Correctness of Concurrent Executions of Closed Nested Transactions in Transactional Memory Systems

Sathya Peri*[1] and K.Vidyasankar[2]

[1] Indian Institute of Technology Patna, India sathya@iitp.ac.in
[2] Memorial University, St John's, Canada vidya@mun.ca

Abstract. A generally agreed upon requirement for correctness of concurrent executions in Transactional Memory systems is that all transactions including the aborted ones read consistent values. Opacity is a recently proposed correctness criterion that satisfies the above requirement. Our first contribution in this paper is extending the opacity definition for closed nested transactions. Secondly, we define conflicts appropriate for optimistic executions which are commonly used in Software Transactional Memory systems. Using these conflicts, we define a restricted, conflict-preserving, class of opacity for closed nested transactions the membership of which can be tested in polynomial time. As our third contribution, we propose a correctness criterion that defines a class of schedules where aborted transactions do not affect consistency of the other transactions. We define a conflict-preserving subclass of this class as well. Both the class definitions and the conflict definition are new for nested transactions.

1 Introduction

In recent years, Software Transactional Memory (STM) has garnered significant interest as an elegant alternative for developing concurrent code. Importantly, transactions provide a very promising approach for composing software components. Composing simple transactions into a larger transaction is an extremely useful property which forms the basis of modular programming. This is achieved through nesting. A transaction is nested if it is invoked by another transaction.

STM systems ensure that transactions are executed atomically. That is, each transaction is either executed to completion in which case it is *committed* and its effects are visible to other transactions or *aborted* and the effects of a partial execution, if any, are rolled back. In a *closed* nested transaction [2], the commit of a sub-transaction is local; its effects are visible only to its parent. When the top-level transaction (of the nested-computation) commits, the effects of the sub-transaction are visible to other top-level transactions. The abort of a sub-transaction is also local; the other sub-transactions and the top-level transaction are not affected by its abort. [3]

To achieve atomicity, a commonly used approach for software transactions is optimistic synchronisation (term used in [6]). In this approach, each transaction has local buffers where it records the values read and written in the course of its execution. When

* This work was done when the author was a Post-doctoral Fellow at Memorial University
[3] Apart from Closed nesting, Flat and Open nesting [2] are the other means of nesting in STMs.

the transaction completes, the contents of its buffers are validated. If the values in the buffers form a consistent view of the memory then the transaction is committed and the values are merged into the memory. If the validation fails, the transaction is aborted and the buffer contents are ignored. The notion of buffers extends naturally to closed nested transactions. When a sub-transaction is invoked, new buffers are created for all the data items it accesses. The contents of the buffers are merged with its parent's buffers when the sub-transaction commits.

A commonly accepted correctness requirement for concurrent executions in STM systems is that all transactions including aborted ones read consistent values. The values resulting from any serial execution of transactions are assumed to be consistent. Then, for each transaction, in a concurrent execution, there should exist a serial execution of some of the transactions giving rise to the values read by that transaction. Guerraoui and Kapalka [5] captured this requirement as *opacity*. An implementation of opacity has been given in [8].

On the other hand, the recent understanding (Doherty et al [3], Imbs et al [7]) is that opacity is too strong a correctness criterion for STMs. Weaker notions have been proposed: (i) The requirement of a single equivalent serial schedule is replaced by allowing possibly different equivalent serial schedules for committed transactions and for each aborted transaction, and these schedules need not be compatible; and (ii) the effects, namely, the read steps, of aborted transactions should not affect the consistency of the transactions executed subsequently. The first point refines the consistency notion for aborted transactions. (All the proposals insist on a single equivalent serial schedule consisting of all committed transactions.) The second point is a desirable property for transactions in general and a critical point for nested transactions, where the reads of an aborted sub-transaction may prohibit committing the entire top-level transaction. The above proposals in the literature have been made for non-nested transactions.

In this paper, we define two notions of correctness and corresponding classes of schedules: *Closed Nested Opacity (CNO)* and *Abort-Shielded Consistency (ASC)*. In the first notion, read steps of aborted (sub-)transactions are included in the serialization as in opacity [5, 8]. In the second, they are discarded. These definitions turn out to be non-trivial due to the fact that an aborted sub-transaction may have some (locally) committed descendents and similarly some committed ancestors.

Checking opacity, like general serializability (for instance, view-serializability), cannot be done efficiently. Very much like restricted classes of serializability allowing polynomial membership test, and facilitating online scheduling, restricted classes of opacity can also be defined. We define such classes along the lines of conflict-serializability for database transactions: *Conflict-Preserving Closed Nested Opacity (CP-CNO)* and *Conflict-Preserving Abort-Shielded Consistency (CP-ASC)*. Our conflict notion is tailored for optimistic execution of the sub-transactions and not just between any two conflicting operations. We give an algorithm for checking the membership in CP-CNO which can be easily modified for CP-ASC as well. The algorithm uses serialization graphs similar to those in [12]. Using this algorithm an online scheduler implementing these classes can be designed.

We note that all online schedulers (implementing 2PL, timestamp, optimistic approaches, etc.) for database transactions allow only subclasses of conflict-serializable schedules. We believe similarly that all STM schedulers can only allow subclasses of conflict-preserving schedules satisfying opacity or any of its variants. Such schedulers are likely to use mechanisms simpler than serialization graphs as in the database area. An example is the scheduler described by Imbs and Raynal [8].

There have been many implementations of nested transactions in the past few years [2, 10, 1, 9]. However, none of them provide precise correctness criteria for closed nested transactions that can be efficiently verified. In [2], the authors provide correctness criteria for open nested transactions which can be extended to closed nested transactions as well. Their correctness criteria also look for a single equivalent serial schedule of both (read-only) aborted transactions and committed transactions.

Roadmap: In Section 2, we describe our model and background. In Section 3, we define CNO, CP-CNO In Section 4, we present ASC and CP-ASC Section 5 concludes this paper.

2 Background and System Model

A transaction is a piece of code in execution. In the course of its execution, a nested transaction performs read and write operations on memory and invokes other transactions (also referred to as sub-transactions). A compuation of nested transactions constitutes a *computation tree*. The operations of the computation are classified as: *simple-memory operations* and *transactions*. Simple-memory operations are *read* or *write* on memory. In this document when we refer to a transaction in general, it could be a top-level transaction or a sub-transaction. Collectively, we refer to transactions and simple-memory operation as nodes (of the computation tree) and denote them as n_{id}.

If a transaction t_X executes successfully to completion, it terminates with a *commit* operation denoted as c_X. Otherwise it *aborts*, a_X. Abort and commit operations are called *terminal operations* [4]. By default, all the simple-memory operations are always considered to be (locally) committed. In our model, transactions can interleave at any level. Hence the child sub-transactions of any transaction can execute in interleaved manner.

To perform a write operation on data item x, a closed-nested transaction t_P creates a x-buffer (if it is not already present) and writes to the buffer. A buffer is created for every data item t_P accesses. When t_P commits, it merges the contents of its local buffers with the buffers of its parent. Any peer (sibling) transaction of t_P can read the values written by t_P only after t_P commits.

We assume that there exists a hypothetical root transaction of the computation tree, denoted as $\mathbf{t_R}$, which invokes all the top-level transactions. On system initialization we assume that there exists a child transaction t_{init} of $\mathbf{t_R}$, which creates and initializes all the buffers of $\mathbf{t_R}$ that are written or read by any descendant of $\mathbf{t_R}$. Similarly, we also assume that there exists a child transaction t_{fin} of $\mathbf{t_R}$, which reads the contents of $\mathbf{t_R}$'s buffers when the computation terminates.

[4] A transaction starts with a begin operation. In our model we assume that the begin operation is superimposed with the first event of the transaction. Hence, we do not explicitly represent it in our schedules.

Coming to reads, a transaction maintains a read set consisting of all its read operations. We assume that for a transaction to read a data-item, say x, (unlike write) it has access to the buffers of all its ancestors apart from its own. To read x, a nested sub-transaction t_N starts with its local buffers. If it does not contain a x-buffer, t_N continues to read the buffers of its ancestors starting from its parent until it encounters a transaction that contains a x-buffer. Since $\mathbf{t_R}$'s buffers have been initialized by t_{init}, t_N will eventually read a value for x. When the transaction commits, its read set is merged with its parent's read set. We will revisit read operations a few subsections later.

2.1 Schedules

A *schedule* is a totally ordered sequence (in real time order) of simple-memory operations and terminal operations of transactions in a computation. These operations are referred to as *events* of the schedule. A schedule is represented by the tuple $\langle evts, nodes, ord\rangle$, where $evts$ is the set of all events in the schedule, $nodes$ is the set of all the nodes (transactions and simple-memory operations) present in the computation and ord is an unary function that totally orders all the events in the order of execution. Example 1 shows a schedule, $S1$. In this schedule, the memory operations $r_{2211}(x)$ and $w_{2212}(y)$ belong to the transaction t_{221}. Transactions t_{22} and t_{31} are aborted. All the other transactions are committed. It must be noted that the t_{221} and t_{222} of t_{22} are committed sub-transactions of the aborted transaction t_{22}.

Example 1.
$S1 : r_{111}(z)w_{112}(y)w_{12}(z)c_{11}r_{211}(b)r_{2211}(x)w_{2212}(y)c_{221}w_{212}(y)c_{21}w_{13}(y)c_1$
$r_{2221}(y)w_{2222}(z)c_{222}a_{22}w_{23}(z)r_{311}(y)c_2w_{312}(y)a_{31}r_{321}(z)w_{322}(z)c_{32}c_3$

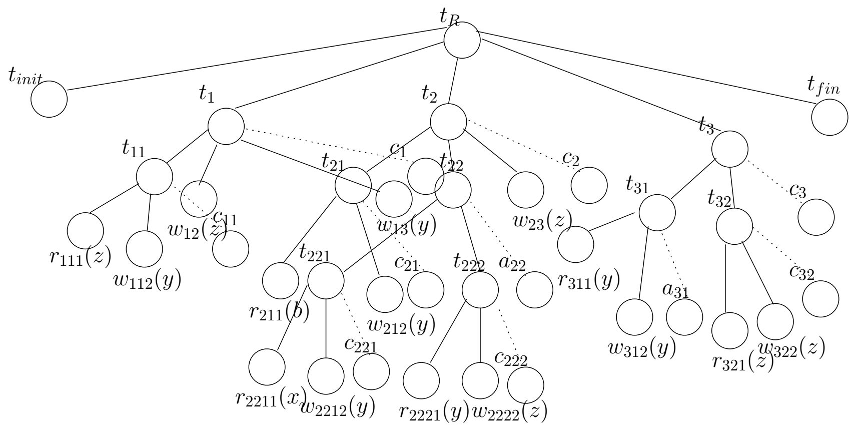

Fig. 1. Computation tree for Example 1

The events of the schedule are the real time representation of the leaves of the computation tree. The computation tree for a schedule $S1$ is shown in Figure 1. The order of execution of memory operations is from left to right as shown in the tree. The dotted edges represent terminal operations. The terminal operations are not part of the computation tree but are represented here for clarity.

For a closed nested transaction, all its write operations are visible to other transactions only after it commits. In $S1$, $w_{212}(y)$ occurs before $w_{13}(y)$. When t_1 commits, it writes $w_{13}(y)$ onto $\mathbf{t_R}$'s y-buffer. But t_2 commits after t_1 commits. When t_2 commits it overwrites $\mathbf{t_R}$'s y buffer with $w_{212}(y)$. Thus when transaction t_{31} performs the read operation $r_{311}(y)$, it reads the value written by $w_{212}(y)$ and not $w_{13}(y)$ even though $w_{13}(y)$ occurs after $w_{212}(y)$.

To model the effects of commits clearly, we augment a schedule with extra write operations. For each transaction that commits, we introduce a commit-write operation for each data item x the transaction writes to or one of its children commit-writes. This writes the latest value in the transaction's x-buffer. The commit-writes are added just before the commit operation and represent the merging of the local buffers with its parent's buffers. Using this representation, the schedule for Figure 1 is:

Example 2.
$S2 : r_{111}(z)w_{112}(y)w_{12}(z)w_{11}^{112}(y)c_{11}r_{211}(b)r_{2211}(x)w_{2212}(y)w_{221}^{2212}(y)c_{221}w_{212}(y)$
$w_{21}^{212}(y)c_{21}w_{13}(y)w_{1}^{12}(z)w_{1}^{13}(y)c_{1}r_{2221}(y)w_{2222}(z)w_{222}^{2222}(z)c_{222}a_{22}w_{23}(z)r_{311}(y)$
$w_{2}^{21}(y)w_{2}^{23}(z)c_{2}w_{312}(y)a_{31}r_{321}(z)w_{322}(z)w_{32}^{322}(z)c_{32}w_{3}^{32}(z)c_{3}$

Some examples of commit-write in $S2$ are $w_{11}^{112}(y), w_{21}^{212}(y), w_{2}^{23}(z)$ etc. The commitwrite $w_{11}^{112}(y)$ represents t_{11}'s write onto t_1's y buffer with the value written by w_{112}. There are no commit-writes for aborted transactions. Hence the writes of aborted transactions are not visible to its peers. Originally in the computation tree only the leaf nodes could write. With this augmentation of transactions, even non-leaf nodes (i.e. committed transactions) write with commit-write operations. For sake of brevity, we do not represent commit-writes in the computation tree. In the rest of this document, we assume that all the schedules we deal with are augmented with commit-writes.

Generalizing the notion of commit-writes to any node of the tree, the commit-write of a simple-memory write is itself. It is nil for a read operation and aborted transactions. Collectively we refer to simple-memory operations along with commit-write operations as memory operations. With commit-write operations, we extend the definition of an operation, denoted as o_X, to represent a transaction, a commit-write operation or a simple-memory operation.

It can be seen that a schedule partially orders all the transactions and simple-memory operations in the computation. The partial order is called *schedule-partial-order* and is denoted $<_S$. For a transaction t_X in S, we define $S.t_X.first, S.t_X.last$ as the first and last operations of t_X. Thus, $S.t_X.last$ denotes the terminal operation of the t_X. For a simple-memory operation m_X, $S.m_X.first = S.m_X.last$. For two nodes n_X, n_Y, in S: $(n_X <_S n_Y) \equiv (S.ord(S.n_X.last) < S.ord(S.n_Y.first))$.

2.2 Function Definitions

For a commit-write operation w_X we define its *holder*, $S.holder(w_X)$ as the transaction t_X to which it belongs to. Extending this function to a node (a transaction or simple-memory operation), the holder of a node is itself. For any operation o_X, we define $S.level(o_X)$ as the distance of $S.holder(o_X)$ in the tree from the root. From this definition $\mathbf{t_R}$ is at level 0. The level of a transaction and all its commit-write operations are the same. For instance in Example 2, $S2.level(w_{21}^{212}) = S2.level(t_{21}) = 2$.

The functions on a tree, namely parent, children, ancestor, descendant, peer (siblings) can be extended to commit-write operations by defining them for $S.holder(o_X)$ over the tree. For instance in $S2$ of Example 2, $S2.parent(w_2^{21}) = t_R$ and $S2.children(w_2^{21} = \{t_{21}, t_{22}, w_{23}(z)\}$. Thus transactions and simple-memory operations are children of a transaction. Two commit-writes of the same node are not peers of each other since they have the same holder.

For a transaction, t_X in a computation, we define its *dSet*, denoted as $S.dSet(t_X)$ as the set consisting of t_X, t_X's commit-writes, t_X's begin and terminal operations and dSets of t_X's descendants (including its children). This set comprises of all the operations in the sub-tree of t_X. A simple-memory operation's dSet is itself. A commit-write's dSet is its holder transactions's dSet. In Example 2, $S2.dSet(t_2) = S2.dSet(w_2^{23} (z)) = \{t_2, t_{21}, t_{22}, w_{23}(z), r_{211}(b), w_{212}(y), w_{21}^{212}(y), c_{21}, t_{221}, r_{2211}(x), w_{2212}(y), w_{221}^{2212}(y), c_{221}, t_{222}, r_{2221}(y), w_{2222}(z), w_{222}^{2222}(z), c_{222}, a_{22}, w_2^{21}(y), w_2^{23}(z), c_2\}$

Next we define a boolean function *optVis* on two operations o_X, o_Y in a schedule S, denoted as $S.optVis(o_Y, o_X)$. It is true if o_Y is a peer of o_X or peer of an ancestor of o_X, i.e., $o_Y \in (S.peers(o_X) \cup S.peers(S.ansc(o_X)))$. Otherwise it is false. This definition implies that if $(o_X \in S.dSet(o_Y))$ then $S.optVis(o_Y, o_X)$ is false. As a result for any commit-write of o_Y, say w_Y, $S.optVis(w_Y, o_X)$ is false as well. One can see that optVis function is not symmetric (but not asymmetric). Hence $S.optVis(o_Y, o_X)$ does not imply $S.optVis(o_X, o_Y)$. In $S2$, $S2.optVis(w_1^{12}(z), r_{211}(b))$ is true as $w_1^{12}(z)$ is a peer of t_2 which is an ancestor of $r_{211}(b)$. Similarly $S2.optVis(t_3, r_{2221}(y))$ is true because t_3 is a peer of t_2 which is an ancestor of $r_{2221}(y)$. But $S2.optVis(r_{2221}(y), t_3)$ is false.

We denote $S.schOps(t_X)$ as the set of operations in $S.dSet(t_X)$ which are also present in $S.evts$. Formally, $S.schOps(t_X) = (S.dSet(t_X) \cap S.evts)$. We define a few notations based on aborted transactions in a schedule S. For an aborted transaction t_X, we denote $S.abort(t_X)$ as the set of all aborted transactions in t_X's dSet. It includes t_X as well, if it is aborted. We define $S.prune(t_X)$ as all the events in the schOps of t_X after removing the events of all aborted transactions in t_X. Formally, $S.prune(t_X) = \{S.schOps(t_X) - (\bigcup\limits_{t_A \in S.abort(t_X)} S.schOps(t_A))\}$. If t_X has no aborted transaction in its dSet then $S.prune(t_X)$ is same as $S.schOps(t_X)$. If t_X itself is an aborted transaction then its pruned set is nil.

2.3 Writes for Read Operations and Well-Formedness

For a read operation $r_X(z)$ belonging to a transaction t_P in S, we associate a write $w_Y(z)$ as its *lastWrite* [5] or $S.lastWrite(r_X(z))$. The read operation will retrieve the value written by the lastWrite. We want the lastWrite $w_Y(z)$ to satisfy the properties: (1) w_Y occurs before r_X in the schedule; (2) w_Y is optVis to r_X; (3) The value written

[5] This term is inspired from [2]

by w_Y is in the z-buffer of an ancestor (starting from its parent t_P) closest to r_X in terms of level and (4) If there are multiple writes satisfying the above conditions then, the w_Y is closest to r_X in the schedule S.

The lastWrite definition ensures that all transactions read values only from committed nodes i.e. a committed transaction or a simple-write operation. Having lastWrite be optVis to the read operation ensures that the buffer in which the lastWrite writes to is accessible by the read operation. In $S2$, the lastWrites are: $(r_{111}(z) : w_{init}(z)), (r_{211}(b) : w_{init}(b)), (r_{2211}(x) : w_{init}(x)), (r_{2221}(y) : w_{221}^{2212}(y)), (r_{311}(y) : w_1^{13}(y)), (r_{321}(z) : w_2^{23}(z))$. Note that the read $r_{2221}(y)$ reads from $w_{221}^{2212}(y)$ even though $w_1^{13}(y)$ is closer to $r_{2221}(y)$ in the schedule. This is because $w_{221}^{2212}(y)$ is closer to it in terms of level.

For a node n_P with a read operation r_X in its dSet, the read is said to be an *external-read* if its lastWrite is not in n_P's dSet. Thus a read operation r_X is an external-read of itself. It can be seen that a nested transaction interacts with its peers through external-reads and commit-writes. Thus, a nested transaction can be treated as a non-nested transaction consisting only of its external-reads and commit-writes. The external-reads and commit-writes of a transaction constitute its *extOpsSet*.

A schedule is called *well-formed* if it satisfies: (1) Validity of Transaction limits: After a transaction executes a terminal operation no operation (memory or terminal) belonging to it can execute; and (2) Validity of Read Operations: Every read operation reads the value written by its lastWrite operation.

We assume that all the schedules we deal with are well-formed.

2.4 Serial Schedules

For the case of non-nested transactions a serial schedule is a schedule in which all the transactions execute serially (as the name suggests) without any interleaving. For a nested transaction we define a serial schedule SS as: for every transaction t_X in SS, its children (both transactions and simple-memory operations) are totally ordered. Formally, $\langle \forall t_X \in SS.trans : \{n_Y, n_Z\} \subseteq S.children(t_X) : (SS.ord(n_Y.last) < SS.ord(n_Z.first)) \vee (SS.ord(n_Z.last) < SS.ord(n_Y.first))\rangle$. Thus in a serial schedule, all the events in the dSet of a transaction appear contiguously.

3 Conflict Preserving Closed Nested Opacity

3.1 Closed Nested Opacity

Guerraoui and Kapalka [5] proposed the notion of *opacity* as a correctness criterion for software transactions. A schedule, consisting of an execution of transactions, is said to be *opaque* if there is an equivalent serial schedule such that it respects the original schedule's real time ordering of the nodes and the lastWrites for every read operation, including the reads of aborted transactions, in the serial schedule is same as in the original schedule. Opacity ensures that all the reads are consistent. An implementation of opacity for non-nested transactions is given in [8] in which aborted transactions are treated as read-only (with read steps executed before the abort) when looking for an equivalent serial schedule consisting of all the transactions.

In the context of nested transactions, an aborted transaction can have a committed sub-transaction whose values are read by other sub-transactions. For instance in $S2$, aborted transaction t_{22}'s sub-transactions t_{221} and t_{222} are committed. The read operation $r_{2221}(y)$ of t_{222} reads from t_{221}. This shows that some writes of aborted (sub) transactions should also be considered for correctness of other sub transactions. On the other hand, a committed transaction can have aborted sub-transactions whose write values should be omitted.

In our characterization of schedules, aborted transactions do not have commit-writes. Thus an aborted transaction's writes do not affect any of its peers or ancestors. But committed sub-transactions of an aborted transactions can have commit-writes and other sub-transactions can read from it. Thus using our representation, opacity can be extended to closed nested transactions. Formally, we define a class of schedules called as *Closed Nested Opacity* or *CNO* as follows: A schedule S belongs to *CNO* if there exists a serial schedule SS such that: (1) Event Equivalence: The operations of S and SS are the same. (2) schedule-partial-order Equivalence: For any two nodes n_Y, n_Z that are peers in the computation tree represented by S, if n_Y occurs before n_Z in S then n_Y occurs before n_Z in SS as well. (3) lastWrite Equivalence: The lastWrites of all read operations in S and SS are the same.

Even though the definition of CNO is similar to opacity, the condition lastWrite equivalence captures the intricacies of nested transactions. This class ensures that the reads of all the transactions including all the sub-transactions of aborted transactions read consistent values.

3.2 Conflict Notion: optConf

Checking opacity, like general serializability (for instance, view-serializability) cannot be done efficiently. Restricted classes of serializability (like conflict-serializability) have been defined based on conflicts which allow polynomial time membership test, and facilitate online scheduling. Along the same lines, we define a subclass of CNO, *CP-CNO*. This subclass is defined based on a new conflict notion *optConf* for closed nested transactions. It is tailored for optimistic execution of sub-transactions. This notion is similar to the idea of conflicts presented in [4] for non-nested transactions.

The conflict notion optConf is defined only between memory operations in extOpsSets (defined in SubSection2.3) of two peer nodes. As explained earlier, a node (or transaction) interacts with its peer nodes through its extOpsSet. Consider two peer nodes n_A, n_B. For two memory operations m_X, m_Y on the same data buffer in the extOpsSets of n_A, n_B, $S.isOptConf(m_X, m_Y)$ is true if m_X occurs before m_Y in S and one of the following conditions hold: (1) r-w conflict: m_X is an external-read r_X of n_A, m_Y is a commit-write w_Y of n_B or (2) w-r conflict: m_X is a commit-write w_X of n_A and m_Y is an external-read r_Y of n_B or (3) w-w conflict: m_X is a commit-write w_X of n_A and m_Y is a commit-write w_Y of n_B.

Consider a read r_X that is in optConf with a write w_Y and let r_X's lastWrite be w_L. By defining the conflicts in this manner we ensure that w_L is in w-r conflict with r_X and if w_Y is also in w-r conflict with r_X, then w-w conflict between w_L and w_Y ensures that w_Y does not become r_X's lastWrite in any optConf equivalent serial schedule.

Similarly if w_Y is in r-w conflict with r_X then it cannot become r_X's lastWrite in the equivalent serial schedule.

For $S2$ in Example 2, we get the set of conflicts as: $(r_{111}(z), w_{12}(z)), (r_{111}(z), w_2^{23}(z)), (w_{11}^{112}(y), w_{13}(y)), (w_1^{12}(z), w_2^{23}(z)), (w_1^{13}(y), w_2^{21}(y)), (w_{221}^{2212}(y), r_{2221}(y)), (w_1^{13}(y), r_{311}(y)), (r_{311}(y), w_2^{21}(y)), (r_{311}(y), w_{312}(y)), (w_1^{12}(z), r_{321}(z)), (w_2^{23}(z), r_{321}(z)), (r_{321}(z), w_{322}(z)))$. It must be noted that there is no optConf between $w_1^{13}(y)$ and $r_{2221}(y)$ or between $w_{21}^{212}(y)$ and $r_{2221}(y)$ even though $w_1^{13}(y)$ and $w_{21}^{212}(y)$ are optVis to $r_{2221}(y)$. This is because the $w_{221}^{2212}(y)$'s level (which is the lastWrite of $r_{2221}(y)$) is greater than $w_1^{13}(y)$ and $w_{21}^{212}(y)$. Hence $r_{2221}(y)$ is not an external-read of any peer of $w_1^{13}(y)$ or $w_{21}^{212}(y)$

Using optConf, we define a class of schedules called as *Conflict-Preserving Closed Nested Opacity* or *CP-CNO*. It differs from CNO in condition (3) in SubSection3.1. The lastWrite equivalence is replaced by optConf Implication: if two memory operations in S are in optConf then they are also in optConf in SS. Since optConf implication subsumes lastWrite equivalence, we have:

Theorem 1. *If a schedule S is in the class CP-CNO then it is also in CNO.*

Benefits of optConf: Traditionally, two memory operations are said to be in conflict if one of them is a (simple) write operation. In STM systems that employ optimistic synchronization, a write of a transaction becomes visible only after it has committed. In this case for conflicts to be meaningful, two memory operations are said to be in conflict if one of them is a commit-write operation (and not a simple-write). Refining the conflict notion further, we define optConf only between an external-read and a commit-write operation (as well as between two commit-write operations). By defining optConf this way, the class CP-CNO is as non-restrictive as possible and yet does not compromise on any desired property.

3.3 Membership Verification Algorithm

Now, we describe the algorithm for testing the membership in the class CP-CNO in polynomial time. Our algorithm is based on the graph construction algorithm by Resende and Abbadi [12] but adapted to optConf. For a schedule S, the algorithm constructs a conflict graph based on optConfs, denoted as $S.optGraph$, and checks for the acyclicity of that graph. We call this *optGraphCons algorithm*. The graph $S.optGraph$ is constructed as follows: (1) Vertices: It comprises of all the nodes in the computation tree. The vertex for a node n_X is denoted as v_X. (2) Edges: Consider each transaction t_X starting from $\mathbf{t_R}$. For each pair of children n_P, n_Q, (other than t_{init} and t_{fin}) in $S.children(t_X)$ we add an edge from v_P to v_Q as follows: (2.1) Completion edges: If $n_P <_S n_Q$. (2.2) Conflict edges: For any two memory operations, m_Y, m_Z such that m_Y is in n_P's dSet and m_Z is in n_Q's dSet, if $S.isOptConf(m_Y, m_Z)$ is true.

Since the position of the transactions t_{init} and t_{fin} are fixed in the tree and in any schedule, we do not consider them in our graph construction algorithm. Now, we get the theorem,

Theorem 2. *For a schedule S, the graph S.optGraph is acyclic if and only if S is in CP-CNO.*

It must be noted that in our construction all the edges are between vertices corresponding to peer nodes. There are no edges between vertices that correspond to nodes of different levels. Thus the graph constructed consists of disjoint subgraphs. If the graph is acyclic, then an equivalent serial schedule can be constructed by executing topological sort on all the subgraphs [11]. Using this algorithm, it can be verified that $S2$ is not in CP-CNO. Further, $S2$ is also not in CNO.

4 Abort-Shielded Consistency

Shortcoming of CNO: A single serial schedule involving all transactions, as required in CNO (and opacity) allows for the reads of an aborted transaction to affect the transactions that follow it. This effect is more pronounced in nested transactions. For instance in $S2$, transactions t_1 and t_2 write to the variables y and z. The aborted sub-transaction t_{31} reads y from t_1. The sub-transaction t_{32} reads z from t_2. As a result there is no equivalent serial schedule having the same lastWrites as in $S2$ and hence it is not in CNO. For that matter, any sub-transaction of t_3 invoked after t_{31}'s invocation (like t_{33}, t_{34} etc) that reads any variable written by t_2 that has also been written by t_1 will cause this schedule to be not opaque. In the worst case, all the sub-transactions of t_3 invoked after t_{31} may satisfy this property and a scheduler (implementing CNO) will abort all of them. This effectively aborts t_3. This shows that with CNO, an aborted sub-transaction can cause its top-level transaction to abort. This can be avoided if the read operations of the aborted transactions are ignored as described below.

4.1 ASC Class Definition

Let t_A be an aborted transaction in a schedule S. If t_A should not affect the transactions following it, then t_A should be dropped while considering the correctness of the remaining transactions. Generalizing this idea to all aborted transactions, we construct a sub-schedule consisting of events only from committed transactions (and committed sub-transactions whose ancestors have not been aborted). Thus, the sub-schedule consists of all the events from $S.prune(\mathbf{t_R})$ (prune is defined in SubSection2.2) and is denoted as $commitSubSch_R$. The ordering of the events is same as in the original schedule. We check for the correctness of $commitSubSch_R$. The sub-schedule $commitSubSch_R$ for $S2$ is: $r_{111}(z)w_{112}(y)w_{12}(z)w_{11}^{112}(y)c_{11}r_{211}(b)w_{212}(y)w_{21}^{212}(y)c_{21}w_{13}(y)w_1^{12}(z)$ $w_1^{13}(y)c_1w_{23}(z)w_2^{21}(y)w_2^{23}(z)c_2r_{321}(z)w_{322}(z)w_{32}^{322}(z)c_{32}w_3^{32}(z)c_3$.

As explained in [5], it is necessary that the aborted transaction t_A also reads consistent values. To ensure this, we construct another sub-schedule of S denoted as $pprefSubSch_A$ (pruned prefix sub-schedule) for t_A. We consider the prefix of all the events until t_A's abort operation. From this prefix we construct the sub-schedule by removing (1) events from transactions that aborted earlier and (2) events from any aborted sub-transaction of t_A. Thus, the sub-schedule consists of events from transactions that committed before t_A, events from pruned sub-transactions of t_A and events from live transactions (i.e., transactions that have not yet terminated) that executed until abort of t_A. The ordering among the events is same as in the original schedule S.

Finally, for each live transaction we add a commit operation after t_A's abort operation to the sub-schedule. But we do not add the commit-writes for these transactions. Then we look for the correctness of this sub-schedule. In $S2$, for the aborted transaction t_{31}, $pprefSubSch_{31}$ is: $r_{111}(z)w_{112}(y)w_{12}(z)w_{11}^{112}(y)c_{11}r_{211}(b)w_{212}(y)w_{21}^{212}(y)c_{21}$ $w_{13}(y)w_{1}^{12}(z)w_{1}^{13}(y)c_{1}w_{23}(z)r_{311}(y)w_{2}^{21}(y)w_{2}^{23}(z)c_{2}w_{312}(y)a_{31}c_{3}$. Similarly the sub-schedules for every aborted transaction can be constructed.

Here all the sub-schedules have events from at most one aborted transaction. One can see that the sub-schedules $commitSubSch_R$ and $pprefSubSch_A$ for every aborted transaction t_A have the property that if any event is in the sub-schedule, then any other event that is relevant to it is also in the sub-schedule. We call this property as *causality completeness*. Hence the lastWrite for any read operation in a sub-schedule will be same as the lastWrite as in the original schedule S. It can also be seen that the events of these sub-schedules form a valid sub-tree of the original computation tree represented by S. We verify the correctness of each of these sub-schedules by looking for an equivalent serial sub-schedule which has the same lastWrite for every read operation.

Based on these sub-schedules, *Abort-Shielded Consistency* or *ASC* is defined. A schedule S belongs to class ASC if there exists a set of sub-schedules of S, denoted as $subSchSet$, such that the sub-schedules, $commitSubSch_R$ and $prefSubSch_A$, for every aborted transaction t_A in S, are in $subSchSet$ and for every sub-schedule $subS$ in $subSchSet$ there exists a serial sub-schedule $ssubS$ such that: (1) Sub-Schedule Event Equivalence: The operations of $subS$ and $ssubS$ are the same (2) schedule-partial-order Equivalence: For any two peer nodes n_Y, n_Z in the computation tree represented by $subS$, if n_Y occurs before n_Z in $subS$ then n_Y occurs before n_Z in $ssubS$ as well. (3) lastWrite Equivalence: For all the read operations in $ssubS$, the lastWrites are the same as in $subS$.

From this definition we get that, CNO is a subset of ASC. The schedule $S2$ is in ASC. Using optConfs with pprefSubSch we define a class of schedules, *Conflict-Preserving Abort-Shielded Consistency* or *CP-ASC*. It differs from the definition of the class ASC only in the condition (3), which is optConf Implication: If two memory operations in $subS$ are in optConf then they are also in optConf in $ssubS$. Using the optGraphCons algorithm we can verify if there exists an equivalent serial sub-schedule for each sub-schedule in subSchSet. Thus checking whether a schedule is in CP-ASC or not, can be done in polynomial time [11]. Further, it can as well be proved that the class CP-CNO is a subset of CP-ASC. The schedule $S2$ is in CP-ASC.

Using the optGraphCons algorithm an elegant online scheduler implementing CP-ASC can be designed [11]. The scheduler can be implemented in a completely distributed manner. The serialization graph has separate components for each (parent) sub-transaction. Each component can be maintained at a different site (process executing the sub-transaction) autonomously and the checking can be done in a distributed manner.

5 Conclusion

Concurrent executions of transactions in Transactional Memory are expected to ensure that aborted transactions, as the committed ones, read consistent values. In addition, it is desirable that the aborted transactions do not affect the consistency for the other transactions. Incorporating these simple-sounding criteria has been non-trivial even for non-nested transactions as can be seen in recent publications [5, 8, 3].

In this paper, we have considered these requirements for closed nested transactions. We have also defined new conflict-preserving classes that allow polynomial time membership test, by means of constructing conflict-graphs and checking acyclicity. Further, a completely distributed STM scheduler can be designed using these conflict preserving classes. Our future work includes the study of how the above two properties manifest in executions with open nested transactions and with non-transactional steps.

References

[1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 70–81, New York, NY, USA, 2006. ACM.

[3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. In *REFINE*, 2009.

[4] Rachid Guerraoui, Thomas Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *DISC '08: Proc. 22nd International Symposium on Distributed Computing*, pages 305–319, sep 2008. Springer-Verlag Lecture Notes in Computer Science volume 5218.

[5] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[6] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[7] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 280–281, New York, NY, USA, 2009. ACM.

[8] Damien Imbs and Michel Raynal. A lock-based stm protocol that satisfies opacity and progressiveness. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 226–245, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] J.E.B.Moss. Open Nested Transactions: Semantics and Support. *In Workshop on Memory Performance Issues*, 2006.

[10] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM.

[11] Sathya Peri and K.Vidyasankar. Correctness criteria for closed nested transactions (in preperation). Technical report, Memorial University of Newfoundland, 2010.

[12] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Inf. Process. Lett.*, 50(4):177–183, 1994.