Multiversion Conflict Notion for Transactional Memory Systems *

Priyanka Kumar
priyanka@iith.ac.in
CSE Department
IIT Hyderabad
India

Sathya Peri
sathya_p@iith.ac.in
CSE Department
IIT Hyderabad
India

Abstract

In recent years, Software Transactional Memory systems (STMs) have garnered significant interest as an elegant alternative for addressing concurrency issues in memory. STM systems take optimistic approach. Multiple transactions are allowed to execute concurrently. On completion, each transaction is validated and if any inconsistency is observed it is aborted. Otherwise it is allowed to commit.

In databases a class of histories called as conflict-serializability (CSR) based on the notion of conflicts have been identified, whose membership can be efficiently verified. As a result, CSR is the commonly used correctness criterion in databases In fact all known single-version schedulers known for databases are a subset of CSR. Similarly, using the notion of conflicts, a correctness criterion, conflict-opacity (co-opacity) which is a sub-class of can be designed whose membership can be verified in polynomial time. Using the verification mechanism, an efficient STM implementation can be designed that is permissive w.r.t co-opacity. Further, many STM implementations have been developed that using the notion of conflicts.

By storing multiple versions for each transaction object, multi-version STMs provide more concurrency than single-version STMs. But the main drawback of co-opacity is that it does not admit histories that are uses multiple versions. This has motivated us to develop a new conflict notions for multi-version STMs. In this paper, we present a new conflict notion multi-version conflict. Using this conflict notion, we identify a new subclass of opacity, mvc-opacity that admits multi-versioned histories and whose membership can be verified in polynomial time. We show that co-opacity is a proper subset of this class.

An important requirement that arises while building a multi-version STM system is to decide "on the spot" or schedule online among the various versions available, which version should a transaction read from? Unfortunately this notion of online scheduling can sometimes lead to unnecessary aborts of transactions if not done carefully. To capture the notion of online scheduling which avoid unnecessary aborts in STMs, we have identified a new concept *ols-permissiveness* and is defined w.r.t a correctness-criterion, similar to permissiveness. We show that it is impossible for a STM system that is permissive w.r.t opacity to such avoid un-necessary aborts i.e. satisfy ols-permissiveness w.r.t opacity. We show this result is true for mvc-opacity as well.

1 Introduction

In recent years, Software Transactional Memory systems (STMs) [10, 23] have garnered significant interest as an elegant alternative for addressing concurrency issues in memory. STM systems take optimistic approach. Multiple transactions are allowed to execute concurrently. On completion, each

---

*A preliminary version of this work was presented at WTTM 2013 and published in [14]

transaction is validated and if any inconsistency is observed it is *aborted*. Otherwise it is allowed to *commit*.

An important requirement of STM systems is to precisely identify the criterion as to when a transaction should be aborted/committed. Commonly accepted correctness-criterion for STM systems is *opacity* proposed by Guerraoui, and Kapalka [7]. Opacity requires all the transactions including aborted to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. Unlike the correctness criterion for traditional databases serializability [19], opacity ensures that even aborted transactions read consistent values.

Another important requirement of STM system is to ensure that transactions do not abort unnecessarily. This referred to as the *progress* condition. It would be ideal to abort a transaction only when it does not violate correctness requirement (such as opacity). However it was observed in [2] that many STM systems developed so far spuriously abort transactions even when not required. A *permissive* STM [6] does not abort a transaction unless committing of it violates the correctness-criterion.

With the increase in concurrency, more transactions may conflict and abort, especially in presence many long-running transactions which can have a very bad impact on performance [3]. Perelman et al [21] observe that read-only transactions play a significant role in various types of applications. But long read-only transactions could be aborted multiple times in many of the current STM systems [11, 4]. In fact Perelman et al [21] show that many STM systems waste 80% their time in aborts due to read-only transactions.

It was observed that by storing multiple versions of each object, multi-version STMs can ensure that more read operations succeed, i.e., not return abort. History $H1$ shown in Figure 1 illustrates this idea. $H1 : r_1(x,0)w_2(x,10)w_2(y,10)c_2r_1(y,0)c_1$. In this history the read on y by T_1 returns 0 instead of the previous closest write of 10 by T_2. This is possible by having multiple versions for y. As a result, this history is opaque with the equivalent correct execution being T_1T_2. Had there not been multiple versions, $r_2(y)$ would have been forced to read the only available version which is 10. This value would make the read cause $r_2(y)$ to not be consistent (opaque) and hence abort.
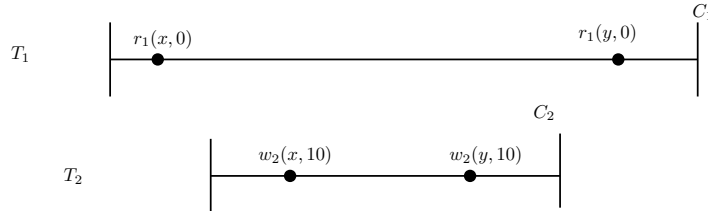

Figure 1: Pictorial representation of a History $H1$

Checking for membership of *multi-version view-serializability (MVSR)* [25, chap. 3], the correctness criterion for databases, has been proved to be NP-Complete [20]. We believe that the membership of opacity, similar to MVSR, can not be efficiently verified.

In databases a sub-class of MVSR, *conflict-serializability (CSR)* [25, chap. 3] has been identified, whose membership can be efficiently verified. As a result, CSR is the commonly used correctness criterion in databases since it can be efficiently verified. In fact all known single-version schedulers known for databases are a subset of CSR. Similarly, using the notion of conflicts, a sub-class of opacity, *conflict-opacity (co-opacity)* can be designed whose membership can be verified in polynomial time. Further, using the verification mechanism, an efficient STM implementation can be designed that is permissive w.r.t co-opacity [17]. Further, many STM implementations have been developed that using the idea of CSR[3, 24].

By storing multiple versions for each transaction object, multi-version STMs provide more concurrency than single-version STMs. But the main drawback of co-opacity is that it does not admit histories that are uses multiple versions. In other words, the set of histories exported by any STM implementation

that uses multiple versions is not a subset of co-opacity. Thus it can be seen that the co-opacity does not take advantage of the concurrency provided by using multiple versions.

This has motivated us to develop a new conflict notions for multi-version STMs. In this paper, we present a new conflict notion *mv-conflict*. Using this conflict notion, we identify a new subclass of opacity, *mvc-opacity* whose membership can be verified in polynomial time. We further show that co-opacityis a proper subset of this class. Further, the conflict notion developed is applicable on non-sequential histories as well unlike traditional conflicts.

In this paper, although we employed this conflict notion on opacity to develop the sub-class mvc-opacity, we believe that this conflict notion is generic enough to be applicable on other correctness-criterion such as local opacity [17], virtual worlds consistency [12] etc.

An important question that arises while building a multi-version STM system using the proposed mv-conflict notion: among the various versions available, which version should a transaction read from? The question was first analyzed in the context of database systems [9, 20]. A transactional system (either Database or STM) must decide "on the spot" or schedule online which version a transaction can read from based on the past history.

Unfortunately this notion of online scheduling can sometimes lead to unnecessary aborts of transactions. For instance, suppose a transaction T_i requests a read on transaction object x. Let the STM system has option of returning a value for x from among two versions, say v_1 and v_2. Suppose that the STM sytsem returns a version v_2. It is possible that this read can cause another T_j to abort in later to maintain correctness. But this abort of T_1 could have been avoided if the system returned v_1 instead. This concept is better illustrated in Section 4 where we show the difficulties with online scheduling.

To capture the notion of online scheduling which avoid unnecessary aborts in STMs, we have identified a new concept *ols-permissiveness*. It is defined w.r.t a correctness-criterion, similar to permissiveness. We show that it is impossible for a STM system that is permissive w.r.t opacity to such avoid unnecessary aborts i.e. satisfy ols-permissiveness w.r.t opacity. We show this result is true for mvc-opacity as well. We believe that this impossibility result will generalize to other correctness-criterion such as LO [17].

Roadmap. We describe our system model in Section 2. In Section 3 we formally define the conflict notion and describe how to verify its membership in polynomial time using graph characterization. In Section 4, we describe about the difficulty of online scheduling and associated impossibility results. In Section 5, we discuss about extending the mv-conflict notion to local-opacity and then give a brief outline of how to develop a STM system using mvc-opacity. Finally we conclude in Section 6.

2 System Model and Preliminaries

The notions and definitions described in this section follow the definitions of [17, 1]. We assume a system of n processes (or threads), $p_1, \ldots, p_n$ that access a collection of *objects* via atomic *transactions*. The STM systems is a software library that exports to the processes with the following *transactional operations* or *methods*: (i) *tbegin* operation, that starts a new transaction. It returns an unique transaction id; (ii) the *write*(x, v) operation that updates object x with value v, (iii) the *read*(x) operation that returns a value read in x; (iv) *tryC*() that tries to commit the transaction and returns *ok* or *abort*; (iv) *tryA*() that aborts the transaction and returns *abort*. The objects accessed by the read and write operations are called as transaction objects. For the sake of simplicity, we assume that the values written by all the transactions are unique. We also assume that the library ensures *deferred update semantics*, i.e. the write performed by a transaction T_k on a transaction object x will be visible to other transactions only after the commit of T_k.

The transactional operations could be non-atomic. To model this, we assume that all these operations have an *invocation* and *response* events. The operations of a transaction consists of the following events:

tbegin consists of $tbegin.inv()$ which is followed by a $tbegin.rsp(i)$ where i is the id of the transaction. The $read$ by transaction T_k is denoted as $read_k(x).inv()$ which is followed by $read_k(x).rsp(v)$ where v is either the current value of x or A. Similarly, the $write$ by transaction T_k is denoted as $write_k(x,v).inv()$ which is followed by $write_k(x,v).rsp(r)$ where r denotes the result of the write operation. It can either be ok or A. The $tryC$ by transaction T_k is denoted as $tryC_k.inv()$ which is followed by $tryC_k.rsp(r)$ where r is either ok or A. Similarly, $tryA$ by transaction T_k is denoted as $tryA_k.inv()$ which is followed by $tryA_k.rsp(A)$. When A is returned by an operation, it implies that the transaction T_k is aborted.

In the case where the operations are atomic, then we simplify the notation. tbegin is represented as $tbegin_k$, read as $read_k(x,v)/read_k(x,A)$, read as $write_k(x,v)/write_k(x,A)$, tryC as $tryC_k(ok)/tryC_k(A)$, tryA as $tryA_k(A)$.

When the *write*, *read* and *tryC()* return A, we say that the operation is *forcefully aborted*. Otherwise, we say that the operation has *successfully* executed. For simplicity we also refer to $tryC_k.rsp(ok)$ ($tryC_k(ok)$ in case of atomic operations) as c_k. Similarly, when a transactional operation returns A, i.e. $read_k(x).rsp(A), write_k(x,v).rsp(A), tryC_k.rsp(A), tryA_k.rsp(A)$ ($read_k(x,A), write_k(x,A)$, $tryC_k(A), tryA_k(A)$ respectively), we denote the event as a_k. Along the same lines, we refer to (non-atomic) read and write operations as $r_k(x,v), w_k(x,v)$ when the invocation and response events are not relevant to the context. Sometimes, we also drop the transaction object x and the value v read/written depending on the context.

For a transaction T_k, we denote all the events (operations in case of sequential histories) of T_k as $evts(T_k)$. All the transaction objects read by T_k are denoted as $rset(T_k)$ and all the transaction objects written by it are denoted as $wset(T_k)$.

Histories. A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $evts(H)$. We denote $<_H$ a total order on the transactional events in H. We identify a history H as tuple $\langle evts(H), <_H \rangle$. Figure 2 shows history $H2 : w_1(x,5).inv() \ w_2(x,10).inv() \ w_1(x,5).rsp(ok) \ w_2(x,10).rsp(ok) \ r_3(x).inv() \ tryC_1.rsp(ok)$ $tryC_2.rsp(ok) \ r_3(x).rsp(5)$. In Figure 2, for simplicity we have not shown inv and rsp events separately.

We say a history is *sequential* if invocation of each transactional operation is immediately followed by a matching response. For simplicity, we treat each transactional operation as atomic in sequential histories. The order $<_H$ is a total order on the transactional operations in H for sequential histories. History $H1$ shown in Figure 1 is a sequential history. We also refer to histories which are not sequential as *non-sequential*.
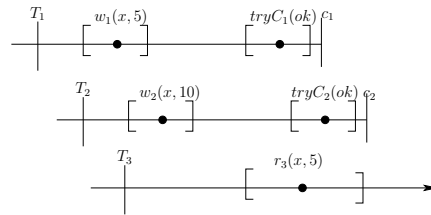

Figure 2: Pictorial representation of a History $H2$

Let $H|T_k$ denote the sub-history consisting of events of T_k in H. We only consider *well-formed* histories here, i.e., (1) each $H|T_k$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly *completed* with a $tryC$ or $tryA$ operation. In the read-only prefix, each transaction consists of read on a transaction object x only once. This restriction brings no loss of generality [18]; (2) a thread invoking transactional operations never invokes another operation before receiving a response from the previous one; it does not invoke any operation op_k after receiving a c_k or a_k response.

We denote the set of transactions that appear in H is denoted by $txns(H)$. A transaction $T_k \in txns(H)$ is *complete* in H if $H|T_k$ ends with a response event. In other words, all the operations in T_k end with a response event. We assume that all the operations in sequential histories are complete. A transaction $T_k \in txns(H)$ is *t-complete* if $H|T_k$ ends with a_k or c_k ; otherwise, T_k is t-incomplete. The history H is t-complete if all transactions in $txns(H)$ are t-complete. The set of committed (resp., aborted) transactions in H is denoted by $committed(H)$ (resp., $aborted(H)$). The set of *incomplete* or *live* transactions in H is denoted by *live(H)* ($live(H) = txns(H) - committed(H) - aborted(H)$). In $H2$, T_3 is live while T_1, T_2 are committed.

We assume that every history has an initial committed transaction T_0 that initializes all the transaction objects with 0. We say that two histories, H and H' are *equivalent*, denoted as $H \approx H'$ if $evts(H) = evts(H')$ i.e. all the events in H and H' are the same. Note that H could be non-sequential whereas H' could be sequential.

Transaction orders. For two transactions $T_k, T_m \in txns(H)$, we say that T_k *precedes* T_m in the *real-time order* of H, denote $T_k \prec_H^{RT} T_m$, if T_k is t-complete in H and the last event of T_k precedes the first event of T_m in H. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then T_k and T_m *overlap* in H. Consider two histories h, H' that are equivalent to each other, i.e. $evts(H) = evts(H')$. We say a history H *respects* the real-time order of another history H' if all the real-time orders of H' are also in H, i.e. $\prec_{H'}^{RT} \subseteq \prec_H^{RT}$.

A history H is *t-sequential* if there are no overlapping transactions in H, i.e., every two transactions are related by the real-time order.

Correctness Criterion. We denote a collection of histories as *correctness-criterion*. Typically, all the histories of a correctness-criterion satisfy some property. Serializability [19] is the well-accepted correctness-criterion in databases. Several correctness-criteria have been proposed for STMs such as Opacity [7], Virtual World Consistency [12], Local Opacity [17], TMS [5] etc.

Implementations. A STM *implementation* provides the processes with functions for implementing read, write, tryC (and possibly tryA) functions. We denote the set of histories *generated* by a STM implementation I as $gen(I)$. We say that an implementation I is correct w.r.t to a correctness-criterion C if all the histories generated by I are in C i.e. $gen(I) \subseteq C$.

Progress Conditions. Let C be a correctness-criterion with H in it. Let T_a be an aborted transaction in H. We say that a history H is permissive w.r.t C if committing T_a, by replacing the abort value returned by an operation in T_a with some non-abort value, would cause H to violate C. In other words, if T_a is committed then H will no longer be in C. We denote the set of histories permissive w.r.t C as $perm(C)$. We say that STM implementation I is permissive [6] w.r.t some correctness-criterion C (such as opacity) if every history H generated by I is permissive w.r.t C, i.e., $gen(I) \subseteq perm(C)$.

3 New Conflict Notion for Multi-Version Systems

In this section, we define a new conflict notion for multi-version STM systems. First, we describe about the *Opacity* [7], a popular correctness-criterion. Then we describe the new conflict notion, *multi-version conflict order*.

3.1 Opacity

We define a few notations on histories for describing opacity.

Valid, Legal and Multi-versioned histories. Let H be a non-sequential history and $r_k(x, v)$ be a successful read operation (i.e $v \neq A$) in H. Then $r_k(x, v)$, is said to be *valid* if there is a transaction T_j in H such that T_j is committed in H, $w_j(x, v)$ is in $evts(T_j)$ and the response of r_k does not occur before invocation of $tryC_j$ in H. Formally, $\langle r_k(x, v)$ is valid $\Rightarrow \exists T_j : (r_k(x).rsp(v) \not\prec_H tryC_j.inv()) \wedge (w_j(x, v) \in evts(T_j)) \wedge (v \neq A) \rangle$. We say that the commit operation $tryC_j.rsp(ok)$ (or c_j) is r_k's *valWrite* and

formally denote it as $H.valWrite(r_k)$. The history H is valid if all its successful read operations are valid. The notion of validity formalizes deferred update semantics described in Section 2.

In $H2$, $tryC_1.rsp(ok) = c_1 = H2.valWrite(r_3(x,5))$, $r_k(x).rsp(5) \not<_{H2} tryC_1.inv()$ and $(w_1(x,5) \in evts(T_1))$. Hence, $r_3(x,5)$ is valid and as a result, $H2$ is valid as well.

For a sequential history H, the definition of validity of $r_k(x,v)$ boils down as follows: a successful read $r_k(x,v)$, is said to be *valid* if there is a transaction T_j in H that commits before r_k and writes v to x. Formally, $\langle r_k(x,v)$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x,v)) \wedge (w_j(x,v) \in evts(T_j)) \wedge (v \neq A)\rangle$.

Consider a sequential history H. We define $r_k(x,v)$'s *lastWrite* as the latest commit event c_k such that c_k precedes $r_k(x,v)$ in H and $x \in Wset(T_k)$ (T_k can also be T_0). Formally, we denote it as $H.lastWrite(r_k)$. A successful read operation $r_k(x,v)$ (i.e $v \neq A$), is said to be *legal* if transaction T_k (which contains r_k's lastWrite) also writes v onto x. Formally, $\langle r_k(x,v)$ is legal $\Rightarrow (v \neq A) \wedge (H.lastWrite(r_k(x,v)) = c_k) \wedge (w_k(x,v) \in evts(T_k))\rangle$. The sequential history H is legal if all its successful read operations are legal. Thus from the definition, we get that if H is legal then it is also valid.

It can be seen that in $H1$, $c_0 = H1.valWrite(r_1(x,0)) = H1.lastWrite(r_1(x,0))$. Hence, $r_1(x,0)$ is legal. But $c_0 = H1.valWrite(r_1(y,0)) \neq c_1 = H1.lastWrite(r_1(y,0))$. Thus, $r_1(y,0)$ is valid but not legal.

We denote a sequential history H as *non-single-versioned* if it is valid but **not** legal. If a history H is non-single-versioned, then there is at least one read, say $r_k(x)$ in H that is valid but not legal. The history $H1$ is non-single-versioned. This definition can not be generalized to non-sequential histories as legality is not defined for non-sequential histories.

Opacity. To define the correctness-criterion opacity, we first define completion of a history that is incomplete. For a history H, we construct the completion of H, *opq-completion* denoted $\overline{H^o}$, as follows (similar to [1]):

1. for every complete transaction T_k in H that is not t-complete, insert the event sequence: $tryA_k.inv()$ $tryA_k.rsp(A)$ after the last event of transaction T_k;

2. for every incomplete operation op_k of T_k in H, if $op_k = read_k \vee write_k \vee tryA_k$, then insert the response event A somewhere after the invocation of op_k;

3. for every incomplete $tryC_k$ operation where T_k is in H, insert response event ok or A somewhere after the invocation of $tryC_k$.

In case of a sequential history H, the completion $\overline{H^o}$ is constructed by inserting an $tryA_k(A)$ (or a_k) after the last operation of transaction T_k, for every transaction T_k in H that is t-incomplete.

A history H is said to be *opaque* [7, 8] if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H^o}$ and (2) S respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$.

By requiring S being equivalent to $\overline{H^o}$, opacity treats all the incomplete transactions as aborted. The validity requirement on H ensures that write operations of aborted transactions are ignored. This definition of opacity is closer in spirit to *du-opacity* [1]. It can be seen that both the histories $H1$ and $H2$ are opaque. The opaque equivalent t-sequential history for $H1$ being T_1T_2 and the equivalent t-sequential histories of $H2$ are $T_1T_3T_2$, $T_2T_1T_3$.

3.2 Motivation for a New Conflict Notion

It is not clear if checking whether a history is opaque or can be performed in polynomial time. Checking for membership of *multi-version view-serializability (MVSR)* [25, chap. 3], the correctness criterion for databases, has been proved to be NP-Complete [20]. We believe that the membership of opacity, similar to MVSR, can not be efficiently verified.

In databases a sub-class of MVSR, *conflict-serializability (CSR)* [25, chap. 3] has been identified, whose membership can be efficiently verified. As a result, CSR is the commonly used correctness-criterion in databases since it can be efficiently verified. In fact all known single-version schedulers known for databases are a subset of CSR. Similarly, using the notion of conflicts, a sub-class of opacity, *conflict-opacity (co-opacity)* can be designed whose membership can be verified in polynomial time. Further, using the verification mechanism, an efficient STM implementation can be designed that is permissive w.r.t co-opacity [16, 17].

As already discussed in Section 1, by storing multiple versions for each transaction object, multi-version STMs provide more concurrency than single-version STMs. But the main drawback of co-opacity is that it does not admit histories that are non-single-versioned. Thus co-opacity does not take advantage of the concurrency provided by using multiple versions. Another big drawback being that co-opacity does not admit histories that are non-sequential. In other words, the set of histories exported by many STM implementation are not a subset of co-opacity. Hence, proving correctness of these STM systems is difficult. In the rest of this sub-section, we formally define co-opacity and show the drawbacks. Some of the definitions and proofs in this section are coming directly from [16, 17].

We define co-opacity using *conflict order* [25, Chap. 3]. Consider a sequential history H. For two transactions T_k and T_m in $txns(H)$, we say that T_k *precedes* T_m *in conflict order*, denoted $T_k \prec_H^{CO} T_m$, (a) (c-c order): $c_k <_H c_m$ and $wset(T_k) \cap wset(T_m) \neq \emptyset$; (b) (c-r order): $c_k <_H r_m(x,v)$, $x \in wset(T_k)$ and $v \neq A$; (c) (r-w order) $r_k(x,v) <_H c_m$, $x \in wset(T_m)$ and $v \neq A$.

Thus, it can be seen that the conflict order is defined only on operations that have successfully executed. Further, it can also be seen that this order is defined only for histories that are sequential.

Using conflict order, co-opacity is defined as follows: A sequential history H is said to be *conflict opaque* or *co-opaque* if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H^o}$ and (2) S respects $\prec_H^{RT}$ and $\prec_H^{CO}$.

From the definitions of conflict order and co-opacity it is clear that these notions are only specific to sequential histories. Thus, history $H2$ is not co-opaque. It must be noted that $H2$ can be generated by a STM system that maintains only a single version of each transaction object. The asynchronous nature of thread execution can result in $H2$ by the STM system.

Having seen a drawback, we will next show that if any sequential history is non-single-versioned, then it can not be in co-opacity.

Lemma 1 *Consider two sequential histories $H1$ and $H2$ such that $H1$ is equivalent to $H2$. Suppose $H1$ respects conflict order of $H2$, i.e., $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. Then, $\prec_{H1}^{CO} = \prec_{H2}^{CO}$.*

Proof. Here, we have that $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. In order to prove $\prec_{H1}^{CO} = \prec_{H2}^{CO}$, we have to show that $\prec_{H2}^{CO} \subseteq \prec_{H1}^{CO}$. We prove this using contradiction. Consider two events p, q belonging to transaction $T1, T2$ respectively in $H2$ such that $(p,q) \in \prec_{H2}^{CO}$ but $(p,q) \notin \prec_{H1}^{CO}$. Since the events of $H2$ and $H1$ are same, these events are also in $H1$. This implies that the events p, q are also related by CO in $H1$. Thus, we have that either $(p,q) \in \prec_{H1}^{CO}$ or $(q,p) \in \prec_{H1}^{CO}$. But from our assumption, we get that the former is not possible. Hence, we get that $(q,p) \in \prec_{H1}^{CO} \Rightarrow (q,p) \in \prec_{H2}^{CO}$. But we already have that $(p,q) \in \prec_{H2}^{CO}$. This is a contradiction. $\square$

Lemma 2 *Let $H1$ and $H2$ be two sequential histories which are equivalent to each other and their conflict order are the same, i.e. $\prec_{H1}^{CO} = \prec_{H2}^{CO}$. Then $H1$ is legal iff $H2$ is legal.*

Proof. It is enough to prove the 'if' case, and the 'only if' case will follow from symmetry of the argument. Suppose that $H1$ is legal. By contradiction, assume that $H2$ is not legal, i.e., there is a read operation $r_j(x,v)$ (of transaction T_j) in $H2$ with its lastWrite as c_k (of transaction T_k) and T_k writes $u \neq v$ to x, i.e. $w_k(x,u) \in evts(T_k)$. Let $r_j(x,v)$'s lastWrite in $H1$ be c_i of T_i. Since $H1$ is legal, T_i writes v to x, i.e. $w_i(x,v) \in evts(T_i)$.

Since $evts(H1) = evts(H2)$, we get that c_i is also in $H2$, and c_k is also in $H1$. As $\prec_{H1}^{CO}=\prec_{H2}^{CO}$, we get $c_i <_{H2} r_j(x,v)$ and $c_k <_{H1} r_j(x,v)$.

Since c_i is the lastWrite of $r_j(x,v)$ in $H1$ we derive that $c_k <_{H1} c_i$ and, thus, $c_k <_{H2} c_i <_{H2} r_j(x,v)$. But this contradicts the assumption that c_k is the lastWrite of $r_j(x,v)$ in $H2$. Hence, $H2$ is legal. $\qquad\square$

Lemma 3 *If a sequential history H is non-single-versioned then H is not in co-opacity. Formally,* $\langle (H\,is\ sequential) \wedge (H\,is\ non\text{-}single\text{-}versioned) \implies (H \notin co\text{-}opacity)\rangle$.

Proof. We prove this using contradiction. Assume that H is non-single-versioned, i.e. H is valid but not legal. But suppose that H is in co-opacity. Since H is sequential, conflict order can be applied on it. From the definition of co-opacity, we get that there exists a t-sequential and legal history S such that $\prec_{H}^{CO}\subseteq\prec_{S}^{CO}$. From Lemma 1, we get that $\prec_{H}^{CO}=\prec_{S}^{CO}$. Combining this with Lemma 2 and the assumption that H is not legal, we get that S is not legal. But this contradicts out assumption that S legal. Hence, H is not in co-opacity. $\qquad\square$

3.3 Multi-Version Conflict Definition

Having seen the shortcomings of co-opacity, we will see how to overcome them. The main reason for the shortcoming is because conflict notion has been defined only among the events of sequential histories. We address this issue here by defining a new conflict notion for non-sequential histories.

To define this notion on any history, we have developed a another definition of completion of any history H, *mvc-completion* denoted as $\overline{H^m}$. It is same as $\overline{H^o}$ except for step 3 which is modified as follows: for every incomplete $tryC_k$ operation where T_k is in H, insert response event A somewhere after the invocation of $tryC_k$. Thus in $\overline{H^m}$, all incomplete tryC operations are treated as aborted.

Definition 1 *For a history H, we define multi-version conflict order(mvc order), denoted as $\prec_{H}^{mvc}$, between operations of $\overline{H^m}$ as follows: (a) commit-commit (c-c) order: $c_i \prec_{H}^{mvc} c_j$ if $tryC_i.rsp(ok) <_H tryC_j.rsp(ok)$ for two committed transaction T_i, T_j and both of them write to x; (b) commit-read (c-r) order: Let $r_i(x,v)$ be a read operation in H with its valWrite as c_k (belonging to the committed transaction T_k). Then for any committed transaction T_j that writes to x, either the response of the T_j's commit occurs before T_k or T_k is same as T_j, formally $(tryC_j.rsp(ok) <_H tryC_k.rsp(ok)) \vee (T_j = T_k)$, we define $c_j \prec_{H}^{mvc} r_i$. (c) read-commit (r-c) order: Let $r_i(x,v)$ be a read operation in H with its valWrite as c_k. Then for any committed transaction T_j that writes to x, if the T_j's commit response event occurs after T_k's commit response event, i.e. $(tryC_k.rsp(ok) <_H tryC_j.rsp(ok))$, we define $r_i \prec_{H}^{mvc} c_j$.*

Observe that the mvc order is defined on the operations (and not events) of $\overline{H^m}$ and not H. The set of conflicts in $H2$ are: [c-r : $(c_0, r_3), (c_1, r_3)$], [r-c : (r_3, c_2)], [c-c : $(c_0, c_1), (c_0, c_2), (c_1, c_2)$]. Here, it can be observed that $tryC_2.rsp(ok)$ occurs before $r_3(x).rsp(5)$. Yet, r_3 occurs before c_2 in the mvc order.

It is not difficult to extend the mvc order to sequential histories: replace the response of a tryC event with the corresponding tryC operation and the response of a read event with the corresponding read operation. The set of conflicts in $H1$ are: [c-r : $(c_0, r_1(x,0)), (c_0, r_1(y))$], [r-c : $(r_1(x), c_2), (r_1(y), c_2)$], [c-c : (c_0, c_2)].

We say that a history H' *satisfies* the mvc order of a history H, $\prec_{H}^{mvc}$, denoted as $H' \vdash \prec_{H}^{mvc}$ if: (1) H' is equivalent to $\overline{H^m}$; (2) Consider two operations op_i, op_j in H. Let e_i, e_j be the corresponding response events of these operations. Then, $op_i \prec_{H}^{mvc} op_j$ implies $e_i <_{H'} e_j$. If H, H' are sequential, then op and e would be the same.

Note that for any sequential history H that is non-single-versioned, H does not satisfy its own mvc order $\prec_H^{mvc}$. For instance the non-single-versioned order in history $H1$ consists of the pair: $(r_1(y, 0), c_2)$. But c_2 occurs before $r_1(y, 0)$ in $H1$. We formally prove this property using the following lemmas.

Lemma 4 *Consider a valid history H. Let H' be a sequential history (which could be same as H). If H' satisfies $\prec_H^{mvc}$ then H' is legal. Formally, $\langle (H \text{ is valid}) \wedge (H' \text{ is sequential}) \wedge (H' \vdash \prec_H^{mvc}) \implies (H' \text{ is legal})\rangle$.*

Proof. Assume that H' is not legal. Hence there exists a read operation, say $r_i(x, v)$, in $evts(H')$ that is not legal. This implies that lastWrite of r_i is not the same as its valWrite. Let $c_l = H'.lastWrite(r_i) \neq H'.valWrite(r_i) = c_v$. Let $w_l(x, u) \in evts(T_l)$ and $w_v(x, v) \in evts(T_v)$ where $\{T_l, T_v, T_i\} \in txns(H')$. As H is valid, we have that $tryC_v.rsp(ok) <_H r_i(x).rsp(v)$. Since $H' \vdash \prec_H^{mvc}$, we have that $evts(H) = evts(H')$. Thus $\{T_l, T_v, T_i\}$ are also in $txns(H)$.

There are two cases w.r.t ordering of events in H:

- $tryC_l.rsp(ok) <_H tryC_v.rsp(ok)$: From the definition of mvc order, we get that $tryC_l.rsp(ok) \prec_H^{mvc} tryC_v.rsp(ok)$. Since H' satisfies $\prec_H^{mvc}$ and is sequential, we get that $c_l <_{H'} c_v <_{H'} r_i$.

- $tryC_v.rsp(ok) <_H tryC_l.rsp(ok)$: Again, from the definition of mvc order, we get that $r_i(x).rsp(v) \prec_H^{mvc} tryC_l.rsp(ok)$. Since H' satisfies $\prec_H^{mvc}$ and is sequential, we get that $c_v <_{H'} r_i <_{H'} c_l$.

In both cases, it can be seen that c_l is not the previous closest commit operation to r_i in H'. Hence, we have a contradiction which implies H' is legal. $\square$

Using this lemma, we get the following corollary,

Corollary 5 *Consider a valid history H. Let H' be a non-single-versioned history equivalent to H (which could be same as H). Then, H' does not satisfy $\prec_H^{mvc}$. Formally, $\langle (H' \text{ is non-single-versioned}) \wedge (H \text{ is valid}) \wedge (H' \approx H) \implies (H' \nvdash \prec_H^{mvc})\rangle$.*

Proof. We are given that H is valid, H and H' are equivalent to each other. Since H' is non-single-versioned, we get that H' is sequential but not legal. Combining all these with the contrapositive of Lemma 4, we get that $H' \nvdash \prec_H^{mvc}$. $\square$

Now, we show that if a history is legal, then it satisfies it own mv-conflict order.

Lemma 6 *Consider a legal history H. Then, H satisfies its own mv-conflict order $\prec_H^{mvc}$. Formally, $\langle (H \text{ is legal}) \implies (H \vdash \prec_H^{mvc})\rangle$.*

Proof. We are given that H is legal. From the definition of legality, we get that S is sequential. We will prove this lemma using contradiction. Suppose, H does not satisfy its own mv-conflict order i.e. $(H \nvdash \prec_H^{mvc})$. Consider two operations, say p_i (belonging to transaction T_i) and q_j (belonging to transaction T_j) in $evts(H)$. From our assumption of contradiction, we get that $(p_i \prec_H^{mvc} q_j)$ but $(p_i \nprec_H q_j)$. This implies that $(q_j <_H p_i)$ since all the operations are totally ordered in H (which is sequential). Let us consider the various cases of mv-conflict between p_i and q_j:

- $p_i = c_i, q_j = c_j$ (c-c order): From mv-conflict definition, we get that $c_i \prec_H^{mvc} c_j$ implies that $c_i <_H c_j$.

- $p_i = c_i, q_j = r_j$ (c-r order): Let the valWrite of r_j in H be c_v belonging to transaction T_v. From mv-conflict definition, we get that either $c_i <_H c_v <_H r_j$ or $c_i = c_v <_H r_j$. In either case, we have that $c_i <_H r_j$.

- $p_i = r_i, q_j = c_j$ (r-c order): Similar to the above case, Let the valWrite of r_i in H be c_v belonging to transaction T_v. From mv-conflict definition, we have two option: (i) $c_v <_H c_i <_H r_j$ or (ii) $c_v <_H r_j <_H c_i$. Since H is legal, option (i) is not possible (unless $c_v = c_i$). This leaves us with option (ii), $r_j <_H c_i$.

Thus in all the three cases, we get that $(p_i <_H q_j)$ which implies that H satisfies $\prec_H^{mvc}$. $\qquad\square$

We now prove an interesting property about satisfaction relation.

Lemma 7 *Consider a valid history H and a sequential history S. If, S satisfies H's mv-conflict order $\prec_H^{mvc}$ then S also respects H's mv-conflict order. Formally, $\langle (H$ is valid$) \wedge (S$ is sequential$) \wedge (S \vdash \prec_H^{mvc}) \implies (\prec_H^{mvc} \subseteq \prec_S^{mvc}) \rangle$.*

Proof. We are given that H is valid, S is sequential and satisfies H's mv-conflict order $\prec_H^{mvc}$. Thus, from Lemma 4 we get that S is legal. From Lemma 6, we get that S satisfies its own mv-conflict order $\prec_H^{mvc}$, i.e. $S \vdash \prec_S^{mvc}$.

Now, we prove this lemma using contradiction. Suppose, S satisfies $\prec_H^{mvc}$ but S does not respect mv-conflict order of H, i.e. $\prec_H^{mvc} \nsubseteq \prec_S^{mvc}$. This implies that there exists two operations, p_i, q_j in H and S such that p_i precedes q_j in H's mvc order but not in S's mvc order. We have that,

$$(p_i \prec_H^{mvc} q_j) \wedge (p_i \nprec_S^{mvc} q_j) \xrightarrow[\text{satisfy def'n}]{S \vdash \prec_S^{mvc}} (p_i \prec_H^{mvc} q_j) \wedge (p_i \nprec_S q_j) \xrightarrow{S \vdash \prec_H^{mvc}} (p_i <_S q_j) \wedge (p_i \nprec_S q_j).$$

This implies a contradiction. Hence, we have that S respects mv-conflict order of H. $\qquad\square$

3.4 Multi-Version Conflict Opacity

We now illustrate the usefulness of the conflict notion by defining another subset of opacity *mvc-opacity* which is a superset of co-opacity. We formally define it as follows (along the same lines as co-opacity):

Definition 2 *A history H is said to be* multi-version conflict opaque *or* mvc-opaque *if H is valid and there exists a t-sequential history S such that (1) S is equivalent to $\overline{H^m}$, i.e. $S \approx \overline{H^m}$; (2) S respects $\prec_H^{RT}$, i.e. $\prec_H^{RT} \subseteq \prec_S^{RT}$ and S satisfies $\prec_H^{mvc}$, i.e. $S \vdash \prec_H^{RT}$.*

It can be seen that both the histories $H1$ and $H2$ are mvc-opaque. The mvc equivalent t-sequential history for $H1$ being $T_1 T_2$ and the equivalent t-sequential history for $H2$ being $T_1 T_3 T_2$.

Consider a history H that is mvc-opaque and let S be the mvc equivalent t-sequential history. Then from Lemma 7, we get that S satisfies H's mv-conflict order, i.e. $\prec_H^{mvc} \subseteq \prec_S^{mvc}$. Please note that we don't restrict S to be legal in the definition. But it turns out that if H is mvc-opaque then S is automatically legal as shown in Lemma 4. Now, we have the following theorem.

Theorem 8 *If a history H is mvc-opaque, then it is also opaque. Formally, $\langle (H \in mvc\text{-}opacity) \implies (H \in opacity) \rangle$.*

Proof. Since H is mvc-opaque, it follows that H is valid and there exists a t-sequential history S such that (1) S is equivalent to $\overline{H^m}$ and (2) S respects $\prec_H^{RT}$ and S satisfies $\prec_H^{mvc}$. Since, S is equivalent to $\overline{H^m}$, it can be seen that S is equivalent to $\overline{H^o}$ as well. This, in order to prove that H is opaque, it is sufficient to show that S is legal. As S satisfies $\prec_H^{mvc}$, from Lemma 4 we get that S is legal. Hence, H is opaque as well. $\qquad\square$

Thus, this lemma shows that mvc-opacity is a subset of opacity. Actually, mvc-opacity is a strict subset of opacity. Consider the history $H3 = r_1(x, 0) r_2(z, 0) r_3(z, 0) w_1(x, 5) c_1 r_2(x, 5) w_2(x, 10) w_2(y, 15) c_2 r_3(x, 5) w_3(y, 25) c_3$. . Figure 3 shows the representation of this history. The set of mv-conflicts in

$H3$ are (ignoring the conflicts with c_0): [c-r : $(c_1, r_2(x, 5)), (c_1, r_3(x, 5))$], [r-c : $(r_3(x, 5), c_2)$], [c-c : $(c_1, c_2), (c_2, c_3)$]. It can be verified that $H3$ is opaque with the equivalent t-sequential history being $T_1 T_3 T_2$. But there is no mvc equivalent t-sequential history. This is because of the conflicts: $(r_3(x, 5), c_2), (c_2, c_3)$. Hence, $H3$ is not mvc-opaque.
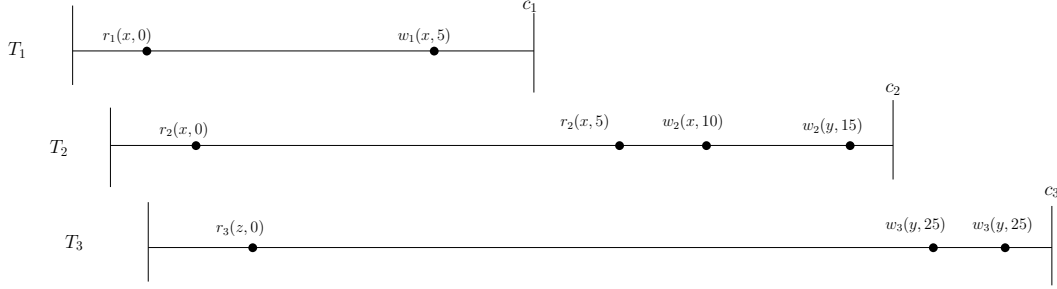

Figure 3: Pictorial representation of $H3$

Next, we will relate the classes co-opacity and mvc-opacity. In the following theorem, we show that co-opacity is a subset of mvc-opacity.

Theorem 9 *If a history H is co-opaque, then it is also mvc-opaque. Formally, $\langle (H \in co\text{-}opacity) \implies (H \in mvc\text{-}opacity) \rangle$.*

Proof. Since H is co-opaque, we get that there exists an equivalent legal t-sequential history S that respects the real-time and conflict orders of H. Thus if we show that S satisfies mvc order of H then H is mvc-opaque. From the definition of co-opacity, we have that H is sequential.

Since S is legal, it turns out that the conflicts and mv-conflicts are the same. To show this, let us analyse each conflict order:

- c-c order: If two operations are in c-c conflict, then by definition they are also ordered by the c-c mvc order.

- c-r order: Consider the two operations, say c_k and r_i that are in conflict (due to a transaction object x). Hence, we have that $c_k <_H r_i$. Let $c_v = H.valWrite(r_i)$. Since, S is legal, either $c_k = c_v$ or $c_k <_H c_j$. In either case, we get that $c_k \prec_H^{mvc} r_i$.

- r-c order: Consider the two operations, say c_k and r_i that are in conflict (due to a transaction object x). Hence, we have that $r_i <_H c_k$. Let $c_v = H.valWrite(r_i)$. Since, S is legal, $c_v <_H r_i <_H c_k$. Thus in this case also we get that $r_i \prec_H^{mvc} c_k$.

Thus in all the three cases, conflicts among the operations in S also result in mv-conflicts among these operations. Hence, S satisfies the mvc order of H. $\qquad \square$

This theorem shows that co-opacity is a subset of mvc-opacity. The history $H1$ is mvc-opaque but not in co-opacity. Hence, co-opacity is a strict subset of mvc-opacity. Figure 4 shows the relation between the various classes.

3.5 Graph Characterization of MVC-Opacity

In this section, we will describe graph characterization of mvc-opacity. This characterization will enable us to verify its membership in polynomial time.

Given a history H, we construct a *multi-version conflict graph*, $MVCG(H) = (V, E)$ as follows: (1) $V = txns(H)$, the set of transactions in H; (2) an edge (T_i, T_j) is added to E whenever
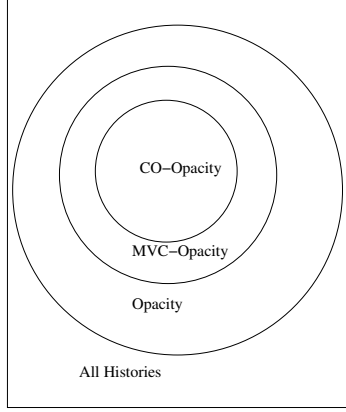
11

Figure 4: Relation between the various classes

2.1 real-time edges: If T_i precedes T_j in H;

2.2 mvc order edges: If T_i contains an operation p_i and T_j contains p_j such that $p_i \prec_H^{mvc} p_j$.

The multi-version conflict graph gives us a polynomial time graph characterization for mvc-opacity. We show it using the following lemma and theorem.

Lemma 10 *Consider a legal and t-sequential history S. Then, $MVCG(S)$ is acyclic. Formally,* $\langle (S \text{ is legal}) \wedge (S \text{ is t-sequential}) \implies (MVCG(S) \text{ is acyclic})\rangle$.

Proof. Since S is t-sequential, we can order all the transactions by their real-time order. We assume w.l.o.g that all the transactions of S are ordered as $T_1 <_S T_2 <_S .... <_S T_n$. Thus, with our assumption we get that $T_i <_S T_j$ implies that $i < j$.

Now we will show that for any edge (T_i, T_j) in $MVCG(S)$, we get that $i < j$. The edge (T_i, T_j) can be one of the following:

- real-time: It follows from this case that T_j started only after the commit of T_i. Hence, we get that $T_i <_S T_j$ and this implies $i < j$.

- c-c conflict: Here, we have that $c_i <_S c_j$. Since S is t-sequential, we get that all the events of T_i occur before all the events of T_j. Hence $T_i <_S T_j$ and thus $i < j$.

- c-r conflict: Here, $c_i <_S r_j$ for a read $r_j(x, v)$. Since S is t-sequential, similar to the above case we get that $T_i <_S T_j$ and hence $i < j$.

- r-c conflict: Here, $r_i <_S c_j$ for a read $r_i(x, v)$. Let valWrite of r_i be c_l. From the definition of mv-conflict, we have two cases. Either (i) $c_l <_S c_j <_S r_i$ or (ii) $c_l <_S r_i <_S c_j$. Since S is legal, we get that case (i) is not possible. Otherwise, c_j would have been the valWrite of r_i. This leaves only case (ii) which implies that $r_i <_S c_j$. Since S is t-sequential, similar to the above two cases we get that $T_i <_S T_j$ and hence $i < j$.

Thus in all the cases, we get that an edge (T_i, T_j) in the $MVCG(S)$ implies that $i < j$. Hence, a cycle is not possible in such a graph. $\qquad\square$

Theorem 11 *A valid history H is mvc-opaque iff $MVCG(H)$ is acyclic.*

Proof. We prove both the directions.

if $MVCG(H)$ is acyclic then H is mvc-opaque: Since $MVCG(H)$ is acyclic, we can perform a topological sort on $MVCG(H)$. Using the order obtained from the topological sort, we order all the transactions of $\overline{H^m}$ to construct a t-sequential history S. Thus from the construction of S, we get that S is equivalent to $\overline{H^m}$.

It can be seen that S respects $\prec_H^{RT}$. If T_i occurs before T_j in H, then there is an edge between T_i between T_j in $MVCG(H)$. This edge ensures that T_i occurs before T_j in S as well.

Consider two operations of H, p_i (belonging to T_i) and q_j (belonging to T_j). If $p_i \prec_H^{mvc} q_j$ then there is an edge between T_i and T_j in $MVCG(H)$. This edge ensures that $T_i <_S T_j$. Thus, we get that $p_i <_S q_j$. This shows that S satisfies $\prec_H^{mvc}$.

if H is mvc-opaque then $MVCG(H)$ is acyclic: Since H is mvc-opaque, we get that there exists a t-sequential, legal history S that is equivalent to H. We also have that S respects the real-time order of H and satisfies mvc order of H. Combining this with Lemma 7, we get that S respects the mv-conflict order of H. Formally, $(\prec_H^{RT} \subseteq \prec_S^{RT}) \wedge (\prec_H^{mvc} \subseteq \prec_S^{mvc})$.

Thus, from the graph construction of $MVCG(H)$, $MVCG(S)$, we get that $MVCG(H) \subseteq MVCG(S)$. Since S is legal and t-sequential, from Lemma 10 we get that $MVCG(S)$ is acyclic. This implies that $MVCG(H)$ is also acyclic since it is a subgraph of $MVCG(S)$. $\square$

Figure 5 shows the multi-version conflict graphs for the histories $H1$, $H2$ and $H3$. In these graphs and other conflict graphs shown in this paper, we have ignored T_0 for simplicity.

Multi-Version Conflict Graph of $H1$

Multi-Version Conflict Graph of $H2$
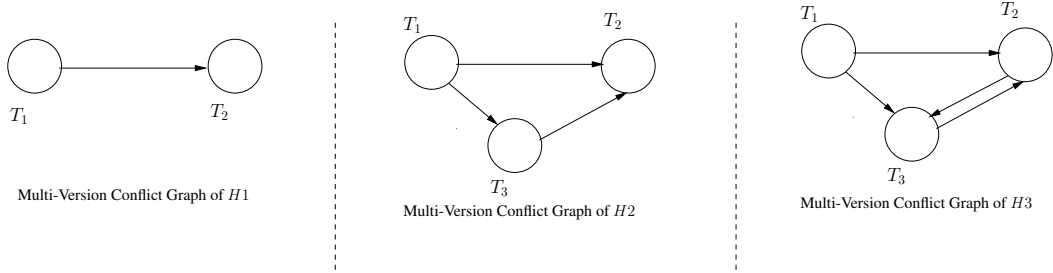
Multi-Version Conflict Graph of $H3$

Figure 5: multi-version conflict graphs of $H1$, $H2$ and $H3$

4 Online Scheduling with Multiple Versions

An important question that arises while building a multi-version STM system is among the various versions available, which version should a transaction read from? The question was first analyzed in the context of database systems [9, 20]. A transactional system (either Database or STM) must decide "on the spot" or schedule online which version a transaction can read from based on the past history.

We say a STM implementation I *schedules online* (i.e. decides on the spot) if every invocation to an operation that it exports (read, write, tryC, tryA) returns in finite time. We denote I as *online schedulable* (OLS) (term inspired from databases). Note that I can make a decision on scheduling based only on the past history of operations seen so far as it does not have any idea of the future. In other words, all the methods of I are wait-free.

But unfortunately this notion of online scheduling can sometimes lead to unnecessary aborts of transactions. We illustrate this idea with an example while considering mvc-opacity as the correctness-criterion. Consider the sequential history $H4 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)c_2r_3(x,?_2^1)$. In this history, $r_3(x)$ has the option of reading 1 from T_1 or 2 from T_2 (denoted as $r_3(x,?_2^1)$). T_3 can not read x from T_0 as it would violate the real-time order requirement between T_0, T_1 imposed by mvc-opacity(as well as opacity). Suppose T_3 reads 2 for x written by T_2. Now consider a sequence of

events that follow the read operation. Let $H5 = w_1(x, 1)w_1(y, v_1)w_2(x, 2)r_k(z, 0)c_1 w_2(z, v_2)r_j(b, 0)$ $c_2 r_3(x, 2)w_3(b, v_3)w_k(b, v_k)w_j(d, v_j)$. $H5$ is a possible extension of $H4$. It can be seen that $H4$ is mvc-opaque (with T_3 reading 2). But $H5$ is not as there is a cycle between the transactions T_2, T_3, T_k in the multi-version conflict graph.

Suppose T_3 had read 1 instead of 2 for x. Now consider the modified history consisting of same extension of $H4$ (assuming that the read of T_3 did not affect the future events), $H6 = w_1(x, 1)w_1(y, v_1)w_2(x, 2)r_k(z, 0)c_1 w$ It can be seen that $H6$ is mvc-opaque. $H5$ will be mvc-opaque if T_k is aborted. This shows that the versions read by a transaction can cause other transactions to abort in future. Figure 6 illustrates this concept.
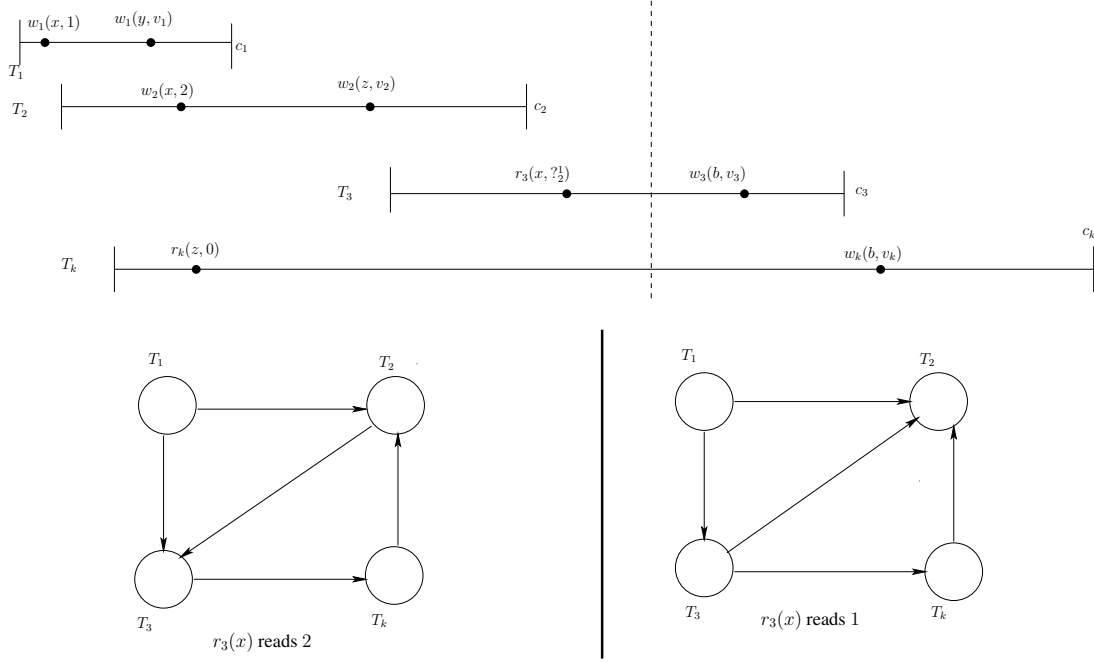

Figure 6: Illustration of difficulties with online scheduling

To capture the notion of online scheduling which avoid unnecessary aborts in STMs, we have identified a new concept *ols-permissiveness* and is defined w.r.t a correctness-criterion, similar to permissiveness.

Let C be a correctness-criterion with a history H being permissive w.r.t C, i.e. $H \in perm(C)$. Then let T_a be an aborted transaction in H. Let $r_i(x, v)$ be any successful read operation(i.e. $v \neq A$) in H that completed before the abort response of T_a, i.e. $(r_i(x).rsp(v) <_H r_a(z).rsp(A)/$ $tryC_a.rsp(A)/tryA_a.rsp(A))$ (for some r_a). Suppose $r_i(x)$ read a different value u $(A \neq u \neq v)$ from among the various versions available (that were created before by update transactions). Then, committing T_a, by replacing the abort value returned by an operation in T_a with some non-abort value, would cause H to violate C. In other words, if T_a were to be committed with $r_i(x)$ reading u, H will no longer be in C. We say that H is *ols-permissive* w.r.t C.

In the above example, $H5$ is not ols-permissive w.r.t mvc-opacity. We denote the set of histories that are ols-permissive w.r.t C as $ols\text{-}perm(C)$. Along the same lines, we say that STM implementation I is ols-permissive w.r.t some correctness-criterion C (such as opacity) if every history H generated by I is ols-permissive w.r.t C, i.e., $gen(I) \subseteq ols\text{-}perm(C)$.

It turns out that multiple versions make online scheduling very difficult. In fact we show in the following sub-section that it is impossible to achieve ols-permissiveness.

4.1 On Impossibility of ols-permissiveness with multiple versions

As mentioned above, multiple versions make online scheduling very difficult. In this sub-section, we first show that it is impossible for an OLS implementation I that to be ols-permissive w.r.t mvc-opacity. Then, we show that it is impossible for I to be ols-permissive w.r.t opacity as well.

To show our result, we consider a centralized adversary $\mathcal{A}$ that has complete knowledge of the working of the implementation I. We assume that the adversary invokes the next method on the implementation I based on the previous responses. It waits for the response of the previous event before it can fire the next invocation event. Hence, the histories considered in following sub-section are sequential. It must be noted that making this assumption does not restrict the generality of the results as sequential histories are a special case of histories.

Theorem 12 *No OLS STM implementation can be ols-permissive w.r.t mvc-opacity.*

Proof. Let us suppose that an OLS STM implementation I is ols-permissive w.r.t mvc-opacity. From the definition of ols-permissiveness, we get that I is also permissive w.r.t mvc-opacity.

Some of the arguments used in this proof are similar to the description in the start of this section. Consider the sequential history $H7 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)r_j(b,0)c_2r_3(x,?_2^1)$ (this history is similar to $H4$). Assume that the adversary $\mathcal{A}$ invokes same operations on I as this history. Since I is permissive w.r.t mvc-opacity, it will not unnecessarily return abort to any of these operations. For the read $r_k(z)$, I will return 0 since so far no write to z has taken place. The same argument holds for $r_j(b,0)$. Thus the output by I is same as $H7$ until $r_3(x)$.

For $r_3(x)$, I has the option of returning either 1 or 2. It can not return 0 (written by T_0) as it violate real-time ordering required by mvc-opacity. Suppose I returned 2 for the read $r_3(x)$. Now consider an extension of $H7$, $H8 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)r_j(b,0)c_2r_3(x,2)w_3(b,v_3)w_k(b,v_k)w_j(d,v_j)$. It can be seen that $H8$ is not mvc-opaque as there is a cycle between the transactions T_2, T_3, T_k in the multi-version conflict graph. Suppose $\mathcal{A}$ invokes the operations of $H8$ on I after the invocation of $r_3(x)$. Since $H8$ is not mvc-opaque, $\mathcal{A}$ invokes the next operation only after receiving the previous response and I is permissive w.r.t mvc-opaque, I would be forced to abort T_k.

Now, consider the case that I had returned 1 for $r_3(x)$ instead of 2. The resulting history, $H9 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)r_j(b,0)c_2r_3(x,1)w_3(b,v_3)w_k(b,v_k)w_j(d,v_j)c_jc_k$. It can be seen that $H9$ is mvc-opaque with an equivalent t-sequential history being $T_1T_jT_3T_kT_2$. Thus, in this case I would not have to abort any transaction. $H9$ is in $ols\text{-}perm(mvc\text{-}opacity)$. Figure 7 illustrates this scenario.

Next consider another extension of the history $H7$, $H10 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)r_j(b,0)c_2r_3(x,1)w_3(b,v_3)w_k(d,v_k)w_j(z,v_j)c_jc_k$. It can be seen that, $H10$ is not mvc-opaque as there is a cycle between the transactions T_2, T_j, T_3 in the multi-version conflict graph. Suppose $\mathcal{A}$ invokes the operations of $H10$ on I. Let I returns 1 for $r_3(x)$ (not knowing what operations could be invoked in future). Then in this case, I would be forced to abort T_j since $H10$ is not mvc-opaque, $\mathcal{A}$ invokes the next operation only after receiving the previous response and I is permissive w.r.t mvc-opaque.

On the other hand, suppose I returned 2 for the above sequence of operation invocation by $\mathcal{A}$. The resulting history is $H11 = w_1(x,1)w_1(y,v_1)w_2(x,2)r_k(z,0)c_1w_2(z,v_2)r_j(b,0)c_2r_3(x,2)w_3(b,v_3)w_k(d,v_k)w_j(z,v_j)$. It can be seen that this history is mvc-opaque with an equivalent t-sequential history being $T_1T_kT_2T_jT_3$. Hence, in this case I would output this history without aborting any transaction. $H11$ is in $ols\text{-}perm(mvc\text{-}opacity)$. Figure 8 illustrates this scenario.

These examples illustrate that given the sequence of operations in $H7$, returning either 1 or 2 for $r_3(x)$ by I can possibly cause some transaction in future to abort depending on the sequence of invocations. Whereas reading the other value would have avoided the abort. This is because when I
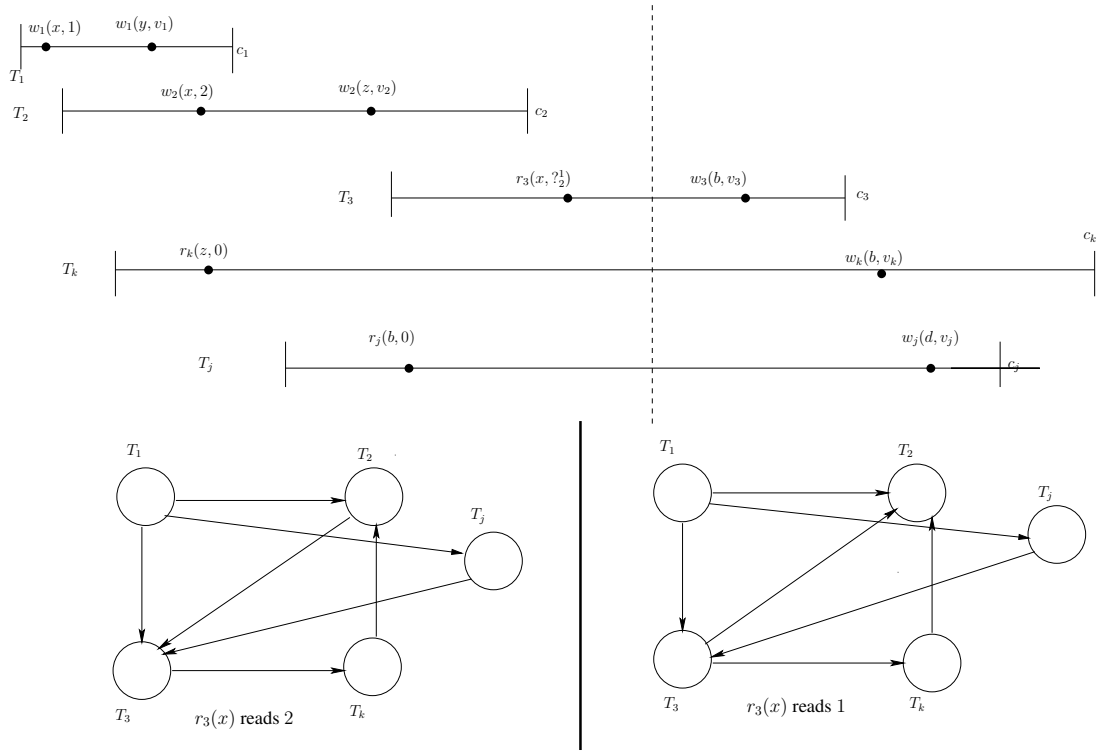
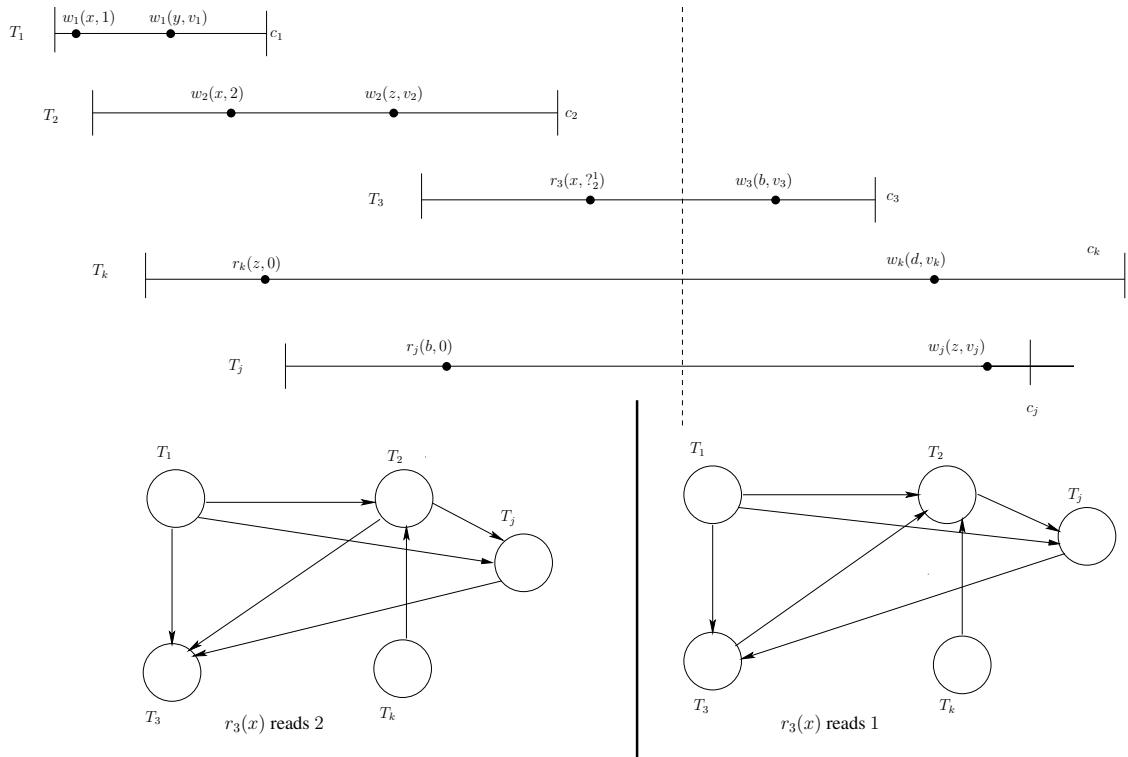Figure 7: $H9$ containing $r_3(x, 1)$ is mvc-opaque

Figure 8: $H11$ containing $r_3(x, 2)$ is mvc-opaque

16

received the event $r_3(x)$, it has no idea about the future events and is OLS. Hence, I can not be in $ols\text{-}perm(mvc\text{-}opacity)$.

The difficulty of online scheduling is not restricted only to mvc-opacity. We now show that the impossibility extends to opacity as well. In showing this, we use arguments very similar what we have used to the above proof.

Theorem 13 *No OLS STM implementation can be ols-permissive w.r.t opacity.*

5 Discussion

5.1 Multi-Version Conflicts on other Correctness Criteria

So far in this paper, we have demonstrated the effectiveness of mvc orders using opacity. This conflict notion can be applied to other correctness-criterion such as local-opacity (LO) [17] and virtual world consistency (VWC) [13]. Both these correctness-criteria were defined for sequential histories.

A history H is locally-opaque if the following conditions hold: (1) Let the sub-history H_{com} consist of events from all the committed transactions in H. Then H_{com} should be opaque; (2) Let T_a be an aborted transaction in H. Suppose H_a be a sub-history consisting of all the transactions that committed before the abort of T_a in H. Then, for each aborted transaction T_a, H_a is opaque.

We say a history H is *multi-version conflict local-opaque* (MVLO) if for each history H, (1) H_{com} is mvc-opaque; (2) for each aborted transactions T_a, H_a is mvc-opaque.

Further, it can be seen that the impossibility results of Section 4, can be extended to MVLO and LO as well.

We believe that along the same lines, the multi-version conflict definition can be extended to VWC.

5.2 Outline of a STM System using Multiversion Conflicts

Having developed a conflict definition that accommodates multiple versions, we describe the outline of a STM system. The main idea behind the algorithm is based on the notion serialization graph testing [22, 17] that was developed for databases [25]. According to this idea, the STM system maintains a graph based on the operations that have been executed so far. A new operation is allowed to execute only if it does not form a cycle in the graph.

But a few important questions arise about the implementation which is typical of any multi-version system: (a) how many version should the STM system store? (b) which version should a transaction read from?

The issue of online scheduling was analyzed in Section 4 which partly addresses the question of which version should a transaction read from. Since whichever version a transaction reads from can possibly cause another transaction to abort, in our implementation we have decided to read the closest available version that does not violate mvc-opaque. Using these ideas, we are currently developing a new algorithm.

To address the question on number of versions maintained, it was shown in [15] that by not maintaining a limit on the number of versions, greater concurrency can be achieved. So, we do not keep any limit on the number of versions maintained in the STM system developed. But with this approach the number of version keep growing over time making the system inefficient. So, a garbage collection strategy that removes the unwanted versions is to be designed. We are currently working on it.

6 Conclusion

In this paper, we have presented a new conflict notion *multi-version conflict*. Using this conflict notion, we developed a new subclass of opacity, mvc-opacity that admits multi-versioned histories and whose

17

membership can be verified in polynomial time. We showed that co-opacity, a sub-class of opacity that is based on traditional conflicts, is a proper subset of this class. Further, the proposed conflict notion mv-conflict can be applied on non-sequential histories as well unlike traditional conflicts.

To demonstrate the effectiveness of the new conflict notion, we employed opacity, a popular correctness-criterion. As discussed, we believe that this conflict notion can be easily extended to other correctness-criterion such as LO and VWC.

An important requirement that arises while building a multi-version STM system using the propose conflict notion is to decide "on the spot" or schedule online among the various versions available, which version should a transaction read from? Unfortunately this notion of online scheduling can sometimes lead to unnecessary aborts of transactions if not done carefully. To capture the notion of online scheduling which avoid unnecessary aborts in STMs, we have identified a new concept *ols-permissiveness*. We show that it is impossible for a STM system that is permissive to avoid such un-necessary aborts i.e. satisfy ols-permissiveness w.r.t opacity. We show this result is true for mvc-opacity as well.

Actually, multi-version conflict notions have been proposed for multi-version databases as well [9]. But in their model of histories, the authors do not specify which version a transaction reads. So it is not clear how their model will be applicable to STM histories. Moreover, their notion of conflicts were applicable only for sequential histories.

As a part of the ongoing work, we plan to develop an efficient STM system using the mv-conflicts and measure the cost of the implementation.

References

[1] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 601–610, July 2013.

[2] Hagit Attiya and Eshcar Hillel. A Single-Version STM that is Multi-Versioned Permissive. *Theory Comput. Syst.*, 51(4):425–446, 2012.

[3] Utku Aydonat and Tarek Abdelrahman. Serializability of Transactions in Software Transactional Memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, feb 2008.

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, sep 2006. Springer-Verlag Lecture Notes in Computer Science volume 4167.

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

[6] 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.

[7] 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.

[8] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[9] Thanasis Hadzilacos and Christos H. Papadimitriou. Algorithmic aspects of multiversion concurrency control. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 96–104, New York, NY, USA, 1985. ACM.

[10] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, Jul 2003.

[12] 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.

[13] Damien Imbs and Michel Raynal. A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In *Proceedings of the 16th international conference on Structural Information and Communication Complexity*, SIROCCO'09, pages 266–280, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Priyanka Kumar and Sathya Peri. Multi-version conflict notion. *CoRR*, abs/1307.8256, 2013.

[15] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version stm algorithm. In *ICDCN*, pages 212–226, 2014.

[16] Petr Kuznetsov and Sathya Peri. On non-interference of transactions. *CoRR*, abs/1211.6315, 2012.

[17] Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. In *ICDCN*, pages 197–211, 2014.

[18] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.

[19] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[20] Christos H. Papadimitriou and Paris C. Kanellakis. On Concurrency Control by Multiple Versions. *ACM Trans. Database Syst.*, 9(1):89–99, March 1984.

[21] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective Multi-Versioning STM. In *DISC*, pages 125–140, 2011.

[22] 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.

[23] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[24] Arnab Sinha and Sharad Malik. Runtime checking of serializability in software transactional memory. In *IPDPS*, pages 1–12, 2010.

[25] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.