rightarrow$counter : Keeps track of the no of sub-transactions created by the parent.
$T_i \rightarrow$localBuffer : It is used to store the all the values written by the transaction.
$T_i \rightarrow$transList : Creates a list of sub-transactions associated with it.
For simplicity we assume that the transaction id itself is the timestamp associated with the transaction.

Apart from that we have two maps:
Total_aborts= It keeps track of the no of aborts associated with each parent
Total_delay = Keeps track of the delay associated with each transaction.

Three variables denote the status symbols associated with each transaction.
'*abort*' : To denote a transaction is aborted.
'*commit*' : To denote a transaction is committed
'*live*' : To denote a transaction is scheduled.

Pseudocode :

Algorithm 1 STM ***initTrans()*** : Invoked by Transaction

```
1: // For the hypothetical transaction t0 create a set of transactions,t0 is defined as global pointer
2: DataItems = vector <shared_memory*>
3: transaction = map <Id,Transaction*>
4: counter=0
```

Algorithm 2 STM ***beginTrans()*** : Invoked by Top level Transaction

```
1: lock.transList
2: int Id = counter
3: counter ++
4: transaction[Id] = new Transaction[Id]
5: Add Transaction[Id] to T0 transaction's list
6: create a conflict graph 'G'
7: unlock.transList
```

Algorithm 3 STM ***beginSub(transaction * parent)*** : Invoked by Lower level Transaction

1: **lock**.parent →transList
2: *int* Id = parent→counter
3: parent→counter ++
4: transaction[Id] = new Transaction[Id]
5: Add Transaction[Id] to parent's transaction's list
6: **unlock**.parent→transList

Algorithm 4 STM ***read(transaction * parent, x)*** : A transaction reads the object x from it's parent's Buffer

1: **lock**.parent→x
2: **lock**.parent→G
3: **if** r_i(x) is the 1_{st} operation of transaction T_i **then**
4: Create a Node T_i in current graph G
5: **else if** r_i(x) is not First operation of transaction t_i **then**
6: For each previously output operation w_j(x) in conflict with r_i(x) where i is not equal to j, edges of the form (tj → ti) are added to G ;
7: **end if**
8: **check whether G is Cyclic or not**
9: **if** *isCyclic*(parent→G) **then**
10: transStatus ← **abort**
11: not allow r_i (x) and **abort** t_i , and the node for t_i and all edges associated with it.
12: **else**
13: Allow r_i(x) to read from parent's buffer
14: localBuffer ← (parent→x.value)
15: **end if**
16: **unlock**.parent→x
17: **unlock**.parent→G

Algorithm 5 STM ***write(transaction * parent, value)*** : A transaction is modifying the value on it's local buffer

1: **lock**.parent→G
2: **if** w_i(x) is the 1_{st} operation of transaction T_i **then**
3: Create a Node T_i in current graph G
4: **else if** w_i(x) is not First operation of transaction t_i **then**
5: For each previously output operation w_j(x) in conflict with r_i(x) where i is not equal to j, edges of the form (tj → ti) are added to G ;
6: **end if**
7: **check whether G is Cyclic or not**
8: **if** *isCyclic*(parent→G) **then**
9: transStatus ← **abort**
10: not allow w_i (x) and **abort** t_i , and the node for t_i and all edges associated with it.
11: **else**
12: localBuffer.value ← value
13: **end if**
14: **unlock**.parent→G

Algorithm 6 STM ***isCyclic(G)*** : Method for checking cycle after every memory operation

1: **for all** vertex V ∈ graph G **do**
2: visited[V]=False ;
3: finished[V]=False ;
4: **end for**
5: **for all** vertex V ∈ graph G **do**
6: DFS(V)
7: **end for**

Algorithm 7 STM ***DFS(V)*** : Method for DFS

```
1: if finished(V) then
2:     return
3: end if
4: if visited(V) then
5:     Cycle Found
6:     return 1
7: end if
8: visited(V)= True
9: for every neighbour of V W do
10:    DFS(W)
11:    finished(V)= True
12: end for
```

Algorithm 8 STM ***tryC(Id)*** : It tries to commit the transaction if commit make the values visible to the parent else abort

```
1: if transaction[Id].transStatus ≠ 'abort' then
2:     Merge transaction localBuffer with Parent's Buffer
3:     return "commit"
4: else
5:     return "abort"
6: end if
```

Algorithm 9 STM ***garbageCollection(G)*** :

```
1: lock.parent→G
2: for all vertex V ∈ graph G do
3:     if T_i.transStatus=="commit" && T_i is source node then
4:         Delete node T_i & edges associate with it
5:     end if
6: end for
7: unlock.parent→G
```

Conclusion :

There are many practical applications of nesting of transactions. In STM's it must be ensured that all transactions including the aborted transactions should read correct values for the correct concurrent execution of the system. In this project we have proposed a serialization graph testing based closed nested algorithm which follows optimistic execution. . We have also given the correctness proof of the algorithm by making sure that it satisfies CP-CNO

References

[1] Sathya Peri and Krishnamurthy Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *Distributed Computing and Networking: 12th International Conference, ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings 12*, pages 95–106. Springer, 2011.

[2] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.