An Efficient Scheduler for Closed Nested Transactions that satisfies All-Reads-Consistency and Non-Interference

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. We denote this as all-reads-consistency. Opacity is a correctness criterion that satisfies the above requirement. A relevant property, which we call as non-interference, is that an aborted transaction should not affect the consistency of the transactions that are executed subsequently. This property is desirable in general and critical for closed nested transactions in the sense that the read steps of an aborted sub-transaction (that were executed before aborting) may make committing its top-level transaction impossible. Recently we proposed a new correctness criterion, Abort Shielded Consistency, that satisfies both all-reads-consistency and non-interference. In this paper, we present an efficient on-line scheduler that implements Abort Shielded Consistency. The scheduler is based on the notion of conflicts which have been appropriately defined for optimistic executions. The scheduler maintains a conflict-graph based on the events executed so far. An event is allowed to be executed only if it does not cause a cycle in the graph. The conflict-graph has separate components for each (parent) sub-transaction. Each component can be maintained autonomously at the site executing the sub-transaction and the checking can be done in a distributed manner.

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 invokes another transaction as a part of its execution.

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

* This work was done when the author was a Post-doctoral Fellow at Memorial University

rolled back. In a *closed* nested transaction [1], the commit of a sub-transaction is local; its effects are visible only to its parent level. 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 synchronization. In this approach, each transaction has local buffers where it records the values read and written in the course of its execution. When 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 local 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. We denote this requirement as *all-reads-consistency*. The values resulting from any serial execution of transactions are assumed to be consistent. Guerraoui and Kapalka [4] captured this requirement as *opacity*. An implementation of opacity has been given in [6].

Weaker notions of correctness have been proposed for STM systems (Doherty et al [3], Imbs et al [5]) : (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. The second point, which we call as *non-interference*, is a desirable property for transactions in general and a critical point for nested transactions, where the effects of an aborted sub-transaction may prohibit committing the entire top-level transaction.

In [9, 10], we defined a correctness criterion, *Abort-Shielded Consistency (ASC)*, for closed nested transactions that satisfies both the above mentioned properties: all-reads-consistency and non-interference. The class definition turns out to be non-trivial due to the fact that an aborted sub-transaction may have some committed descendants and similarly some committed ancestors. It also turns out that this class is a superset of opacity. In this paper, we present an online scheduler that implements ASC.

Checking opacity and the more general ASC cannot be done efficiently, like general serializability (for instance, view-serializability). In the context of databases, restricted (conflict preserving) classes of serializability have been defined which allow polynomial membership test and thus facilitate online scheduling of transactions. Along the same lines a restricted class of ASC, *Conflict-Preserving Abort-Shielded Consistency (CP-ASC)* can be defined [9, 10]. The conflict notion used in CP-ASC is tailored for optimistic execution of the sub-transactions and not just between any two conflicting operations.

[3] Apart from Closed nesting, Flat and Open nesting [1] are the other means of nesting in STMs.

The algorithm that we have presented in this paper specifically implements CP-ASC. The algorithm maintains a conflict-graph based on the events executed so far. An event is allowed to be executed only if it does not cause a cycle in the graph. The conflict-graph has separate components for each (parent) sub-transaction. Each component can be maintained autonomously at the site executing the sub-transaction and the checking can be done in a distributed manner.

There have been many implementations of nested transactions in the past few years [1, 8, 7, 2]. However, none of the systems proposed satisfy both all-reads-consistency and non-interference properties. Nor they provide precise correctness criteria for closed nested transactions that can be efficiently verified and implemented.

Roadmap: In Section 2, we describe our model and background. In Section 3, we present the class ASC which satisfies the properties all-reads-consistency and non-interference and the conflict preserving subclass CP-ASC. In Section 4 we present the detailed design of an online scheduler that implements CP-ASC. In Section 5 we give the outline of the proof. Section 6 concludes this paper and Section 7 gives the pseudocode.

2 Background and System Model

2.1 Notations

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 computation 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. Collectively, we refer to transactions and simple-memory operation as nodes (of the computation tree).

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*. By default, all the simple-memory operations are always considered to be committed.

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 writes to. 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 it 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.

A *schedule* S captures the execution of a computation. It 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, $S.evts$.

For a closed nested transaction, all its write operations are visible to other transactions only after it commits. To effectively capture a computation in a schedule we use model defined by us in [9, 10]. In addition to simple-memory operation and terminal operations (which are part of the computation tree), a schedule is augmented with extra write operations called commit-writes. For each transaction that commits, we introduce a commit-write operation for each data item x that 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, a sample schedule is:

Example 1.
$S1 : r_{111}(x)w_{112}(y)w_{11}^{112}(y)c_{11}w_{12}(z)r_{211}(z)w_{212}(y)w_{21}^{212}(y)c_{21}w_{13}(y)w_{1}^{12}(z)w_{1}^{13}(y)$
$c_1r_{221}(x)w_{222}(z)a_{22}w_{2}^{21}(y)c_2r_{31}(y)r_{32}(z)w_{33}(z)w_{3}^{33}(z)c_3$

This schedule consists of t_1, t_2, t_3 as top level transactions. The operation c_1 represents the commit of t_1. The child transaction of t_1 is represented as t_{11}. The operations $r_{111}(x), w_{112}(y)$ represent read and write operations of transaction t_{11} on data items x and y respectively. The operations $w_{11}^{112}(y), w_{21}^{212}(y), w_{3}^{33}(z)$ etc are commit-writes. 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 operation 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 either a transaction or a commit-write operation or a simple-memory operation.

The functions on a tree, namely parent, children, ancestor, descendant and peer can be extended to commit-write operations by defining them for the node to which commit-write belongs to. For instance, consider a node with n_X with a commit-write $w_X(d)$. Then peers of $w_X(d)$ are the peers of n_X on the tree.

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 descendants, the descendant's commit-writes and the terminal operation of t_X and its descendants terminal operations. This set comprises of all the operations in the sub-tree of t_X. A simple-memory operation's dSet is itself.

We denote $S.schOps(t_X)$ as the set of operations in $S.dSet(t_X)$ which are also present in $S.evts$. 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. 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.

A schedule SS is said to be *serial* if for every transaction t_X in SS, its children (both transactions and simple-memory operations) are totally ordered. There is no interleaving amongst the transactions in a serial schedule at any level.

2.2 LastWrites for Read Operations and OptConf Conflict notion

To read a data-item x a nested sub-transaction t_N starts with its local buffers. If it does not contain a x-buffer, t_N reads 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.

In a schedule S, an operations o_Y is said to be optVis to o_X, denoted as $S.optVis(o_Y, o_X)$, if o_Y is a peer of o_X or peer of an ancestor of o_X. For a read operation $r_X(z)$ in S, we associate a write $w_Y(z)$ as its *lastWrite* [4] or $S.lastWrite(r_X(z))$ which satisfies the properties: (1) w_Y occurs before r_X in the schedule. (2) w_Y is optVis to r_X. (3) The value of w_Y is from the z-buffer of the ancestor closest to r_X (starting from t_P). (4) If there are multiple writes satisfying the above conditions then, the w_Y is closest to r_X in the schedule S.

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.The external-reads and commit-writes of a transaction constitute its *extOpsSet*.

Given a schedule, a conflict function optConf [9, 10] can be defined between memory operations of the schedule. The conflict notion optConf is defined only between memory operations in extOpsSets of two peer nodes. Consider two peer nodes n_A, n_B. For two memory operations m_X, m_Y 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.

3 Abort-Shielded Consistency

A single serial schedule involving all transactions, as required in opacity [4] allows for the reads of an aborted transaction to affect the transactions that follow it. Thus opacity does not satisfy non-interference property which can greatly limit concurrency specifically in the context of nested transactions. *Abort-Shielded Consistency* or *ASC* is a correctness condition that satisfies both all-reads-consistency and non-interference [9, 10].

The class ASC is defined using multiple sub-schedules. Consider a schedule S. For S, $commitSubSch_R$ is a sub-schedule consisting of all the events from $S.prune(\mathbf{t_R})$. It consists of events only from committed transactions. For each aborted transaction t_A in S, a sub-schedule, $ppref SubSch_A$ (pruned prefix sub-schedule), can be defined. To construct the $ppref SubSch_A$, the prefix of S containing all the events until t_A's abort operation is considered. From this prefix, the sub-schedule is constructed by removing (1) events from transactions that aborted earlier and (2) events of any aborted sub-transaction of t_A. The ordering among the events in these sub-schedules is same as in the original schedule S.

[4] This term is inspired from [1]

Based on these sub-schedules, the class ASC is defined. For a schedule S a set of sub-schedules of S, denoted as $subSchSet$, is defined such that the sub-schedules, $commitSubSch_R$ and $ppref SubSch_A$, for every aborted transaction t_A in S, are in $subSchSet$. A schedule S belongs to class ASC if 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 $subS$. The schedule $S1$ is in ASC.

Testing for ASC's membership, 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. Along the same lines, a subclass of ASC, *Conflict-Preserving Abort-Shielded Consistency* or *CP-ASC* is defined using the conflict notion optConf [9, 10]. Its definition is very similar to ASC but differs from it 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 $ssubS$.

4 CP-ASC-Sched: An online scheduler based on CP-ASC

In [9, 10] we developed *optGraphCons algorithm* for verifying if a schedule S has an optConf equivalent serial schedule or not. The algorithm constructs a conflict graph based on optConfs for S, denoted as $S.optGraph$, and checks for the acyclicity of that graph. 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 terminates before n_Q start. (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.

The algorithm can be used for CP-ASC membership verification [9, 10] as: for all the sub-schedules in subSchSet of S, optGraphCons algorithm is used for checking if there exists optConf equivalent serial sub-schedules. Using this membership verification algorithm, we give the implementation of the STM system (scheduler) CP-ASC-Sched that implements CP-ASC.

Several data structures are maintained for each transaction by the scheduler CP-ASC-Sched. Each transaction maintains a graph, the vertices of which consists of all its children. The scheduler on receiving a request from the child of a transaction, say t_X, adds appropriate vertices and edges in t_X's graph. If the added edges do not cause t_X's graph to become cyclic, then the given operation is allowed to proceed. Otherwise, the operation is not allowed to proceed. To perform these tests, the algorithm maintains the following structures on t_X:

- A set $t_X.childNodes$ consists of all the child nodes (simple-memory operations and transactions) of t_X which have been initialized but not necessarily terminated.

- A set $t_X.chrnComplete$ that contains all the child nodes of t_X that have completed.
- A directed multigraph G_X, with each child node n_C of t_X having a vertex v_C in the graph.
- A $t_X.status$ field which can be in one of the states: *live, committed, aborted*. When a transaction is created, this field is initialized to live.
- For each child n_C of t_X, we define a set $t_X.extReads(n_C)$ which denotes the set of external-reads of n_C.
- A set $t_X.childExtReadList$ which is the union of external-reads of all children nodes in t_X. A read r_Y is added to this list if it is an external-read of some child say, n_C of t_X.
- A table structure $t_X.dataBuffers$ for storing all the writes performed by the child operations of this transaction. It consists a set of key and value pairs. The key corresponds to any data-item d written by the transaction and the value field is the value written to this key. For each data-item d, the value is accessed as $t_X.dataBuffers.v(d)$
- For each child n_C of t_X, we define a set $t_X.commitWrites(n_C)$ which denotes the set of commit-writes of n_C.
- A set $t_X.wrConfNodes(r_Y)$ is defined for each read r_Y in the set $t_X.childExtReadList$. This set contains child nodes of t_X. A child node n_C of t_X is in this set if a commit-write of n_C, say w_C, is in w-r conflict with r_Y.
- A set $t_X.rwConfNodes(r_Y)$ similar to wrNodes defined for each read r_Y in the set $t_X.childExtReadList$. This set contains child nodes of t_X. A child node n_C of t_X is in this set if a commit-write of n_C, say w_C, is in r-w conflict with r_Y.

We assume that a transaction can communicate with its parent and child transactions or with any other transaction if it knows other transaction's id. (The actual communication is performed by the processes that invoke the transactions.) The STM system maintains a queue for each transaction. The messages are stored in the queue and are executed in the order. We assume that the message delivery between two transactions is in FIFO order. The STM system provides a few functions which can be invoked by any transaction: STM-Invoke, STM-Write, STM-TryCommit, STM-Read, STM-Abort. We use two constructs in our implementation: block-until and sleep-until. The construct 'block-until' implies a transaction is blocked while waiting for a message and does not perform any other function until it has received the given message. If a transaction is waiting in the sleep-until state for an expected message, it can still process other messages while waiting on this message. All the messages are prefixed by 'm'. If a function is invoked to process a message then the function name is prefixed by 'proc'.

Main Idea: Here we give the main idea behind how STM-Write, STM-TryCommit and STM-Read are implemented. In our implementation, all these operations are executed in atomic manner if they execute successfully. In STM-Write operation, on receiving a write request $w_Y(d)$ from a transaction t_P, CP-ASC-Sched adds a node v_Y in t_P's graph. Then it adds completion edges from all the peers of w_Y that have completed before it. For any peer node n_Z of w_Y that has an external-read $r_X(d)$, a r-w conflict edge is added from v_Z to v_Y in G_P. Similarly for any peer node n_T that has a commit-write $w_T(d)$ a w-w conflict edge is added from v_T to v_Y. After adding these edges, if a

cycle is formed then the write is not allowed, edges are removed and the transaction t_P is aborted. Otherwise, the write operation is allowed to proceed.

In STM-TryCommit operation, the conflict edges are added similar to the STM-Write operation for all the commit-writes. After adding the edges if a cycle is formed in the parent transaction's graph then the corresponding child transaction is aborted. Otherwise it allowed to commit. The STM-Write and STM-TryCommit operations only involve operations on the data structures of a single transaction and hence can be implemented atomically if they are not aborted.

The STM-Read operation on the other hand involves multiple nodes. Thus in order to implement this functionality atomically, blocking of transactions involved is required. When a transaction t_P wishes to perform a read operation $r_X(d)$, if t_P does not contain d, it contacts its ancestors starting from its parent until the required data-item is found. On receiving the read request, an ancestor transaction t_Q adds w-r conflict edges in its graph G_Q if it contains the data-item d, and then checks for acyclicity. If the graph is acyclic then t_Q sends the value of d to t_P. If the graph turns out to be cyclic, then t_P is aborted; its vertex is deleted from its parent's graph and the reads of t_P stored in its ancestors as external-reads are deleted. On the other hand if t_Q does not contain the data-item d then it also stores the read r_Y in its external-read list, forwards the request to its parent and enters blocked state. It changes to unblocked state only when it receives an mUnblock message from its child after the read operation has terminated. The read r_Y is stored in the external-read list of transactions for identifying future r-w conflicts. On abort, the edges formed by the external-reads are deleted from the graph to satisfy the non-interference property. We use *block-until* construct in our pseudocode to represent the blocked state.

The abortCleanup function is invoked by t_X when it wishes to abort or when a read or write operation has failed. This function is invoked to perform 'cleanup' after a transaction t_X has been aborted. It is for performing these functionalities: (1)t_X's vertex is deleted from its parent's graph (2)If t_X was aborted when executing a STM-Read or STM-Write operation(and has live sub-transactions), then CP-ASC-Sched aborts all the live sub-transactions of t_X as well (the status of already committed transactions can not be altered). It sends messages to all the live sub-transactions to be aborted (which sets the transaction's status field as aborted). The sub-transactions in turn send abort messages to their sub-transactions along the tree. This is described in procSetDescAbort(). (3)Before the current operation, t_X's read operations could be stored in some of its ancestor transaction's external-read list (which is required as a part of CP-ASC-Sched). To correctly implement ASC, all these reads must be removed from the respective sets. To achieve this, CP-ASC-Sched sends messages to the ancestors of t_X (along the tree) which have stored reads of t_X in their external-read list. On receiving this message, an ancestor t_S removes the reads of t_X from its external-read list, removes all the edges caused by w-r and r-w conflicts with these reads from its graph G_S. The set of conflicts are kept track with the help of sets wrNodes and rwNodes. This is described in procRemExtReads().

In Section 7 we have described the pseudocode for the basic functions:STM-Write, procTryCommit, STM-Read and abortCleanup functions. Due to lack of space we have not described the following important functions: (i) $readAnsc(r_X(d), t_S, t_P)$ func-

tion which is invoked from $procReadReq(r_X(d), t_P)$. This function is very similar to $readNode(r_X(d), t_P)$ function. (ii) STM-Invoke, STM-Abort: STM-Invoke is a system function that creates and initializes the data structures for a new transaction. STM-Abort is called when the user wants to abort a given transaction. This function invokes abortCleanup function.

5 Reasoning about Safety and Liveness

Safety - All the schedules accepted by CP-ASC-Sched are a subset of CP-ASC (and ASC): In [9, 10], it was proved that for any given schedule there exists optConf equivalent serial schedule if and only if the corresponding conflict graph is acyclic. The conflict graph maintained is such that all the edges in the graph are only between vertices that are peers (in the computation tree). There are no edges between vertices of different levels. Hence, the conflict graph consists of many disconnected components.

Using the same idea, the conflict graph maintained by CP-ASC-Sched is distributed over all the transactions in the system. Each transaction maintains a disconnected component of the conflict graph. CP-ASC-Sched allows an operation belonging to a transaction t_X to execute only if it does not cause the its component of the conflict graph to become cyclic. This way, the entire conflict graph stays acyclic. Further, the graph consists of events from transactions that are either committed or live (events of aborted transactions are deleted). Hence, for any transaction t_A that gets aborted, $prefSubSch_A$ is optConf equivalent to a serial sub-schedule (since the graph is always maintained acyclic). Similarly $commitSubSch_R$ also has a optConf equivalent to a serial sub-schedule. Thus any schedule accepted by CP-ASC-Sched are in CP-ASC.

Liveness: To prove liveness, we have to show that all the STM functions eventually complete and no transaction ever stays in blocked state forever. One can see that a transaction t_X executing functions STM-Write, STM-TryCommit, STM-Abort can wait only on abortCleanup. Transaction t_X executing STM-Read can wait on abortCleanup or response for mReadReq message.

Transaction t_X executing abortCleanup can wait for a sub-transaction which currently is in the middle of a read operation. Similarly, t_X waiting on response for mReadReq message could be due to the the read operation of a sub-transaction. One can see that in this way the causal chain can be followed onto a single read operation, r_F waiting to complete as there is no circular wait (attributed to the tree nature of the computation). When the read operation r_F completes then the remaining waiting operations will accordingly terminate.

Another important aspect which can cause functions to not be able to execute is starvation: transaction t_X is not able to complete execution of a function as other higher priority transactions always take preference. But such a situation does not arise with CP-ASC-Sched as function and messages are kept to wait in FIFO queues. Hence, a transaction can not cause another transaction to starve.

6 Conclusion

Two important properties that concurrent executions of transactions in memory are expected to satisfy are: all-reads-consistency and non-interference. In this paper, we have considered the correctness criterion ASC [9, 10] that satisfies both these properties. We developed a scheduler CP-ASC-Sched for this class based on conflict preserving subclass of ASC. In our current scheduler implementation, the read operation, which can involve several transactions, is blocking. As a part of future work we plan to modify the lastWrite and optConf definitions such that the read operation does not block. Our future work also 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, 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.

[2] Joao Baretto, Aleksandar Dragojevic, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging Parallel Nesting in Transactional Memory. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Computing*, pages 91–100. ACM, 2010.

[3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards Formally Specifying and Verifying Transactional Memory. In *REFINE*, 2009.

[4] 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.

[5] 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.

[6] 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.

[7] Ranjeet Kumar and Krishnamurthy Vidyasankar. HParSTM: A Hierarchy-based STM Protocol for Supporting Nested Parallelism. In *TRANSACT 2011*, June 2011.

[8] 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.

[9] Sathya Peri and K.Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. Technical report, Memorial University of Newfoundland, July, 2011.

[10] Sathya Peri and K.Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *12th International Conference on Distributed Computing and Networking*, pages 95–106, 2011.

7 Pseudocode

Algorithm 1 abortCleanup(): Performs cleanup actions before a transaction t_X is aborted

```
1: procedure ABORTCLEANUP(t_X)
2:     invoke procSetDescAbort on t_X; // mSetDescAbort message is sent to itself
3:     send mRemExtReads(t_X) message to t_X's parent t_P;
4:     sleep-until mExtReadsDel message is received from t_P;
5: end procedure
```

Algorithm 2 procSetDescAbort(): Invoked when transaction t_X receives mSetDescAbort message from t_P

```
1: procedure PROCSETDESCABORT(t_X, t_P)
2:     if (t_X.status == live) then
3:         t_X.status = abort;
4:         User-InformAbort(t_X)
5:         // Send mSetDescAbort message to all the descendants since the current
       transaction is live
6:         for all (child transaction t_C of t_X) do
7:             send mSetDescAbort message to t_C;
8:         end for
9:         // Wait until all the descendants have aborted
10:        for all (child transaction t_C of t_X) do
11:            sleep-until a mAbortSetDone message is received from t_C;
12:        end for
13:        t_X.descAborted = true;
14:    else if (t_X.status == abort) then
15:        sleep-until t_X.descAborted becomes true;
16:    end if
       // Send mAbortSetDone message to parent
17:    if (t_X != t_P) then
18:        send mAbortSetDone message to t_P;
19:    end if
20: end procedure
```

Algorithm 3 procRemExtReads(): Invoked when a transaction t_S receives a mRemExtReads(t_X) message from its child t_R

```
1: procedure PROCREMEXTREADS(t_X, t_S, t_R)
2:     abortedReads = {the reads of t_X stored in t_S.childExtReadList )};
3:     // If t_S has no external-read from t_X's dSet then, there are no more nodes to
       cleanup
4:     if (abortedReads == nil) then
5:         send mExtReadsDel message to t_X;
6:         return;
7:     end if
```

Algorithm 4 procRemExtReads(): Continuation

```
8:    Remove the edges caused by the reads in abortedReads in t_S's graph;
9:    // Remove the reads in abortedReads from t_S
10:   for all (n_C in t_S.childNodes) do
11:       for all (read r_B in t_S.extReads(n_C) and also in abortedReads) do
12:           remove r_B from t_S.extReads(n_C);
13:       end for
14:   end for
15:   remove the reads in abortedReads from t_S.childExtReadList;
16:   // Remove t_X's vertex v_X from the graph G_P if t_R is same as t_X.
17:   if (t_R == t_X) then
18:       remove v_X from G_S and the corresponding edges if it is not already deleted;
19:       delete n_X from t_S.childNodes;
20:   end if
21:   // If the curNode is t_0 then there are no more nodes to verify
22:   if (t_S == t_0) then
23:       send mExtReadsDel message to t_X;
24:       return;
25:   end if
26: end procedure
```

Algorithm 5 readNode(): invoked on t_P from STM-Read

```
1:  function READNODE(r_X(d), t_P)
2:      Declarations found:boolean, ret:data-value;

3:      found = false;
4:      for all (child n_Y in t_P.childNodes such that a write w_Y(d) is in
        t_P.commitWrites(n_Y)) do
5:          add n_Y to t_P.wrConfNodes(r_X);
6:          add an edge from v_Y to v_X in G_P;
7:          found = true;
8:      end for
9:      // Check if t_P's dataBuffers contain data-item d;
10:     if (found == true) then
11:         if (isAcyclic(t_P) == false) then
12:             Remove the cycle, remove r_X from t_P's structures;
13:             return abort;
14:         else
15:             ret = t_P.dataBuffers.v(d);
16:             return ret;
17:         end if
18:     else
19:         return continue;
20:     end if
21: end function
```

Algorithm 6 STM-Read(): A transaction t_P performs a read operation $r_X(d)$. The return values are: d or `abort`

```
1:  procedure STM-READ(r_X(d), t_P)
2:      Declarations readValue:data-value;

3:      if (t_P.status == aborted) then
4:          return abort;
5:      end if
6:      // create node and vertex for this read operation
7:      add a node n_X to the list t_P.childNodes;
8:      add r_X to t_P.childExtReadList and t_P.extReads(t_C);
9:      create a vertex v_X for r_X in G_P;
10:     // add completion edges
11:     for all (n_C in t_P.chrnComplete ) do
12:         add an edge from v_C to v_X in G_P;
13:     end for
14:     readValue = t_P.readNode(r_X(d), t_P);
15:     if (readValue == continue) then
16:         send mReadReq(r_X(d),t_P) message to t_P's parent;
17:         block-until readValue message is obtained (from an ancestor);
18:         // Send mUnblock(readValue,m,e)ssage to its parent which unblocks and in
        turn send this message to its parent
19:         send mUnblock(r_X(d),t_P,readValue) message to t_P's parent;
20:     end if
21:     if (readValue == abort) then
22:         invoke abortCleanup(t_P);
23:         return abort;
24:     else
25:         add n_X to t_P.chrnComplete;
26:         return readValue;
27:     end if
28: end procedure
```

Algorithm 7 procReadReq(): This procedure is invoked when a transaction t_S receives a mReadReq($r_X(d)$,t_P) message from its child transaction t_R

```
1:  procedure PROCREADREQ(r_X(d), t_P)
2:      Declarations readValue:data-value;

3:      // readAnsc() function reads the value d on a given ancestor t_S on behalf of t_P
4:      readValue = readAnsc(r_X(d), t_S, t_P);
5:      if (readValue == continue) then
6:          send mReadReq(r_X(d),t_P) message to t_S's parent;
7:          block-until mUnblock(r_X(d),t_P,readValue) is received from child t_R
```

Algorithm 8 procReadReq(): Continuation

```
8:          // store the read as an external-read, if the return value is not abort
9:          if (readValue != abort) then
10:             add r_X(d) to t_P.childExtReadList;
11:             add r_X(d) to t_P.extReads(t_S);
12:         end if
13:         send mUnblock(r_X(d),t_P,readValue) message to t_S's parent;
14:     else
15:         send readValue message to t_P;
16:     end if
17: end procedure
```

Algorithm 9 STM-Write(): A transaction t_P performs a write operation $w_X(d)$. The return values are `ok` or `abort`

```
 1: procedure STM-WRITE(w_X(d), t_P)
 2:     if (t_P.status == aborted) then
 3:         return abort;
 4:     end if
 5:     add a node n_X to the list t_P.childNodes;
 6:     add w_X(d) to the set t_P.commitWrites(n_X);
 7:     create a vertex v_X for n_X in G_P;
 8:     // add completion edges
 9:     for all (n_C in t_P.chrnComplete ) do
10:         add an edge from v_C to v_X in G_P;
11:     end for
12:     // add rw conflict edges
13:     for all (n_Z in t_P.childNodes such that r_B(d) is in t_P.extReads(n_Z)) do
14:         // r_B(d) is in childExtReadList of t_P
15:         add n_Z to t_P.rwConfNodes(r_B(d));
16:         add an edge from v_C to v_X in G_P;
17:     end for
18:     // add ww conflict edges
19:     for all (n_Y in t_P.childNodes and w_Y(d) in t_P.commitWrites(n_Y)) do
20:         add an edge from v_Y to v_X in G_P;
21:     end for
22:     // check for acyclicity of the graph
23:     if (isAcyclic(t_P) == false) then
24:         invoke abortCleanup(t_P);
25:         return abort;
26:     else
27:         create a buffer for d in t_P.dataBuffers if it does not exist already;
28:         // the value is written by storing it in the data-buffers
29:         t_P.dataBuffers.v(d) = w_X(d)
30:         // make a note that the given write operation has completed
31:         add n_X to t_P.chrnComplete
32:         return ok;
33:     end if
34: end procedure
```

Algorithm 10 procTryCommit: Invoked when a transaction t_P receives a mTryCommit message from its child t_X. Returns commit or abort message to t_X

```
1: procedure PROCTRYCOMMIT(t_X, t_P)
2:     if (t_P.status == aborted) then
3:         send abort message to t_X;
4:         return;
5:     end if
6:     // add completion edges
7:     for all (n_C in t_P.chrnComplete ) do
8:         add an edge from v_C to v_X in G_P;
9:     end for
10:    // add rw conflict edges
11:    for all (commit-write w_X(d) in t_X.dataBuffers ) do
12:        for all (n_Z in t_P.childNodes such that r_B(d) is in t_P.extReads(n_Z)) do
13:            // n_Z is the node corresponding to t_Z
14:            add n_X to t_P.rwConfNodes(r_B(d));
15:            add an edge from v_Z to v_X in G_P;
16:        end for
17:        // add ww conflict edges
18:        for all (n_Y in t_P.childNodes such that w_Y(d) is in t_P.commitWrites(n_Y)) do
19:            add an edge from v_Y to v_X in G_P;
20:        end for
21:    end for
22:    if (isAcyclic(t_P) == false) then
23:        // Remove the cycle caused by t_X
24:        for all (child n_Z of t_P which has an external-read r_B such that n_X ∈
       t_P.rwConfNodes(r_B)) do
25:            remove n_X from t_P.rwConfNodes(r_B);
26:        end for
27:        // v_X is the vertex of t_X in G_P
28:        delete v_X and all its edges from G_P;
29:        send abort message to t_X;
30:        invoke abortCleanup(t_P);
31:    else
32:        for all (data-item d in t_X.dataBuffers) do
33:            if (t_P.dataBuffers does not contain a buffer for data-item d) then
34:                create a buffer for d in t_P.dataBuffers;
35:            end if
36:            // merge the commit-writes of t_X with t_P's buffers
37:            t_P.dataBuffers.v(d) = t_X.dataBuffers.v(d)
38:        end for
39:        add the contents of t_X.dataBuffers to t_P.commitWrites(n_X);
40:        add n_X to t_P.chrnComplete;
41:        send commit message to t_X;
42:    end if
43: end procedure
```