International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 2 Issue: 11 3389 - 3397

A Flexible Framework For Implementing Multi-Nested Software Transaction Memory

[1]Dominic Damoah, [2]Roy Villafane, [3]Edward Ansong, [4]James B. Hayfron-Acquah[5]Henry Quarshie, [6]Lewis Selby Maxwell Jnr., [7]Shamo Sebastian

[13567]Department of Computer Science
Valley View University
Oyibi, Ghana
[4]Department of Computer Science
KNUST, Kumasi
Ghana
[2]Department of Computer Science and Engineering
Andrews University
Berrien Springs, Michigan, United States
Corresponding author: kwddamoah@yahoo.com

Abstract: Programming with locks is very difficult in multi-threaded programmes. Concurrency control of access to shared data limits scalable locking strategies otherwise provided for in software transaction memory. This work addresses the subject of creating dependable software in the face of eminent failures. In the past, programmers who used lock-based synchronization to implement concurrent access to shared data had to grapple with problems with conventional locking techniques such as deadlocks, convoying, and priority inversion. This paper proposes another advanced feature for Dynamic Software Transactional Memory intended to extend the concepts of transaction processing to provide a nesting mechanism and efficient lock-free synchronization, recoverability and restorability. In addition, the code for implementation has also been researched, coded, tested, and implemented to achieve the desired objectives.

Keywords: *Transaction Processing, Nested software transaction memory, Multi-Database Concepts, Concurrency Control, Two Phase Commit Protocol, Distributed Systems.*

*****

I. THE PROBLEM

The problem is to design a transactional framework that supports nested transactions in a single processing or multiprocessing environment and promotes concurrency, recoverability, and liveliness of the program, making sure that there are no deadlocks, indeterminate waiting, priority inversion, and obstructions, which are difficult to detect and solve and badly affect the excellent performance of transaction execution. Coupled with these problems that are normally difficult to achieve are militant problems such as larger storage overheads, synchronizing asynchronous transaction, and managing contention. This paper work explores efficient techniques to solving these problems without exacerbating them. It also explores the creation of a new framework that supports an advanced feature, such as nesting transactions. Among other things it is intended that this advanced transaction model for transaction execution will ensure correctness in multiple autonomous nested subsystems, enhance operational semantics on multilevel transactions and concurrency or parallelism, and improve user-defined or system-defined intra-transaction rollback, otherwise known as partial rollback for full recovery from failed situations.

This is to evaluate how advanced features of Software Transaction Memories (STM) out-perform the conventional programming with locks and how important properties pertaining to transaction processing in databases are implemented in software engineering mechanisms to maximize performance and fault tolerance.

This work is a continuation of earlier results reported on DSTM2 by [10]. It provides ample theoretical bases to justify the applicability of multilevel nested transactions in DSTM2.

This paper makes the following contributions: It explains why multilevel nested transactions are correct and efficient, and how nested transactions are dealt with in this environment. It develops a realistic framework model for nested transactional processing and execution on top of DSTM2 or other STMs.

Significant landmarks achieved here is that STMs can support such advanced feature as nesting. This is a giant contribution to the quest to optimize multiple processing cores in shared environments or distributed systems in nested transaction fashion with or without using locks and significantly affect how object-oriented-based STMs are implemented.

BASIC CONCEPTS OF SOFTWARE TRANSACTION MEMORY

I. SOFTWARE TRANSACTION PROCESSING

This sections discusses the fundamental concepts and theories of software transactional memory as a new paradigm especially in single and multiprocessing cores, what are transactions, how they provide correctness guarantees or fault tolerance and what practical limitations they have. This is followed by a segment that sets up the environment that no operation can take place without the call to atomicity. The next section talks about the intent and purpose of STMs and how they are used to achieve atomicity, concurrency, and recovery. It goes on to describe

the DSTM2 Library. This section provides the necessary fundamental issues not only for the rest of the material but also for design, development, and operations of the STMs.

Software Transaction Memory is a model for executing processes atomically and exactly once to synchronize processes that were otherwise asynchronous. There are no overhead costs in terms of memory access. Transactions can run concurrently and commit concurrently mostly when they have nothing in common or share the same memory locations. Transactions can easily update shared memory locations and instantiate objects.

Consider the normal conventional procedural method of executing the instruction:
Void foo () { x(): y(): }.It will be impossible to roll back changes made by x(), if y() fails in the cause of execution or throws an exception. The fact that x() has made an irreversible change poses difficult challenges. But if transactions were employed under this circumstance, the difficulty and overheads will be greatly reduced. At the failure of y(), it is aborted and x() is rolled back because it has not been committed. Hence restoring foo() to its original state is guaranteed and can be re-executed until it commits. The concept of transactions holds a powerful solution to software engineering problems with shared memory updates.

A recent study by [2] explains that a transaction is an atomic unit of work that is either completed in its entirety or not done at all in its scope and ensuring that all its shared resources are protected from multiple users. A transaction is a dynamic execution of a sequenceof operations, which should appear to execute instantaneously with respect to other concurrent transactions. For recovery purposes, systems need to keep track of when the transaction starts, terminates, and commits or aborts. Therefore a recovery manager keeps track of the following operations: begin_transaction, read or write, end_transaction, commit_transaction, and rollback or abort.

The state transition in a typical transaction execution describes how a transaction goes into activestate (BEGIN_TRANSACTION) immediately after it starts execution, where it can perform READ or WRITE operations. When the transaction ends (END_TRANSACTION), it moves to the partially committed state. At this point some recovery protocols need to ensure that a system failure will not result in an inability to permanently record the changes of the transaction. Once this check is successful, the transaction is said to have reached its commit point and enters the committed state (COMMIT_TRANSACTION). Optimistic concurrency control technique, known as validation or certification, requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions while the transaction was executing. Two transactions conflict if they issue operations that conflict. Transaction systems impose concurrency control to prevent conflicting transaction executions. Once a transaction is committed, it has concluded its execution successfully, and all its changes must be recorded permanently. However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during the active state. The transaction may have to be rolled back to undo the effect of the changes made (ROLLBACK OR ABORT). The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in the system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later, either automatically or after being resubmitted by the user as brand-new transactions.

II. PROPERTIES OF STMS

STMs have a lot of properties enforced by the principles of concurrency and recoverability, representing Atomicity, Consistency, Isolation, and Durability (ACID). Atomicity and Isolation properties will be discussed in detail because they hold important issues for this project. Reference [10] states the other characteristics as given briefly below:

1. In transactional programming the code that accesses shared memory is divided into transactions, and executed atomically.
2. Operations of two different transactions are not interleaved.
3. A transaction may commit or abort.
4. If two transactions conflict, then one must wait for the other either to commit or abort.
5. Aborted transaction may be typically retried until it commits when contention has been resolved [19].
6. A transaction must be consistent and preserving.
7. Changes applied after commit must persist to ensure recoverability.

III. ATOMICITY

A transaction is said to be atomic if it completes all its set of activities successfully (that is, commit) or if it fails (that is, abort) when all of its effects executed are undone. Atomicity features prominently in this project. Therefore, much time will be devoted to establish what it is and how much it brings to bear on the entire process.

The concept of atomicity is not original with this work, having been used extensively in database applications projects by [3] and [4]. These properties are borrowed from database concepts and adopted for software engineering purposes.

The quest to guarantee atomicity is a fundamental concept underpinning this application programming interface (API).

Two properties distinguish an activity as being atomic: indivisibility and recoverability. The usual method of providing indivisibility in the presence of concurrency and the one adopted is to guarantee serializability [3].

Reference [14] explains that actions are scheduled in such a way that their overall effect is as if they had been run sequentially in some order. To prevent one action from observing or interfering with the intermediate states of another action, we need to synchronize access to shared objects. In addition, to implement the recoverability property, we need to be able to undo the changes made to objects by aborted actions.

A. *Consistency*

The main purpose of running transactions on shared memory or data is to ensure constancy, reliability, and stability. If problems such as dirty read and non-repeatable reads are to be avoided, then consistency becomes an important issue to receive so much attention. According to [2], a transaction is consistency preserving, if its complete execution takes the database or the shared object from one consistent state to another (p. 620).

Ensuring consistency in concurrent programming is very hard to achieve, but that is the only way to reach the target of efficiency and improving response time. According to [15] consistency embraces reliability issues while coordinating concurrent transactions. The indivisibility of transactions ensures that consistent results are obtained even when requests are processed concurrently or failures occur during a request. Consistency improves stability, constancy, dependability, and reliability issues of transaction processing. The transactions must maintain the semantic and physical integrity of the data.

B. *Isolation*

Transactions are prone to dirty read, non-repeatable read, and a host of others if the isolation level is set so low. Put differently, transaction must execute independently to avoid unnecessary interferences from competing transactions running concurrently. Partial results are to be prevented from being altered or seen by other concurrent transactions. Lower levels of isolation allow other concurrent transactions to gain access to dirty reads, which make it very unsuitable for distributed systems where strict isolation ensures data integrity.

C. *Durability*

Durability ensures that nothing can cause updates to be lost once a transaction is committed. Changes made after the transaction is committed must persist to ensure that recoverability is possible. Committed changes must not be lost because of system failure and failed transactions.

IV. WHY SOFTWARE TRANSACTIONAL MEMORIES?

Parallel or concurrent execution of programs holds the prospects of optimization and efficiency but the question remains, how can consistency be maintained? How possible is it that we can consistently run two or more programs or threads to manipulate shared data without resorting to locks? An outstanding technique that provides an unprecedented solution is transaction memories. Transaction memories have incredibly wonderful solutions in both hardware and software applications [10]. Software transactional memory is an approach to solving this problem using software techniques. In transactional programming, the code that accesses shared memory can be grouped into transactions that are intended to be executed atomically: Operations of different transactions should not appear to be interleaved. A transaction is a single unit of work that can contain several programming steps that must execute collectively and successfully to ensure data integrity. Software transactional memory promotes liveliness of the program ensuring no deadlocks, indeterminate waiting or convoying, priority inversion, and obstruction so that there is the guarantee that there will be no performance issues rearing their ugly heads, which are extremely difficult to find and correct.

According to [4] conventional programming languages do not understand transactions and there is no easy way to fix that. When using transactions with savepoints it is important to understand that savepoints, like transactions themselves, provide the control and therefore the means for state restoration for only those components that understand transactions including savepoints. For example, if an application invokes a ROLLBACK function, its state will only partially be reestablished. Some components, such as database manager, may cooperate and fully support the protocol, but the memory manager and the run-time system of a conventional programming language ignore both transaction and savepoint completely. In other words, the database contents will return to the state as of the specified savepoint, but the local programming language variables will not. The reason for this is that conventional programming languages lack durability and persistence.

Reference [10] and [13],[17] have asserted that fine-grained locking has been used by programmers in multiprocessing programming environment instead of coarse-grained lock, which has poor concurrency and scalability. Also, if the locking protocol is not deadlock-free, deadlock detection of the locks must be considered to be part of lock maintenance overhead.

When executing in the shared environment, if the program crashes while in its critical section, an indeterminate waiting will keep other processes from acquiring locks and making progress. This situation is usually difficult to detect and solve easily. It is, therefore, a step in the right direction to avoid this condition by employing transaction semantics to ensure recoverability and avoid starvation of other processes. Transaction semantics guarantees that programs executing in their critical sections are obstruction free and will commit, with the net effect of making successful memory changes or rollback leaving memory intact. By extension, transactional semantics has an inherent recovery mechanism to revert to the original state when the process crashes and aborting the transaction becomes inevitable.This approach of getting over the problem of indeterminate waiting gives software transactional memories competitive urge over conventional programming with locks.

In multicore processing environments, where there is excessive use of locks on blocks of shared memory due to operations in critical sections. It is impossible to have share data run parallel as well as independently. They risk becoming serialized instead and introduce scalability concerns [13].

Programs become vulnerable to error as a result of sharing data or memory resources. Any undetected corrupt data that slip through tend to wreak havoc and more often than not they are extremely difficult to debug. Dirty-read problems are also present in transactions depending on the level ofisolation. Transactions possess an inherent advantage if the strict isolation is kept.

Conventional locking provides poor support for code composition and reuse. That is, if two or more tables methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. Exporting tables locks, then, compromises modularity and safety [10].

Finally, basic issues, such as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are based on convention, and violations are notoriously difficult to detect [13].

In summary, lock-based synchronization can lead to deadlock, makes fine-grained synchronization error-prone, precludes composition of atomic primitives, and provides no support for error recovery. Reference [22] explains that applications that use coarse-grained monitors may see limited scalability, since the execution of lock-protected monitors is inherently serialized. Application that use fine-grained locks, in contrast, generally provide good scalability, but see increased overheads and sometimes contain subtle bugs Transactional programming addresses these problems and provides a viable alternative synchronization [7] and [11]. For these reasons the transactional paradigm has evolved over the years to provide a viable alternative to conversional locking which is heavily laden with inherent problems extremely difficult to solve without necessarily aggravating them.

CONCEPTS OF NESTED SOFTWARE TRANSACTION FRAMEWORK

I. MULTILEVEL NESTED TRANSACTIONS

A transaction is said to be nested if and only if new transactions are commenced by instructions that are already inside an existing transaction. These new transactions or sub-transactions are said to be nested within existing transactions, which means they have multiple successors. This is in sharp contrast to the classical model called the flat transaction that has no internal structure or allows transactions to be embedded within other transactions. In the nested transaction model, transactions at the inner levels must commit before the outer ones can commit. This model transaction may contain any number of sub-transactions resulting in arbitrary deep hierarchy of nested transactions. Nested transaction hierarchies are a collection of nested spheres of control where the outermost sphere is formed by the top level transaction which incorporates the interface of the outside world [6]. The atomicity and isolation properties of transactions make it possible for the root or host transaction (the node without a predecessor) to register changes only when the nested transactions have committed. Nested transactions have the ability to roll back without any side effects. Innermost transactions are rolled back first, followed by the next nearest inner ones until the outermost one is reached. Transaction properties such as Atomicity, Isolation, Consistency, and Durability as mentioned earlier remain valid for nested transactions. That is, nested transactions which are atomic in nature with properly isolated execution are guaranteed to make meaningful progress to the point where they will be committed. In the event of failed transactions, a rollback is invoked to restore modified data in shared memory. Otherwise the transactions are committed and persistent updates in the shared memory are registered. Also transactions may terminate when root or top level transaction is committed or aborted. Though sub-transactions may commit independently because they appear isolated to other competing transactions, their success depends on parent transactions. They may abort with no side effects on surrounding transactions. Aborting any parent transaction will undo the effects committed by child transactions. The concept of nested and multilevel transactions is simplified when looked at as a general tree that can be traversed sequentially or concurrently as indicated in Figure 1.

This tree structure shows multilevel for parallel execution with nested transactions. For the root transaction TR to commit, all sub-transactions on the various nodes and at different levels must commit successfully. Likewise, transaction TR_1 can only commit when transaction $TR_{11,}$ transaction $TR_{12,}$ and transaction TR_{13} have succeeded sequentially or concurrently as a result of their subsequent sub-transaction such as TR_{121} and TR_{31} successfully committing respectively. Transactions at the same level follow the isolation property and are obstruction-free. They can run concurrently on different servers. It follows that consistency as applied in this distributed fashion is not as required as atomicity and isolation.

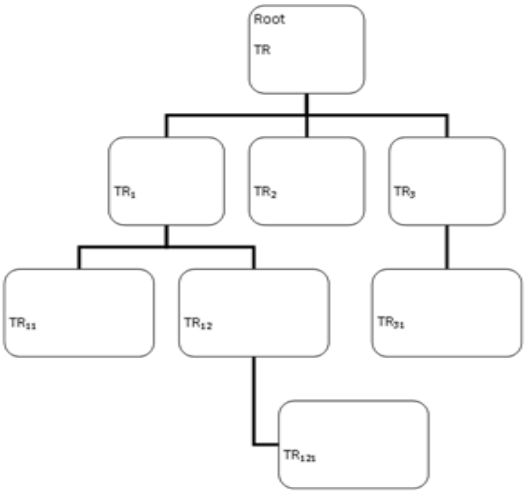

Figure 1 A general tree structure showing multilevel

A closed nested transaction proposed by [15] is viewed as the tree described in Figure 1 where the root transaction can nest descendant transactions any number of levels deep. Transaction nodes in the tree structure lend themselves to strict adherence to the ACID properties, the two-phase commit protocol, and the serializability paradigm. These transactions are well suited for short-lived transactions which are interdependent on one another and are synchronized in execution to produce consistent outcome. In this model, parent transaction inherits all locks of a committing child transaction. However, on conflict a child transaction can be aborted without aborting the parent, reducing the overall cost of an abort**.** It is important to note here that under no circumstances in closed nested models will a child's operations conflict with the operations of its parent or any of its ancestors.

On the other hand is the open nested transactions proposed by [15] and [1] usually used in long-lived transactions where the strict compliance to isolation at the global level, serializability and atomicity rules are relaxed to increased concurrency. Open-nested transactions offer higher concurrency than closed-nested or non-nested transactions and that allow child transactions to release locks early to avoid denial of service to other transactions.

Other nesting paradigms such as binary andlinear nesting have special restrictions on ancestor-descendant relationships for various reasons. Linear nested transaction describes strictly a single-parent single-child relationship, and binary-nested transactions allow a maximum of two children per parent. The policies of either closed or open nesting could be imposed on these other restrictive nesting paradigms. These sharply contrast the general tree structure depicted by the open-nested or closed-nested transactions where parent transactions are allowed to have an unlimited number of children. Reference [8] and [9] proposed sagas to use semantics knowledge to allow a transaction to release locks early and fully blend both short-lived and long-lived transactions.

II. GENERAL CHARACTERISTICS OF NESTED TRANSACTIONS

The term nested as used here suggests that a root or parent transaction can recursively be decomposed into sub-transactions. It follows that the root or parent transaction may have multiple children or sub-transactions. The root or parent transaction is successfully completed and permanently updated when the sub-transactions are committed because all other descendant transactions are also committed. This implies that if a child transaction fails, the parent is free to retry or try an alternative task, compensating transactions, to rescue the overall process or abort and override the failed transactions if they are not critical to the general success of that section of the transaction.

Nested transactions as we know are a concurrency scheme [15]. Needless to say they support top-level transactions with all of the ACID properties, in addition to supporting concurrent execution of independent actions within these transactions. They allow a topmost-level transaction to be the root of the tree for nested transactions. A transaction is serializable with respect to its siblings, that is, accesses to shared resources by sibling transactions have to obey the read-write and write-write synchronization rules. A transaction is a unit of recovery, that is, it can be aborted independently of its siblings. Nested transaction evolved from the requirement to allow transaction designers to design complex functionality decomposable transaction from the top down. A nested transaction model allows transaction services to be built independently and later combined into applications. Each service can determine the scope of its transaction boundaries. The application or service that orchestrates the combination of services controls the scope of the top-level transaction. It occurs when a new transaction is started on a session that is already inside the scope of an existing transaction. This new sub-transaction is said to be nested within or below the level of the existing transaction. Nested transaction allows an application to create a transaction that is embedded in an existing transaction [16].

III. ADDITIONAL PROPERTIES OF DESCENDANT TRANSACTIONS

A nested transaction's child actions are not considered to conflict with its parent's actions. Thus, it can lock a resource locked by its parent as long as none of its siblings have locked it. A nested transaction can lock a datum in some mode only if its parent has locked the datum in the same mode. A parent transaction's actions are considered not to conflict with its child's actions but not vice versa. Thus, it cannot access a resource if a child's lock prohibits the access. Thus, the child's lock wins. An abort by a child transaction does not automatically abort the parent transaction. The parent is free to try alternative compensating transactions or override it and try other nested transactions. A commit by a child transaction releases the locks held by it to its parent and makes its actions part of the action set of its parent transaction. Thus, when the parent commits, it commits not only those actions it performed directly but also those performed by its descendants.

IV. CONDITIONS FOR MULTILEVEL TRANSACTIONS

As noted by [5], the principles of multilevel transactions can be stated in three rules.

1. Abstraction hierarchy: objects of layer N are completely implemented by using operations of layer $N - 1$.
2. Discipline: there are no shortcuts from layer N to layers lower than $N - 1$.
3. Multilevel transactions rely on the existence of a compensation for each operation on any layer. Moreover, the compensations on layer $N - 1$ are scheduled by layer N or higher, which introduces a recovery dependency across layers.

V. SEMANTICS OF NESTED TRANSACTIONS

In summary, [12] state the following:

1. A parent can create children sequentially so that one child finishes before the next one starts. Alternatively, the parent can specify that some of its children can run concurrently. The transaction tree structure shown in Figure 1 does not distinguish between children that run concurrently or sequentially. A parent node does not execute concurrently with its children. It waits until all children in the same level are complete and then resumes execution. It may then decide to spawn additional children.
2. A sub-transaction and all of its descendants appear to execute as a single isolated unit with respect to its concurrent siblings. For instance, if TR1 and TR3 run concurrently, TR3 views the sub-tree TR1, TR11, and TR12 as a single unit of isolated transaction and does not observe its internal structure or interfere with its execution. The same applies to TR1 and the sub-trees TR3 and T31. Siblings are thus serializable, and the effect of their concurrent execution is the same as if they had executed sequentially in some serial order.
3. Sub-transactions are atomic. Each sub-transaction can commit or abort independently. The commitment and the durability of the effects of sub-transactions are dependent on the commitment of its parent. A sub-transaction commits and is made durable when all of its ancestors including the root transaction commit. At this point the entire nested transaction is said to have committed. If an ancestor aborts, then all of its descendants are aborted.
4. If a sub-transaction aborts, its operations have no effect. Control is returned to its parent, which may take appropriate action. In this way, an aborted sub-transaction may have an impact on the state of the shared

data as it may influence its parents to alter its execution path. This situation can be contrasted with flat transactions where an aborted transaction cannot alter the transaction path of the transaction coordinator.

5. A sub-transaction is not necessarily consistent. However, the nested transaction is consistent as a whole.

Table 1 gives a summary of activities demonstrating the semantics in the coordinator of the nested transactions.

VI. JUSTIFICATION FOR THE MULTILEVEL NESTED TRANSACTIONS

To set up the theoretical bases for the multilevel transactions it is imperative to throw more light on flat transactions implemented by DSTM2 supported by obstruction-free and shadow factories to establish the distinguishing factors.

When a flat transaction reaches the COMMIT point, its consistency rules put it beyond the reach of the system so that it cannot review its consistency constraints. In other words, the transaction is said to be consistent if it reaches the COMMIT and hence the outcome can be guaranteed. For flat transactions, inconsistencies in the data shared at COMMIT mean nothing can be done about it. This is largely true in flat transactions because control does not go beyond transaction boundaries. The flat transaction model of the DSTM2 is necessary in order to write reliable applications which share or require persistent or consistent data.

On the other hand, multilevel nested transaction is able to overcome this limitation because it is hierarchical. As shown in Figure 1, the levels in a nested transaction ensure that inconsistencies or dirty reads are not grossed over. The ACID property guarantees that if a sub-transaction fails, either it is resubmitted by a compensating transaction or entirely aborted to ensure consistency. The ability to roll back in this model to get the option of stepping back to an earlier start inside that same transaction gives it the competitive urge over the flat transactions. This implies that if one does not want to implement nested transactions, there must be alternative ways of informing the system about a state of the application program, for recovery purposes, so that the application can return to it later on if the transaction is aborted. This is what the proposed nested transactions model seeks to accomplish.

Thenested transactions model does not necessarily abort the whole transaction, as it is in the case of flat transactions, depending on how deep the failed transaction is nested. Support for nesting of transactions is essential for realizing the full potential of transactions.Other rescue policies, such as parent transactions overruling or superseding child transactions or running a compensating transaction by resubmitting the failed transaction, may be adopted to prevent a premature cessation of the entire process. The case of flat transaction cannot be guaranteed because it is simply impossible to integrate rescue policies in any way.

Table 1: The Summary of activities of the coordinator in the Nested Transaction

Coordinator	Children	Commit List	Provisional status	Abort List
TR	TR1, TR2, TR3	TR1, TR2, TR3, TR11,TR31	Commit/Yes	
TR1	TR11, TR12	TR11	Commit/Yes	TR12, TR121
TR2		TR2	Commit/Yes	
TR3	TR31,	TR3	Commit/Yes	
TR11		TR11	Commit/Yes	
TR12	TR121		Abort/ No	TR121
TR121			Abort (parent aborted)/ No	
TR31		TR31	Commit/Yes	

VII. IMPLEMENTATION OF THE NESTED TRANSACTIONS FRAMEWORK

Briefly this is how the implementation works. The user or client requests the framework to create a user object or transaction, usually the root transaction. The framework starts and populates the constructor with user inputs. It waits on the user to request more user objects to be created or added. This call can be repeated as many times as the user may require transactions for a specific or the intended application. After all the required objects are created and nested appropriately, the framework will request the result of the processing from the user objects so far created. The user objects return provisional ready to commit or abort signals to the framework. The resulting tree structure created from the nesting process is traversed depth first with control shifting from children transactions to parent transactions. All results are reported to the root transaction who informs the coordinator or the transaction manager to enter the second phase of the Two-Phase Commit Protocol to commit changes and move them to stable storage. It must be noted that this application allows descendant transactions to do dirty reads from ancestor transactions.

VIII. DETAIL SEMANTICS OF THE NESTED TRANSACTIONS FRAMEWORK

In this model for nested transactions the user starts off by creating instances of the NestedTransactionInterceptor( ) and the root transaction objects and sends the runRootTransaction( ) method with the root transaction object as a parameter to invoke the transactionCode( ) operations. The transactionCode( ) method in turn fires off the interceptorExecutor( ) method to embedd sub-transactions whose arguments specify the parent and child relationship. This helps to establish relationships between transactions and each transaction at this juncture executes its operations and returns a boolean result to the framework. If it returns true, it indicates that the internal operations of the transaction were successful and for that matter they have provisionally committed. On the other hand, if it returns false, it shows that the transaction failed and was provisionally aborted. It must be noted that if the aborted transactions were parents then all of their children or descendant transactions will be forced to abort. But where the sub-transactions are ready to commit they must wait for

the entire duration for the examination of the coordinator and for the root transaction to be committed before they are actually committed. See Figure 2.

New transactions are created which extend TransactionBase() and implement TransactionWrapper(). These transactions will also implement transactionCode behavior and if they return true, they are ready to commit, else they are ready to be aborted. The transaction programmer creates as many transactions as may be needed or is applicable under the peculiar circumstances of the application. Transactions are nested when an instance of the transaction InterceptorExecutor() is created and a message of the new transaction is passed to it to invoke its transactioncode() member function. It returns with a vote of whether it intends to commit or abort. A constraint worth noting is that the nestedTransactionInterceptor() cannot be used to create more than one tree structure of nested transactions.

```
class NestedTransactionInterceptor {

interceptorExecutor(TransactionWrapper tparent, TransactionWrapper
tchild) {
Start transaction; Status = Active;
if (tparent is not the root) {
                        Set parent child relationship
                        }
Get  childStatus from transaction code
        if (childStatus == true) {//Ready to commit
Change the status from Active to ready to commit
} else {
         Set ChildStatus to ABORTED
        /*An ancestral transaction failed to Commit. Hence descendant
        transactions are provisionally forced to abort by traversing the
        tree preorder Depth First Search;*/
                        }
interceptorStatus = childStatus;
End of transaction
return interceptorStatus;
}
```

Figure 2. The nested transaction interceptor coordinates transactions

Room is available for other business logic to be provided as compensation or as additional requirements for transactions to show willingness to commit or otherwise. Figure 3 shows a typical transactioncode() method for inserting a value into a list data structure. This is where the operations of the transactions are coded by the transaction programmer.

```
public class RootTransaction extends TransactionBase implements
TransactionWrapper {
public boolean transactionCode(NestedTransactionInterceptor c) {
boolean result = false;
final int value = 79;
result = Thread.doIt(new Callable<Boolean>() {
                        public Boolean call() {
                        return GlobalList.inSet.insert(value);
                        }
                        });
        return result;
        }
}
```

Figure 3. A typical transaction code method.

The runRootTransaction() checks to see if the root is ready to commit and if so calls the transactionCoordinator() method passing the root transaction as a parameter to it to execute the globalGroupCommit() to complete a group commit of the second phase of the 2PCP. The justification for the group commit is that when changes are made to protected resources, there must be a guarantee that the changes are made correctly. For instance, if a bank customer attempts to transfer money from a savings account to a checking account, there must be a guarantee that when the money is deducted from the savings account it is added to the checking account simultaneously. Partial completion of this transaction will have money deducted from the savings account but not added to checking account. This transfer transaction may involve several nested transactions that may be required if the transfer is to be made possible. If there are problems with descendant transactions required in order to successfully complete the overall transaction, then it is unwise to commit the root transactions. Rolling back all the nested transactions will undo changes before the root transaction can commit. This problem must be avoided, hence the need to do global or group commit to fully coordinate and manage nested transactions that are ready to be committed in transactional fashion. See Figure 4.

```
runRootTransaction(TransactionWrapper rootTransaction) {
Set rootStatus to false; // root transaction states its intention
Set  treeStatus  to false; // transaction tree was committed or aborted
get the status of the root transaction from the interceptor Executor
Execute rootStatus = interceptorExecutor(null, rootTransaction);
if (rootStatus == true) {
Call the transactions Coordinator to do global Commit
// if all goes well; but this might be false sometimes
treeStatus = true; // the tree was committed
//The second phase was successfully committed
//The tree was committed.
} else { treeStatus = false; // the tree was aborted
//The entire tree was aborted.
}
return treeStatus;
}
```

Figure 4. The second phase of the Two-Phase Commit Protocol.

The all-or-nothing principle is implemented on the second phase of the 2PCP. What is the implementation strategy? The implementation strategy adopted in this application is that, instead of committing the individual transactions and updating their status updaters, we create a global status updater and point each of the local status updaters to the global one and perform the group commit on it. Finally, point the status updater of the individual transactions to the local status updaters and change their values to commit. The strategy is illustrated in Figure 5.

```
transactionsCoordinator(TransactionWrapper
processedTransactions) {
if (childStatus = = true) {
//Transactions can be committed permanently"
Point Status Updater To StatusUpdateGlobalCommit
Perform a global Group Commit
Locally Commit
Point StatusUpdater To Local Status Updater
}
}
```

Figure 5. Globally committing provisionally committed transactions.

Log files and permanent storages are immediately updated and the participating transactions are duly informed about the changes. Otherwise the transactionDoAbort() is called to end the transaction and its descendants. The implication is that no updates or changes are made to log files and permanent storage. Instead the aborted transaction must undo its sequence of operations, by executing inverses of those operations in the reverse order to restore the state of any modified location to what it was before the transaction modified it. The log files are called into action or maintained for the purposes of recovery and fault tolerance.

The NestedTransactionExecutor() kicks in the coordinator otherwise known as the transaction manager which performs the 2PCP. All descendant transactions of the root transaction report their current states along with a tree structure except the descendants of aborted transactions. With all the sub-transactions in the tree provisionally committed, the coordinator has the yes votes from such individual transactions. This is where their statuses change from ACTIVE to READYTOCOMMIT to end the first phase of 2PCP. The coordinator then invokes the globalGroupCommit() to commit whatever is left in the tree to stable storage to complete the second phase of the 2PCP. At this stage the coordinator informs the participating transactions that the log files and databases have been updated accordingly.

When a child commits, its log is appended to its parent's log. When a child is forced to abort or is aborted because it failed, all of its actions and updates on the log files are undone and discarded. As mentioned before, aborting a child does not affect its parents or ancestors, though the parent must be informed to take a firm decision on making progress to committing or aborting itself.

The user closes the set of nested transactions by invoking the endTransaction() or abort() or transactionDoAbort() sub-routines on the root transaction or on a specific sub-transaction.

The rules and policies on isolation and concurrency enforced on failed transactions in this nested transaction model are either relaxed or made more intense. Strict isolation rules, as in a single system with or without internal structure, permit sub-transactions at the same level to be run concurrently with additional rules to grant locks.

However, ancestor transactions do not run concurrently with their descendant transactions because they must remain atomic and serializable with other sibling transactions. For that matter, individual transactions in the sub-transaction act independently. They are isolated and atomic, strictly enforcing the all-or-nothing principles in their own rights. Reference [16] confirms that the concurrency control scheme introduced by the closed-nested transaction model guarantees isolated execution for sub-transactions and that the schedules of concurrent nested transaction are serializable.

On the other hand with the rules on isolation at the global level relaxed on failed transactions, room is created for a long-lived transaction to be accommodated in this model. Reference [1] has proposed several extensions to the closed-nested transactions to increase concurrency and throughput by relaxing the consistency and isolations rules of a typical traditional transaction model.

In this nested transaction model parent transactions reserve the prerogative to make progress in the event of descendant transactions failing. This guarantees that long-lived transactions are not put on hold forever and much work completed at the time of failure is not lost. Much of the data resources locked is released when transactions are aborted to prevent the starvations of other transactions that might need those resources.

It is interesting to note that this proposed nested model combines the strengths of both closed and open nesting and a blend of synchronized transactions that can be automatically rolled back to immediate parent transaction, implement the 2PCP on top of it and extensions that override failed transactions.

IX. CONCLUSIONS AND RECOMMENDATIONS

The concept of transactions has been employed in software engineering to permit management of activities and resources in a reliable computing environment tolerating faults. Transactions are able to guarantee efficient software engineering techniques, bringing consistency and concurrency into applications in the face of eminent failure. Nested transactions allow fine-grained control over serializability, concurrency, and recovery.Programmers need not concern themselves with the complexity of deciding how to best allocate locks, deadlocks, livelocks, priority inversion, locking granularity, and other typical lock-based programming issues[20].

The flat transaction model proposed by the DSTM2 is necessary in order to write reliable short-lived applications that share or require persistent or consistent data. Nevertheless, the new advanced feature of nesting transactions proposed by the paper supports enough flexibility and performance for complex transactions and long-lived transactions, making it suitable and applicable in single or multicore or even distributed systems. For most of these systems, much will depend on the policies adopted for the failed or aborted transactions.

In a typical nested transaction, the outermost or root transaction is not aborted because one of the sub-transactions failed in the course of execution. The failure of a transaction does not necessarily lead to the failure of the root transaction, and until the root transaction is committed no durable state changes are made to the shared resource. For these reasons, it can conveniently be concluded that there are no requirements for failure recovery mechanisms. Since the effects of the multilevel nested transaction are provisional upon commit or abort of the root transaction, the effects are easily recovered if the root transaction aborted even though the descendant transactions committed.

As far as contention management is concerned, access rights acquired by parents are inherited by children transactions and executed depth first sequentially or guaranteed serializability for concurrent transactions to avoid possible conflicts among sibling transactions.In order to maximize performance, several transction memory implementations included mechanisms allowing the programmer to specify whether transaction memory should ignore certain conflicts[21].

This multilevel nested transaction that is hereby proposed has two main advantages over flat transactions.

First, this model allows the potential internal consistent parallelism and nesting transactions many levels deep to be exploited. Second, it provides finer control over failures by limiting the effects of failures to a small part of the global transaction. These properties are achieved by allowing nested transactions within a given transaction to fail independently of their invoking transactions. Changes made by the nested transaction, when it is committed, remain contingent upon commitment of all of its ancestors [18].

Finally more than one section of the tree could be created and traversed concurrently or sequentially in this multilevel nested transaction model. Isolation rules ensure that the result of each concurrent tree's provisionally committed or aborted descendant transactions are not visible to other transactions, except their parent transaction to maintain consistencies throughout the lifetime of the transactions.

For further study I recommend extending this framework to support conditional waiting and integration into the DSTM2 model or other similar models.

REFERENCES

[1] Elmagarmid, A. K. (1991). Transaction models for advanced database application. THE INDIANA CENTER FOR DATABASE SYSTEMS DEPARTMENT OF COMPUTER SCIENCE, PURDUE UNIVERSITY, W. LAFAYETTE, IN.

[2] Elmasri, R., & Navethe, S. B. (2007). Fundamentals of database systems (5th ed.).

[3] Eswaran, K.P., Gray, J.N., Lorie, R.A., & Traiger, I.L. (1976, November). The Notions Of Consistency and Predicate Locks in a Database System. Communications of the ACM, 19(11), 624-633.

[4] Gray, J., McJones, P., Blasgen, M., Lorie, R., Price, T., Putzolu, F., & Traiger, I. (1981, June). The recovery manager of the system R database manager. ACM Computing Surveys, 13(2), 223-242.

[5] Gray, J., & Reuter, A. (1993). Transaction processing: concepts and techniques. Morgan Kaufmann. San Francisco, CA.

[6] Haerder, T., & Rothermel, K. (1987). Concepts for transaction recovery in nested transactions. ACM Sigmod Record, 16(3), 239-248.

[7] Harris, T.L., Marlow, S., Peyton J., & S., Herlihy, M.(2005). Composable memory transactions. PPoPP 2005. Principles and practice of parallel programming, pages 48–60, New York, NY, USA.

[8] Garcia-Molina H. (1983). Using semantic knowledge for transaction processing in a distributed database. ACM Trans. Database Syst.,8(2):186–213.

[9] Garcia-Molina H., & Salem, K. SAGAS.(1987) SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data.

[10] Herlihy, M., Luchangco, V., & Moir, M. (2006). A flexible framework for implementing software transactional memory. ACM Sigplan Notices, 41(10), 253-262.

[11] Herlihy, M. & Moss, J.E.B.(1993) Transactional memory: architectural support for lock-free data structures. ISCA. International symposium on computer architecture, pages 289–300, New York, NY, USA.

[12] Kifer, M., Bernstein, A. & Lewis, P M. (2005). Database system: Application-oriented Approach (2nd ed). Addition-Wesley.

[13] Kung, H. T., & Robinson, J. T. (1981, June). On optimistic methods for concurrency control. Pittsburgh: Carnegie-Mellon University.

[14] Liskov, B., & Scheifler, R. (1983, July). Guardians and actions: Linguistic support for robust, distributed programs. ACM Transactions on Programming Languages and Systems, 5(3), 381–404.

[15] Moss, E. B. (1981). Nested transactions: An approach to reliable distributed computing. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

[16] Papazoglou, M. P. (2008). Web Services: Principles and Technology. Prentice Hall Publications. Tilburg University, The Netherlands.

[17] Saha, B., Adl-Tabatabai, A., Hudson, R. L., Minh, C.C., Hertzberg, B., (2006). McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime, Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming: pages187 – 197.

[18] Saheb, M., Karoui, R., & Sédillot, S. (1999). Open nested transaction: A support for increasing performance and multi-tier applications. Le Chenay Cedex, France: Institut National Recherché En Informatique Et En Automatique.

[19] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott,(2004) Design Tradeoffs in Modern Software Transactional Memory Systems? http://web.cse.ohio-state.edu/~agrawal/788-su08/Papers/week4/design_tradeoffs_modern_stm.pdf

[20] Alexander Hogue, Software Transactional Memory and the Rotate- Free Tree (2012).

[21] M. Couceiro, P. Romano, Where does Transactional Memory research standand what challenges lie ahead?WTM 2012, EuroTM Workshop on Transactional Memory, IST/INESC-ID, Lisbon, Portuga (2012)

[22] Richard M. Yoo, Sandhya Viswanathan, Vivek R. Deshpande, Christopher J. Hughes, Shirish Aundhe, Early Experience on Transactional Execution of JavaTM Programs Using Intel R Transactional Synchronization Extensions, TRANSACT 2014 9th ACM SIGPLAN Workshop on Transactional Computing (2014) Salt Lake City, Utah, USA (co-located with ASPLOS 2014)