Multiversion Conflict Notion

Priyanka Kumar
priyanka@iitp.ac.in
Indian Institute of Technology Patna, India

Sathya Peri
sathya@iitp.ac.in
Indian Institute of Technology Patna, India

Abstract

This work introduces the notion of multi-version conflict notion. Using this conflict notion, we define a subclass of opacity, mvc-opacity whose membership can be verified in polynomial time.

1. Introduction

In recent years, Software Transactional Memory systems (STMs) [8, 16] 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*.

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 [5]. 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 [13], 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 [1] that many STM systems developed so far spuriously abort transactions even when not required. A *permissive* STM [4] 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 [2]. Perelman et al [15] 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 [3, 9]. In fact Perelman et al [15] 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 $H1$ 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.

Checking for membership of *multi-version view-serializability (MVSR)* [18, chap. 3], the correctness criterion for databases, has been proved to be NP-Complete [14]. We believe that the membership of opacity, similar to MVSR, can not be efficiently verified.

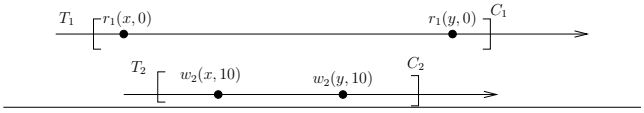

Figure 1. Pictorial representation of a History $H1$

In databases a sub-class of MVSR, *conflict-serializability (CSR)* [18, 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 [11]. Further, many STM implementations have been developed that using the idea of CSR[2, 17].

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. As a result, it is not clear if a multi-version STM implementation can be developed that is permissive w.r.t some sub-class of opacity.

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* whose membership can be verified in polynomial time. We further show that co-opacityis a proper subset of this class. Even though we used this conflic notion on opacity to develop sub-classes, we believe that this conflict notion is generic enough to be applicable on other correctness-criterion such as virtual worlds consistency [10] etc.

2. System Model and Preliminaries

The notions and definitions described in this section follow the definitions of [11]. We assume a system of n processes, $p_1, \ldots, p_n$ that access a collection of *objects* via atomic *transactions*. The processes are provided with four *transactional operations*: the *write*(x, v) operation that updates object x with value v, the *read*(x) operation that returns a value read in x, *tryC*$()$ that tries to commit the transaction and returns *commit* (c for short) or *abort* (a for short), and *tryA*$()$ that aborts the transaction and returns A. 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.

Operations *write*, *read* and *tryC*() may return a, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operation has *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations returns a or c. Abort and commit operations are called *terminal operations*.

For a transaction T_k, we denote all its read operations as $Rset(T_k)$ and write operations $Wset(T_k)$. Collectively, we denote all the operations of a transaction T_i as $evts(T_k)$.

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 $evts(H)$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let $<_H$ denote the total order on the transactional operations incurred by H. With this assumption the only relevant events of a transaction T_k are of the types: $r_k(x, v)$, $r_k(x, A)$, $w_k(x, v)$, $w_k(x, v, A)$, $tryC_k(C)$ (or c_k for short), $tryC_k(A)$, $tryA_k(A)$ (or a_k for short). We identify a history H as tuple $\langle evts(H), <_H\rangle$.

Let $H|T$ denote the history consisting of events of T in H, and $H|p_i$ denote the history consisting of events of p_i in H. We only consider *well-formed* histories here, i.e., (1) each $H|T$ 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[a], and (2) each $H|p_i$ consists of a sequence of transactions, where no new transaction begins before the last transaction completes (commits or a aborts).

We assume that every history has an initial committed transaction T_0 that initializes all the data-objects with 0. The set of transactions that appear in H is denoted by $txns(H)$. 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 *incomplete*(H) (*incomplete*$(H) = txns(H) - committed(H) - aborted(H)$). For a history H, we construct the *completion* of H, denoted $\overline{H}$, by inserting a_k immediately after the last event of every transaction $T_k \in$ *incomplete*(H).

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 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. 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.

Valid, legal and multi-versioned histories. Let H be a 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 that commits before r_K and $w_j(x, v)$ is in $evts(T_j)$. Formally, $\langle r_k(x, v)$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in evts(T_j)) \wedge (v \neq A)\rangle$. We say that the commit operation 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.

We define $r_k(x, v)$'s *lastWrite* as the latest commit event c_i such that c_i precedes $r_k(x, v)$ in H and $x \in$ *Wset*(T_i) (T_i 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_i (which contains r_k's lastWrite) also writes v onto x. Formally, $\langle r_k(x, v)$ is legal $\Rightarrow (v \neq A) \wedge (H.lastWrite(r_k(x, v)) = c_i) \wedge (w_i(x, v) \in evts(T_i))\rangle$. The history H is legal if all its successful read operations are legal. Thus from the definitions 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 define a history H as *non-single-versioned* if it is valid but **not** legal. If a hisory H is non-single-versioned, then there is at least one read, say $r_i(x)$ in H that is valid but not legal. The history $H1$ is non-single-versioned. Along the same lines, we say that a STM implementation is non-single-versioned if it exports atleast one history that is non-single-versioned.

Opacity. We say that two histories H and H' are *equivalent* if they have the same set of events. Now a history H is said to be *opaque* [5, 6] if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subseteq \prec_S^{RT}$. By requiring S being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted.

3. New Conflict Notion for Multi-Version Systems

3.1 Motivation for a New Conflict Notion

As already discussed in Section 1, co-opacity does not admit histories that are non-single-versioned. In this section, we will formally prove this which is the motivation for developing a new conflict notion for STMs.

Some of the definitions and proofs in this section are coming directly from [11]. We define co-opacity using *conflict order* [18, Chap. 3] as: for two transactions T_k and T_m in $txns(H)$, we say that T_k *precedes* T_m *in conflict order*, denoted $T_k \prec_H^{CO} T_m$, if (c-c order) $c_k <_H c_m$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$, (c-r order) $c_k <_H r_m(x, v)$, $x \in Wset(T_k)$ and $v \neq A$, or (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. Using conflict order, co-opacity is defined as follows:

DEFINITION 1. *A 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 $\overline{H}$ and (2) S respects $\prec_H^{RT}$ and $\prec_H^{CO}$.*

Readers familiar with the databases literature may notice co-opacity is analogous to the *order commit conflict serializability* (OCSR) [18].

We can now prove that if a history is non-single-versioned, then it is not in co-opacity. Due to lack of space, we are only outlining the lemma and theorem statements in this write-up.

LEMMA 1. *If a history H is non-single-versioned then H is not in co-opacity. Formally, $\langle (H$ is non-single-versioned$) \implies (H \notin$ co-opacity$)\rangle$.*

Having shown the shortcoming of conflicts, we now define a new conflict notion in the next sub-section that will accommodate non-single-versioned histories as well.

3.2 Multi-Version Conflict Definition

We define a few notations to describe the conflict notion. Consider a history H. For a read $r_i(x, v)$ in H, we define r_i's *valWrite*, formally $H.valWrite(r_i)$, as the commit operation c_j belonging to the transaction T_j that occurs before r_x in H and writes v to x. If there are multiple such committed transactions that write v to x then the valWrite is the commit operation closest to r_x.

DEFINITION 2. *For a history H, we define multi-version conflict order, denoted as $\prec_H^{mvc}$, between operations of $\overline{H}$ as follows: (a)*

[a] This restriction brings no loss of generality [12].

commit-commit (c-c) order: $c_i \prec_H^{mvc} c_j$ *if* $c_i <_H c_j$ *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_j *(belonging to the committed transaction* T_j*). Then for any committed transaction* T_k *that writes to* x *and either commits before* T_j *or is same as* T_j, *formally* $(c_k <_H c_j) \vee (c_k = c_j)$ *, we define* $c_k \prec_H^{mvc} r_i$. *(c) read-commit (r-c) order: Let* $r_i(x, v)$ *be a read operation in* H *with its lastWrite as* c_j *(belonging to the committed transaction* T_j*). Then for any committed transaction* T_k *that writes to* x *and commits after* T_j, *i.e.* $c_j <_H c_k$, *we define* $r_i \prec_H^{mvc} c_k$.

Observe that the multi-version conflict order is defined on the operations of $\overline{H}$ and not H. The set of conflicts in $H1$ are: [c-r : $(c_0, r_1(x, 0)), (c_0, r_1(y, 0))$], [r-c : $(r_1(x, 0), c_2), (r_1(y, 0), c_1)$], [c-c : (c_0, c_2)].

We say that a history H' *satisfies* the multi-version conflict order of a history H, $\prec_H^{mvc}$ if: (1) the events of H' are same as $\overline{H}$, i.e. H' is equivalent to $\overline{H}$. (2) For any two operations op_i and op_j, $op_i \prec_H^{mvc} op_j$ implies $op_i <_{H'} op_j$. We denote this by $H' \vdash \prec_H^{mvc}$. Otherwise, we say that H' does not satisfy $\prec_H^{mvc}$ and denote it as $H' \nvdash \prec_H^{mvc}$.

Note that for any history H that is non-single-versioned, H does not satisfy its own multi-version conflict 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 2. *Consider a (possibly non-single-versioned) valid history* H. *Let* H' *be a history which satisfies* $\prec_H^{mvc}$. *Then* H' *is valid and* $\prec_{H'}^{mvc} = \prec_H^{mvc}$. *Formally,* $\langle (H \text{ is valid}) \wedge (H' \vdash \prec_H^{mvc}) \implies (H' \text{ is valid}) \wedge (\prec_{H'}^{mvc} = \prec_H^{mvc}) \rangle$.

LEMMA 3. *Consider a (possibly non-single-versioned) valid history* H. *Let* H' *be a valid history (which could be same as* H*). If* H' *satisfies* $\prec_H^{mvc}$ *then* H' *is legal. Formally,* $\langle (H' \text{ is valid}) \wedge (H' \vdash \prec_H^{mvc}) \implies (H' \text{ is legal}) \rangle$.

Using this lemma, we get the following corollary,

COROLLARY 4. *Consider a (possibly non-single-versioned) valid history* H. *Let* H' *be a non-single-versioned history (which could be same as* H*). Then,* H' *does not satisfy* $\prec_H^{mvc}$. *Formally,* $\langle (H' \text{ is non-single-versioned}) \implies (H' \nvdash \prec_H^{mvc}) \rangle$.

4. Multi-Version Conflict Opacity

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

DEFINITION 3. *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* $\overline{H}$ *and (2)* S *respects* $\prec_H^{RT}$ *and* S *satisfies* $\prec_H^{mvc}$.

It can be seen that the history $H1$ is mvc-opaque, with the multi-version conflict equivalent t-sequential history being: $T1T2$. 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 the following lemma.

THEOREM 5. *If a history* H *is mvc-opaque, then it is also opaque.*

Proof. Since H is mvc-opaque, it follows that there exists a t-sequential history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$ and S satisfies $\prec_H^{mvc}$. In order to prove that H is opaque, it is sufficient to show that S is legal. Since S satisfies $\prec_H^{mvc}$, from Lemma 3 we get that S is legal. Hence, H is opaque as well. □

Thus, this lemma shows that mvc-opacity is a subset of opacity. Actually, mvc-opacity is a strict subset of opacity. Consider the history $H2 = r_1(x, 0) r_2(z, 0) r_3(z, 0) w_1(x, 5) c_1 r_2(x, 5) w_2(x, 10) w_2(y, 15) c_2 r_3(x, 5) w_3(y, 25) c_3$. The set of multi-version conflicts in $H2$ 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 $H2$ is opaque with the equivalent t-sequential history being $T_1 T_3 T_2$. But there is no multi-version conflict equivalent t-sequential history. This is because of the conflicts: $(r_3(x, 5), c_2), (c_2, c_3)$. Hence, $H2$ is not mvc-opaque.

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 6. *If a history* H *is co-opaque, then it is also mvc-opaque.*

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 multi-version conflict order of H, it then implies that H is also mvc-opaque. Since S is legal, it turns out that the conflicts and multi-version 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 multi-version conflict.
- 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 $c_j = H.valWrite(r_i)$. Since, S is legal, either $c_k = c_j$ 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 $c_j = H.valWrite(r_i)$. Since, S is legal, $c_j <_H r_i <_H c_k$. Thus in this case also we get that $r_i \prec_H^{mvc} c_k$.

Thus in all the three cases, conflict among the operations in S also reults in multi-version conflict among these operations. Hence, S satisfies the multi-version conflict order. □

This lemma shows that co-opacity is a subset of mvc-opacity. The history $H1$ is mvc-opaque but not in co-opaque. Hence, co-opacity is a strict subset of mvc-opacity. Figure 2 shows the relation between the various classes.

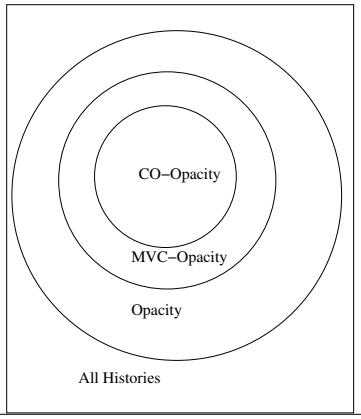

Figure 2. Relation between the various classes

4.1 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

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

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

Based on the multi-version conflict graph construction, we have the following graph characterization for mvc-opacity.

THEOREM 7. *A valid history H is mvc-opaque iff $MVCG(H)$ is acyclic.*

Figure 3 shows the multi-version conflict graphs for the histories $H1$ and $H2$.

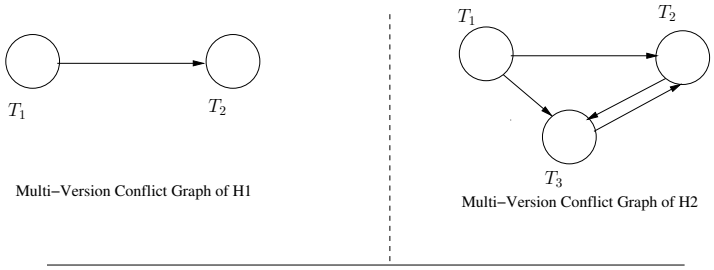

Figure 3. multi-version conflict graphs of $H1$ and $H2$

5. Discussion and Conclusion

In this paper, we have proposed multi-version conflict notion that which has enabled us to develop mvc-opacity, a sub-class of opacity, whose membership can be verified in polynomial time. Unlike co-opacity, a sub-class of opacity based on conflicts, multi-version conflict admits histories that are non-single-versioned. The polynomial time membership verification is through graph characterization which ensures that a history H is in mvc-opacity if and only if the corresponding $MVCG(H)$ is acyclic. We believe that this conflict definition is very generic and is applicable to other correctness criteria as well.

Actually, multi-version conflict notions have been proposed for multi-version databases as well [7]. 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.

For future directions, we plan to generalize this notion to non-sequential histories. Then, develop an efficient STM system that will be permissive w.r.t mvc-opacity and measure the cost of the implementation.

References

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

[2] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, feb 2008.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

[16] 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.

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

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