ALGORITHMIC ASPECTS OF MULTIVERSION CONCURRENCY CONTROL

Thanasis Hadzilacos[1] **and Christos H. Papadimitriou**[2]
Division of Computer Science
National Technical University of Athens, Greece

Abstract:*Multiversion schedulers are now a widely accepted method for enhancing the performance of the concurrency control component of a database. In this paper we introduce a new notion of multiversion serializability (MVSR) based on conflicts (MVCSR), and discuss and its relation with the well known single version conflict serializability (CSR). On-line schedulable (OLS) subsets of MVSR were defined in [PK]. We prove here that it is NP-complete to decide whether a set of schedules is OLS. We next introduce the concept of maximal OLS sets, and show that no efficient scheduler can be designed that recognizes maximal subsets of the MVSR or MVCSR schedules. Finally, a general framework for algorithms based on MVCSR is presented.*

1. Introduction

It has been recently pointed out by several authors in the area of database concurrency control that, by maintaining multiple versions of each data entity, we can achieve concurrency control schemes of enhanced performance [BHR, SR, BG, SB]. A typical multiversion concurrency control algorithm keeps several versions of each entity. When a read request arrives, the algorithm makes available to it not necessarily the latest version of the entity to be read, but the one that best serves our ultimate goals: correctness and performance. Consequently, any multiversion algorithm, besides deciding at each moment whether to grant, delay, or abort an arriving read or write step of a transaction (as all concurrency control algorithms do), must also decide which of the existing versions to present to a read step, and which, if any, to overwrite. These latter decisions, particular to multiversion concurrency control, constitute the main added complexity of this approach.

In [PK] a theoretical treatment of the subject was attempted. The class of multiversion serializable schedules (MVSR) was introduced. A schedule is MVSR if there is a mapping from read steps to previous writes of the same entity (a *version function* as we call it here) which makes the schedule equivalent to a serial schedule. The importance of this class is that it includes, besides serializable schedules, all non-serializable schedules which can be in principle output by multiversion algorithms. Since the set of schedules output by an algorithm is considered a measure of its performance, MVSR is in some sense the limit of the multiversion approach. It was then shown in [PK] that this limit is unattainable, for two reasons. First, telling whether a schedule is MVSR is NP-complete, and therefore MVSR is inappropriate as a concurrency control principle. Perhaps more importantly, it was shown that no algorithm, however slow, can output all MVSR schedules, essentially because different MVSR schedules may require conflicting version functions. A subset of MVSR was called *on-line schedulable (OLS)* if no such conflict exists among its schedules. OLS sets can be the basis of multiversion concurrency control algorithms, at least in principle.

Negative results usually lead to less ambitious, "approximate" solutions. For example, the fact that serializability is an NP-complete concept made the class of conflict-serializable schedules (CSR) more attractive. This choice found an important justification in [Ya], where it was shown that only CSR schedules can be output by locking algorithms. A schedule is CSR if a certain conflict graph is acyclic. In this paper we define a tractable subset of MVSR, called the *multiversion conflict serializable* (MVCSR) schedules. This set is a generous relaxation of CSR. We prove alternative characterizations of MVCSR, and its relationships with other classes of schedules. (With a very differ-

[1]Also, Amber Computer Services, P.O.Box 3500, Athens, Greece.

[2]Also, Stanford University, California, U.S.A. Research supported by the National Science Foundation and by the Hellenic Ministry for Research and Technology.

ent approach, a subset of MVCSR was briefly discussed in [PK] under the name DMVSR.) MVCSR seems to be the right "approximation" to the NP-complete class MVSR, as all known multiversion algorithms realize subsets of MVCSR. Unfortunately, MVCSR is itself not an OLS class, and thus the second difficulty remains.

OLS is the *main requirement for a class of schedules*, in order for it to be implementable by some multiversion algorithm. Can we at least recognize an OLS class, if we see one, in reasonable time? The answer given here is negative. We show that, given a set of shcedules, it is NP-complete to tell whether they are OLS, even if the set contains just two MVCSR schedules.

In ordinary (single-version) database concurrency control there is a maximum set of correct schedules that can be output by an algorithm, namely the class of serializable schedules (SR). Unfortunately, it is intractable. The concept of an OLS class, particular to the multiversion approach, creates an infinity of possible maximal OLS subsets of MVSR (or MVCSR). It would be a worthwhile goal to identify certain such classes that are easy to recognize, and can therefore be the basis of a concurrency control algorithm. We show a rather surprising negative result here: Any one of the *infinitely many maximal* OLS subsets of MVSR is NP-hard, and thus the schedules output by any reasonable algorithm must form a non-maximal class. A similar (slightly weaker) result holds for maximal OLS subsets of MVCSR.

In single-version concurrency control, the most liberal algorithm known is based on CSR. The conflict graph is maintained and updated, so that no cycle is ever created. Since we know that it is probably not possible to design multiversion algorithms whose performance is in any sense "best", can we use MVCSR as the basis of a new multiversion algorithm? The concept of OLS makes the answer here too, quite complex. We present a framework for multiversion algorithms based on MVCSR. Various algorithms can be derived by fixing certain non-deterministic choices in our scheme. The algorithm is as efficient as the CSR-based one, although it is considerably more general, and quite non-trivial graph theoretically. We argue that several known algorithms are *instantiations of this scheme*, and that our scheme is the most general one could expect to implement efficiently.

The rest of the paper is organized as follows. In Section 2 we describe our model giving the necessary definitions. Section 3 presents the concept of multiversion conflict serializability and its relationship with more conventional notions of correctness. Section 4 presents our first negative result, that testing whether a subset of MVSR is OLS is NP-complete. Section 5 examines maximal OLS sets of MVSR and MVCSR schedules, and their complexity. Finally in Section 6 we present a framework for multiversion algorithms based *on the concept of* MVCSR.

2. Definitions and the Model

Our model of a database and concurrency control system is described in detail in [Pa2]. In this Section we present a summary and the definitions necessary for this paper.

A database has a finite collection of *entities*. It is immaterial for our model what these entities are; they can be files, records, data items, physical disk blocks etc. No structure on the entities and no specific interpretation on their domains is assumed.

Entities are accessed atomicly by *transactions*. A transaction is a finite sequence of accesses on entities, called *steps*. There are two types of steps: read steps and write steps. A read step (respectively: write step) of transaction T_i that accesses entity x is denoted $R_i(x)$ (respectively $W_i(x)$). A write step changes the value of the entity it accessed; the new value is an uninterpreted function of the values read by the transaction in its previous read steps. The value written by $W_j(x)$ is denoted x_j; x_j is formally a triple: the entity x, the transaction T_j, and the value written.

A *transaction system* $\tau = \{T_1, \ldots, T_n\}$ is a finite set of transactions. A *schedule* s of τ is a sequence of steps in the shuffle of τ. A schedule s is *serial* if any two adjacent steps of a transaction are also adjacent in s.

A *database state* is an assignment of values to the entities. IC (stands for Integrity Constraints) is a subset of the set of all database states. A state in IC is called *consistent*.

The *read* (respectively: *write*) set of a transaction is the set of entities accessed by a read (respectively: write) step of the transaction. We say that step $R_i(x)$ reads x from transaction T_j in schedule s (and write this as $R_i(x_j)$) if $W_j(x)$ is the last write step that accesses x before $R_i(x)$ in s. The READ-FROM relation of s is: $\{(T_i, x, T_j : R_i(x_j) \text{ in } s\}$. The *view* of transaction T_i in schedule s is $\{x_j : R_i(x_j)\}$.

A basic assumption in concurrency control is that *transactions are correct*. By this statement we mean that if the view of a transaction is consistent then its output is also consistent. An immediate consequence is that *serial schedules are correct*.

Non-serial schedules are considered correct if they are *serializable*, i.e., in some sense *equivalent to* a *serial schedule* of the same transaction system. There are several notions of equivalence to which correspond different notions of correctness. For our discussion it is often useful to pad a schedule with an initial T_0 and a final T_f transaction. T_0 writes all entities and T_f reads all entities. The padded schedule of s is correct iff s is correct. Working with padded schedules makes some of the definitions and proofs easier to state and comprehend. Padding is a natural concept, as T_0 models the state of the database before schedule s and T_f models the state of the database when s finishes. We shall rarely distinguish a schedule from its corresponding padded schedule.

Our starting notion of correctness is *view-serializability (VSR)* corresponding to a notion of equivalenceof schedules called *view-equivalence.* Two schedules are view-equivalent iff they have identical READ-FROM relations. Testing whether a schedule is VSR is an NP-complete problem [Pa1].

A stronger notion of correctness is *conflict-serializability (CSR)*. Two steps of a schedule *conflict* iff they access the same entity and (at least) one of them is a write step. Two schedules are conflict-equivalent iff all pairs of conflicting steps are in the same order in both. The following is a crucial fact about this concept: If a schedule is CSR then it is VSR. Testing whether a schedule is CSR can be done in polynomial time in the number of steps in the schedule.

For the purposes of this paper, a scheduler is an algorithm that recognizes (we shall say, "outputs") a set of view-serializable schedules. The scheduler examines each step of the schedule in sequence and accepts it if the sequence of steps examined so far is a prefix of a schedule in the set it recognizes; otherwise it rejects the step (and the schedule).

So far we have described a single-version database. In a multiple-version database we have the following additions and alterations:

Each entity has an ordered set of values associated with it. Each write step adds a value at the end of the set of values of the entity if accesses.

A schedule s can be supplemented with a *version function* V to form a *full schedule* (s, V). V is a function which assigns to each read step a previous write step (not necessarily the last previous write step) of the same entity. We say that T_i reads x from T_j in (s, V), (written also $R_i(x_j)$) if V maps $R_i(x)$ to $W_j(x)$. The READ-FROM relation of the full schedule (s, V) is $\{(T_i, x, T_j : R_i(x_j) \text{ in } (s, V)\}$.

V_s, the *standard version function for* s, assigns to each read step the last previous write step of the same entity. All our definitions on single version schedulers can be restated in multiversion terminology using the standard version function. For example, a serial full schedule is (s, V_s) where s is a serial schedule.

A full schedule (s, V) is *serializable* iff there is a serial schedule, say r, such that (s, V) is equivalent to (r, V_r). Again different notions of equivalence lead to different notions of serializability -correctness. Two full schedules are *view-equivalent* iff they have identical READ-FROM relations. Fianally, schedule s is multiversion serializable (MVSR) if there is a version function V such that (s, V) is serializable.

A multiversion scheduler is an algorithm that recognizes a set of MVSR schedules. The difference from an ordinary scheduler is that the multiversion scheduler must also compute the version function that makes the schedule MVSR by assigning versions to read steps.

In our proofs of NP-hardness, we shall use the graph-theoretic concept of a *polygraph* [Pa1]. A polygraph is a triple (N, A, B) where N is a set of nodes, A is a set of arcs, i.e. ordered pairs of nodes, and B is a set of *choices*, which are ordered triples of nodes such that if (i, j, k) is a choice then (i, j) is an arc. A directed graph (N', A') is *compatible* with the polygraph (N, A, B) iff N is a subset of N', A is a subset of A' and if (i, j, k) is in B then (j, k) or (k, i) is in A'. A polygraph is *acyclic* if it has a compatible acyclic directed graph. Testing polygraph acyclicity is an NP-complete problem [Pa1].

3. Multiversion Conflict Serializability

Let us recall from the previous section the definition of conflict-serializable (CSR) schedules. A schedule s is CSR iff there is a serial schedule that is conflict equivalent with s. The *conflict graph* of s has the transactions as nodes, and an arc from A to B if a step of A is followed in s by a conflicting step of B. It can be shown that a schedule is conflict-serializable iff its conflict graph is acyclic and also iff it can be transformed to a serial schedule by a sequence of switchings of adjacent non-conflicting steps [Pa2]. The importance of CSR schedules, in addition to being easily testable, bears on the fact that they can be implemented by schedulers using locking. An important result [Ya] is that they are the only schedules that can be output by schedulers using locking.

For multiversion schedulers we define a new, relaxed, notion of conflict. We say that two steps of a schedule s conflict if the first (in the order of s) is a read step while the second is a write step on the same entity. Then a schedule s is *multiversion conflict-equivalent* to schedule s' iff all pairs of conflicting steps of s are in the same order in s' as they are in s. Note the asymmetry induced by our definition of (multiversion) conflict: That s is multiversion conflict-equivalent to s' does not imply that s' is multiversion conflict equivalent to s. (In fact the term is a bit misleading, as multiversion conflict-equivalence is not an equivalence relation for schedules.) A schedule is *multiversion conflict-serializable (MVCSR)* if there exists a serial schedule, say r, such that s is multiversion conflict equivalent to r. The multiversion conflict graph of s (MVCG(s)) has the transactions as nodes, and an arc from T_i to T_j labeled x if $W_j(x)$ follows $R_i(x)$ in s.

Theorem 1. A schedule s is MVCSR iff MVCG(s) is acyclic.

Proof: (if) Let r be a serial schedule with transactions ordered in accordance with the topological sort of MVCG(s). Let $R_i(x)$ and $W_j(x)$ be conflicting steps of s; then (T_i, T_j) is an arc in MVCG(s) so T_i comes before T_j in r. Hence the conflicting pairs of s are in the same order in r, so s is multiversion conflict equivalent to r, i.e. it is MVCSR.

(only if) Suppose s is multiversion conflict equivalent to serial schedule r. Let $<$ be the order of transactions in r. Arcs of MVCG(s) are compatible with $<$, so MVCG(s) is acyclic.$\square$

Another useful characterization of MVCSR is the following:

Theorem 2. A schedule s is MVCSR iff it can be transformed to a serial schedule by a sequence of switchings of adjacent non-conflicting steps.$\square$

However, the importance and relevance of MVCSR stems from the following result:

Theorem 3. If a schedule is MVCSR then it is MVSR.

Proof: Let s be multiversion conflict equivalent to serial schedule r. We will show a version function V such that (s, V) has the same READ-FROM relation as (r, V_r).

Let T_i read x from T_j in (r, V_r). Then step $W_j(x)$ precedes $R_i(x)$ in s (otherwise we would have a reversed conflicting pair) so V can assign x_j to $R_i(x)$.$\square$

This new concept of conflict may seem counterintuitive, since it suggests that in the multiversion model $W_j(x)-R_j(x)$ and $W_i(x)-W_j(x)$ do not conflict whereas $R_i(x) - W_j(x)$ do conflict. This has the following rationale: Let s' be a schedule that has been formed from schedule s by a number of switchings of non-conflicting adjacent steps. Then for any full schedule (s', V') there exists a version function V such that (s, V) is view-equivalent to (s', V'). To see this notice that $R_j(x)$ can "read behind" $W_i(x)$ in a $W_i(x) - R_j(x)$ switching and similarly for $W_i(x) - W_j(x)$. The same however is not true for $R_j(x) - W_i(x)$ conflicts. If these steps are switched, then T_i might read x from T_j, which is impossible when $W_j(x)$ follows $R_i(x)$. To put it in (completely) non-mathematical terms, the multiversion approach can help a read request that arrived "too late" [BG] but it can do nothing about a read request that arrived "too early".

It was shown in [PK] that MVSR is polynomial in the restricted model of transactions, in which no transaction writes an entity it has not read. [PK] then define a schedule (in the general model) to be *DMVSR* if it is MVSR once an appropriate read step is inserted before each "readless write". [IK] study similar subclasses of MVSR by considering combinations of constraints based on all four types of "conflicts": write-write, write-read, read-write, and read-read. MVCSR corresponds to their MRW set which they show to be a superset of DMVSR (MWW in their notation).

In conclusion, the "topography" of all schedules now looks like Figure 1, with an example schedule for each region:

4. The Complexity of OLS Sets of Schedules

When a multiversion scheduler accepts a read step, it decides "on the spot" which of the values of the entity to assign to it. So at any point in a schedule, the scheduler has made some decisions that possibly limit the set of continuations that may be accepted by the scheduler. Therefore, a step may very well be rejected by a scheduler despite the fact that the schedule is MVSR, because some earlier read step was given the "wrong" read-from. This observation leads to the concept of *on-line schedulable* sets of schedules, first defined in [PK].

A subset S of MVSR is called *on-line schedulable (OLS)* if, for any prefix p of a schedule in S, there is a version function V defined on p such that, each schedule pq in S has a serializable version function which is an extention of V. In other words, there are no incompatible continuations of the same prefix. OLS is necessary for a set of schedules to be recognizable by a multiversion algorithm; it is the basic limitation of the multiversion approach. [PK] prove that DMVSR is not OLS, even for the restricted (no readless writes) two step model. Since MVCSR is a superset of DMVSR it is *a fortiori* not OLS. This is established, for example, step while the second is a write step on the same entity. Then a schedule s is *multiversion conflict-equivalent* to schedule s' iff all pairs of conflicting steps of s are in the same order in s' as they are in s. Note the asymmetry induced by our definition of (multiversion) conflict: That s is multiversion conflict-equivalent to s' does not imply that s' is multiversion conflict equivalent to s. (In fact the term is a bit misleading, as multiversion conflict-equivalence is not an equivalence relation for schedules.) A schedule is *multiversion conflict-serializable (MVCSR)* if there exists a serial schedule, say r, such that s is multiversion conflict equivalent to r. The multiversion conflict graph of s (MVCG(s)) has the transactions as nodes, and an arc from T_i to T_j labeled x if $W_j(x)$ follows $R_i(x)$ in s.

Theorem 1. A schedule s is MVCSR iff MVCG(s) is acyclic.

Proof: (if) Let r be a serial schedule with transactions ordered in accordance with the topological sort of MVCG(s). Let $R_i(x)$ and $W_j(x)$ be conflicting steps of s; then (T_i, T_j) is an arc in MVCG(s) so T_i comes before T_j in r. Hence the conflicting pairs of s are in the same order in r, so s is multiversion conflict equivalent to r, i.e. it is MVCSR.

(only if) Suppose s is multiversion conflict equivalent to serial schedule r. Let $<$ be the order of transactions in r. Arcs of MVCG(s) are compatible with $<$, so MVCG(s) is acyclic.□

Another useful characterization of MVCSR is the following:

Theorem 2. A schedule s is MVCSR iff it can be transformed to a serial schedule by a sequence of switchings of adjacent non-conflicting steps.□

However, the importance and relevance of MVCSR stems from the following result:

Theorem 3. If a schedule is MVCSR then it is MVSR.

Proof: Let s be multiversion conflict equivalent to serial schedule r. We will show a version function V such that (s, V) has the same READ-FROM relation as (r, V_r).

Let T_i read x from T_j in (r, V_r). Then step $W_j(x)$ precedes $R_i(x)$ in s (otherwise we would have a reversed conflicting pair) so V can assign x_j to $R_i(x)$.□

This new concept of conflict may seem counterintuitive, since it suggests that in the multiversion model $W_j(x) - R_j(x)$ and $W_i(x) - W_j(x)$ do not conflict whereas $R_i(x) - W_j(x)$ do conflict. This has the following rationale: Let s' be a schedule that has been formed from schedule s by a number of switchings of non-conflicting adjacent steps. Then for any full schedule (s', V') there exists a version function V such that (s, V) is view-equivalent to (s', V'). To see this notice that $R_j(x)$ can "read behind" $W_i(x)$ in a $W_i(x) - R_j(x)$ switching and similarly for $W_i(x) - W_j(x)$. The same however is not true for $R_j(x) - W_i(x)$ conflicts. If these steps are switched, then T_i might read x from T_j, which is impossible when $W_j(x)$ follows $R_i(x)$. To put it in (completely) non-mathematical terms, the multiversion approach can help a read request that arrived "too late" [BG] but it can do nothing about a read request that arrived "too early".

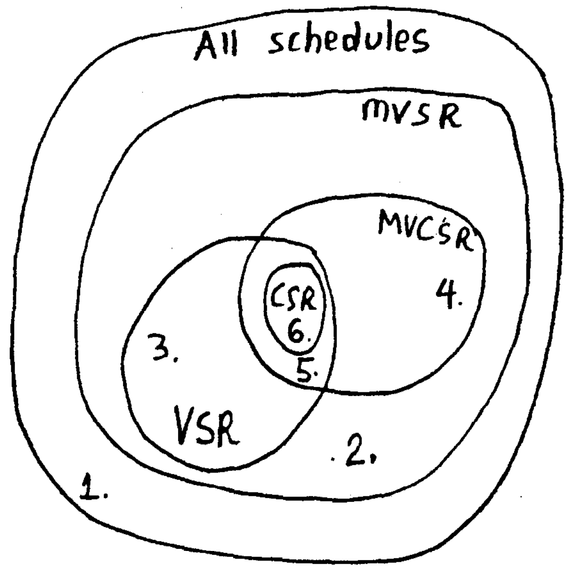
It was shown in [PK] that MVSR is polynomial in the restricted model of transactions, in which no transaction writes an entity it has not read. [PK] then define a schedule (in the general model) to be *DMVSR* if it is MVSR once an appropriate read step is inserted before each "readless write". [IK] study similar subclasses of MVSR by considering combinations of constraints based on all four types of "conflicts": write-write, write-read, read-write, and read-read. MVCSR corresponds to their MRW set which they show to be a superset of DMVSR (MWW in their notation).

In conclusion, the "topography" of all schedules now looks like Figure 1, with an example schedule for each region:

4. The Complexity of OLS Sets of Schedules

When a multiversion scheduler accepts a read step, it decides "on the spot" which of the values of the entity to assign to it. So at any point in a schedule, the scheduler has made some decisions that possibly limit the set of continuations that may be accepted by the scheduler. Therefore, a step may very well be rejected by a scheduler despite the fact that the schedule is MVSR, because some earlier read step was given the "wrong" read-from. This observation leads to the concept of *on-line schedulable* sets of schedules, first defined in [PK].

A subset S of MVSR is called *on-line schedulable (OLS)* if, for any prefix p of a schedule in S, there is a version function V defined on p such that, each schedule pq in S has a serializable version function which is an extention of V. In other words, there are no incompatible continuations of the same prefix. OLS is necessary for a set of schedules to be recognizable by a multiversion algorithm; it is the basic limitation of the multiversion approach. [PK] prove that DMVSR is not OLS, even for the restricted (no readless writes) two step model. Since MVCSR is a superset of DMVSR it is a *fortiori* not OLS. This is established, for example,



1. A non-MVSR schedule:

$$s_1 = \begin{array}{lllll} A: & R(x) & & W(x) & \\ B: & & R(x) & & W(x) \end{array}$$

2. An MVSR schedule that is not SR or MVCSR:

$$s_2 = \begin{array}{lllll} A: & W(x) & & & \\ B: & & R(x) & & W(y) \\ C: & & & R(y)W(x) & \end{array}$$

3. An SR schedule that is not MVCSR:

$$s_3 = \begin{array}{llllll} A: & W(x) & & & & \\ B: & & R(x) & & W(y) & \\ C: & & & R(y)W(x) & & \\ D: & & & & & W(y) \end{array}$$

4. An MVCSR schedule that is not SR:

$$s_4 = \begin{array}{llll} A: & R(x)W(x) & & R(y)W(y) \\ B: & & R(x)R(y)W(y) & \end{array}$$

5. An MVCSR schedule that is SR but not CSR:

$$s_5 = \begin{array}{lllll} A: & R(x)W(x) & & W(y) & \\ B: & & R(x)W(y) & & \\ C: & & & & W(x) \end{array}$$

6. Any serial schedule.

Fig. 1 Topography of the various classes with examples.

by the following pair of schedules [PK]:

$$s = \begin{array}{lllll} A: & R(x)W(x) & & R(y)W(y) & \\ B: & & R(x) & & R(y)W(y) \end{array}$$

$$s' = \begin{array}{llll} A: & R(x)W(x) & & R(y)W(y) \\ B: & & R(x)R(y)W(y) & \end{array}$$

s is equivalent to the serial schedule T_1T_2 if $R_2(x)$ reads x from T_1, and this is the only version function that makes s equivalent to a serial schedule, whereas s' is equivalent to serial T_2T_1 if $R_2(x)$ reads from T_0, this again being the only way to serialize s'.

Given a set of schedules, can we tell efficiently whether it is OLS? The answer is negative —unless of course P = NP.

Theorem 5. Testing whether a given set of schedules is OLS is NP-complete even in the restricted case of pairs of MVCSR schedules.

Sketch: NP-membership is immediate, since a pair of compatible version functions exhibits that the pair (or, more generally, set) is OLS. To establish NP-hardness, we employ a reduction from polygraph acyclicity [Pa1]. **Construction of s_1, s_2.** Without loss of generality, the polygraph P is assumed to have no arcs without corresponding choices. We construct the two schedules from the following parts: For each arc $a = (i, j)$ and corresponding choice $b = (j, k, i)$ of P we add:

(i) $W_k(b)W_i(b)R_j(b)$ to both s_1, s_2.
(ii$_1$) $W_i(b')W_k(b')R_j(b')$ in s_1i, (ii$_2$) $W_i(b'')R_j(b)W_k(b')$ to s_2.
(iii$_1$) $R_i(a)W_j(a)$ in s_1, (iii$_2$) $W_j(a)R_i(a)$ in s_2.

Here a, b, b', and b'' are entities particular to this part of P. Let p be the concatenation of parts (i), q_1 (resp., q_2) the concatenation of parts (ii$_1$) (resp. (ii$_2$), and r_1 (resp. r_2) that of parts (iii$_1$) (resp. (iii$_2$). Then, $s_1 = pq_1r_1$, $s_2 = pq_2r_2$. The longest common prefix is p. The intention of the construction is that $R_j(b)$ has the choice of reading b from transactions T_0, T_i and T_j. However parts (ii$_2$) and (iii$_1$) make $R_j(b_i)$ the only possible choice. So the problem for the scheduler is not which version to present, but whether T_k will come before T_i or after T_j in the serial schedule equivalent to s_1. This encodes the choices of P.

s_1 and s_2 are MVCSR This can be seen by noticing the special structure of the polygraphs in the proof of polygraph acyclicity (reduction from satisfiability, [Pa1], [Pa2]).

If P is acyclic, then $\{s_1, s_2\}$ is OLS. Let V be a version function that assigns the following read-froms:

$R_j(a_i)$, $R_j(b_i)$, $R_i(a_j)$. Let r be a serial schedule with transactions in the order of the topological sort of a dag compatible with P. (s_1,V) and (s_2,V) are both view equivalent to (r,V_s), so $\{s_1,s_2\}$ is OLS.

If $\{s_1,s_2\}$ is OLS, then P is acyclic. There exist version functions V_1,V_2, and serial schedules r_1,r_2 such that (s_i,V_i) is view-equivalent to (r_i,V_r) for $i = 1,2$, and V_1,V_2 agree on p. Because of q_1 and r_2, V_1 and V_2 must assign b_j to $R_i(b)$. Let $<$ be the precedence order of transactions in r_1. Let G be the graph MVCG(s_1), with the following additional arcs: for each schedule segment $W_k(b)W_i(b)R_j(b_i)$ add the arc (k,i) if $k < i$ and the arc (j,k) if $j < k$ (since T_j reads b from T_i in r_1 then either $k < i$ or $j < k$). G is acyclic (its arcs agree with $<$) and compatible with P, so P is acyclic.□

The main significance of this result is, of course, that OLS is a disappointingly complex concept, and it probably cannot be the basis of a theory, or more importantly, algorithms.

5. Maximal OLS Classes of Schedules

Among all subsets of MVSR, only the OLS ones are interesting from the point of view of concurrency control algorithms. One would like to design multiversion schedulers that output *maximal* sets of OLS schedules, that is, sets of MVSR schedules such that, if any MVSR schedule is added to the set, it ceases being OLS. There are *infinitely* many such subsets of MVSR. A scheduler that outputs a maximal OLS MVSR set is called a *maximal multiversion scheduler.*

Lemma 2. A maximal multiversion scheduler rejects a database step h, only if there is no serializable completion of the prefix output before h with the read-froms already assigned to the read steps of the prefix.□

Corrollary 1. If the version function of a prefix p of an MVSR schedule s is uniquely determined (i.e. there are no read-from choices) then p is accepted by all maximal multiversion schedulers.

Theorem 6. Suppose that a set S of schedules is (1) a subset of MVSR, (2) OLS, (3) maximal with respect to (1) and (2). Then it is NP-hard to tell whether a schedule is in S.

Sketch: The reduction is from polygraph acyclicity. Given a polygraph P we shall construct a schedule that is accepted by all maximal multiversion schedulers if P is acyclic and is accepted by none if P is not acyclic.

For each choice $b = (i,j,k)$ of P, we add the following segment to s, the schedule being constructed: $R_i(a)W_j(a)W_i(b)R_j(b)W_k(b)W_k(b')W_i(b')R_j(b')$. Note that the read-froms in s are forced ($R_i(a_0)$, $R_j(b_i)$, and $R_j(b'_i)$), so s is accepted by all maximal multiversion schedulers if it is MVSR (and of course it is rejected by all if it is not.) However, s is MVSR iff P is acyclic.□

What if we restrict ourselves to MVCSR schedules? Then the previous theorem holds in the following, slightly weaker, form, which still excludes the possibility of constructing efficient maximal multiversion schedulers.

Theorem 7. Suppose that a set S of schedules is (1) a subset of MVCSR, (2) OLS, (3) maximal with respect to (1) and (2). Let R be a scheduler that recognizes S. Then R cannot operate in polynomial time, unless P=NP.

Sketch: We start from a polygraph P of the (specially structured but sufficiently general) type produced by the reduction of satisfiability to polygraph acyclicity, and we construct a schedule s step by step, at the same time submitting each step to the scheduler R. Our construction depends not only on P, but also on the read-froms assigned by R to the read steps of s.

The basic idea of the construction if to have steps $W_i(b)W_k(b)R_j(b)$ in s for each choice $b = (i,j,k)$ of P. The intention is that T_j reads b from T_i, and the scheduler has the choice of having T_k before T_i or after T_j in the equivalent serial schedule. It then would follow that s is MVCSR and will be accepted by R iff P is acyclic.

The problem is that R may choose the read-from $R_j(b_k)$ or $R_j(b_0)$ instead of $R_j(b_i)$. To face this problem we must rename the transactions or add new transactions and steps. It follows from our detailed construction that s is accepted by R iff another polygraph P' is acyclic; P' is acyclic iff P is.□

6. A Generic Multiversion Algorithm

Can MVCSR be the basis of an interesting multiversion algorithm, just as CSR is the basis of the most "parallel" (single version) algorithm known [Pa1]? Unfortunately, OLS complicates matters enough, so that we can only define a *general framework* for such algorithms. Individual algorithms are obtained by fixing two "non-deterministic" steps of our framework. Several concurrency control algorithms that have been proposed in the literature (multiversion or not [BG, SB]) can be viewed as special cases of this scheme, in that it can be made, by appropriate choices, to output a superset of the set of schedules output by those.

An interesting exception is the multiversion timestamping as presented by [BG]. For example:

$$\begin{array}{llll} T_1 : R(x) & & & W(y) \\ T_2 : & W(y) & & \\ T_3 : & & R(y)W(x) & \end{array}$$

with timestamps $TS(T_1) < TS(T_2) < TS(T_3)$ is not MVCSR but is accepted by the multiversion timestamping algorithm of [BG], with the version function defined by the read-froms $R_3(y_2)$, $R_f(x_3)$, and $R_f(y_b)$, thus making the schedule equivalent to the serial one $T_1T_2T_3$. One could comment that this is somehow by chance, for if it so had happened that T_2 had started before T_1 (perhaps with a totally irrelevalt $R_2(z)$ step), then this schedule would not be accepted by multiversion timestamping.

For simplicity we assume that the algorithm has no prior knowledge of the transactions and the steps, before they arrive. (This is called dynamic mode of information acquisition in [Pa2]. This assumption, as usual, necessitates abort actions, which can be removed in a static version of the algorithm.)

The algorithm maintains an acyclic directed graph $G(p)$, initially empty, depending on the prefix p of the schedule output thus far, and the actions taken up to this point. The nodes of $G(p)$ are the transactions appearing in p. Suppose now that a step a of transaction A arrives. The scheduler takes the following actions:

0. If a is the first step of A, then we create a new node A in $G(p)$.

1. Suppose that a is a $W(x)$ step. For each transaction B which has read x in p we draw an arc from B to A. If a cycle is formed we abort A. Otherwise, we grant a and keep a new version of x.

2. a is a $R(x)$ step. Consider the set $T = \{B_1, \ldots, B_m\}$ of non-active transactions that wrote x in p. Among those we must choose a version for A to read. This entails the following:

2.1 Among T, consider those transactions B such that there is a path from A to B in $G(p)$ — we write this $A \rightarrow B$. Call this set I. Evidently these versions should not be read.

2.2 Consider now the set J of transactions B in T such that $B \rightarrow C$ and $C \rightarrow A$, for some C in T. These versions are also inappropriate.

*2.3 Choose a version of A to read among $T - I - J$, say B. If no such B exists, abort A. Add the arc (B, A) to G(p).

3. We must now decide for each other transaction in T, whether it will go before B or after A in the equivalent serial schedule. We carry out this in three steps:

3.1 For each C in T, if $C \rightarrow A$, then add the arc (C, B) to $G(p)$.

3.2 For each C in T, if $B \rightarrow C$, then add the arc (A, C) to $G(p)$.

*3.3 Consider all the remaining transactions in T. Divide them into two sets K and L such that there is no arc in $G(p)$ from a transaction in L to one in K. Add arcs from each transaction of K to B and from A to each transaction of L.
(* marks the nondeterministic steps.)

Theorem 6. The above algorithm can be implemented in $0(n^2)$ time per step where n is the number of transactions involved, and outputs only MVCSR schedules.□

Note that this, rather pessimistic, performance estimate is identical to the best way known for implementing the CSR scheduler. Naturally, n need not be the total number of transactions, but those that are still relevant (but see [HY] for the intricacies of this). Many different algorithms can be obtained by fixing the nondeterministic steps 2.3 and 3.3. We can argue that this is the most general technique based on MVCSR, by observing that: (i) No choice is ruled out with no reason, and (ii) The only non-forced decisions are made in step 3.3; delaying such decisions would have the effect of creating a polygraph to the maintained acyclic, which is known to be an intractable problem.

This algorithm (rather framework for algorithms) is quite open-ended. We intend to further study its various instantiations in terms of their performance, and to attack the intricate optimization problems involved.

References

[BHR] R. Bayer, H. Heller, A. Reiser. "Parallelism and Recovery in Database Systems" *ACM TODS, vol.* 5, no. 2, June 1980.

[BG] P. Bernstein and N. Goodman, "Multiversion Concurrency Control: Theory and Algorithms", *ACM TODS vol. 8*, no. 4, December 1983.

[HY] T. Hadzilacos, M. Yannakakis, "When can a Scheduler Forget About a Transaction?", manuscript, 1985.

[IK] T. Ibaraki and T. Kameda, "Multi-Version versus Single-Version Serializability", Technical Report, LCCR TR 83-1, CMPT TR 83-14, December 1983, Simon Fraser University, Burnaby, B.C. Canada.

[Pa1] C. H. Papadimitriou, "Serializability of Concurrent Database Updates", *JACM, vol. 26*, no. 4, October 1979.

[Pa2] C. H. Papadimitriou, *Theory of Database Concurrency Control*, monograph in preparation, 1985.

[PK] C. Papadimitriou and P. Kanellakis, "On Concurrency Control by Multiple Versions", *ACM TODS, vol. 9*, no. 1, March 1984.

[SB] A Silberschatz and G. Buckley, "A Family of Multi-Version Locking Protocols with no Rollbacks", TR-218, March 1983, University of Texas at Austin.

[SR] R. E. Stearns, D. J. Rosenkrantz. "Distributed Concurrency Control Using Before-Values", *Proc. ACM-SIGMOD*, pp. 74-83, 1981.

[Ya] M. Yannakakis, "Concurrency Control by Locking", *Proc. 1981 STOC.*