VLDB Journal, 3, 1-28 (1994), Walter A. Burkhard, Editor
©VLDB

Chronological Scheduling of Transactions with Temporal Dependencies

Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Witold Litwin

Received December 30, 1990; revised version received October 5, 1992; accepted August 13, 1993.

Abstract. Database applications often impose temporal dependencies between transactions that must be satisfied to preserve data consistency. The extant correctness criteria used to schedule the execution of concurrent transactions are either time independent or use strict, difficult to satisfy real-time constraints. On one end of the spectrum, serializability completely ignores time. On the other end, deadline scheduling approaches consider the outcome of each transaction execution correct only if the transaction meets its real-time deadline. In this article, we explore new correctness criteria and scheduling methods that capture temporal transaction dependencies and belong to the broad area between these two extreme approaches. We introduce the concepts of *succession dependency* and *chronological dependency* and define correctness criteria under which temporal dependencies between transactions are preserved even if the dependent transactions execute concurrently. We also propose a *chronological scheduler* that can guarantee that transaction executions satisfy their chronological constraints. The advantages of chronological scheduling over traditional scheduling methods, as well as the main issues in the implementation and performance of the proposed scheduler, are discussed.

Key Words. Transaction ordering, synchronization, execution correctness, concurrent succession, partial rollbacks.

1. Introduction

The concept of time is fundamental because it allows us to describe the order in which events occur. We use time to determine the order of past events, synchronize

Dimitrios Georgakopoulos, Ph.D., is Senior Member of Technical Staff, Distributed Object Computing Department, GTE Laboratories Incorporated, 40 Sylvan Road, MS-62, Waltham, MA 02254. Marek Rusinkiewicz, Ph.D., is Professor, Department of Computer Science, University of Houston, Houston, TX 77204-3475. Witold Litwin, Ph.D., is Professor, University Paris 9, Place du Mal de Lattre de Tassigny, Paris, France.

our current activities, and plan for future actions. As the success of many of the actions we perform depends on the earlier completion of other actions, database applications often require transactions to be executed in a particular order to produce correct results. However, this is not always easy to enforce (e.g., when transactions are submitted by different users) and, more importantly, it rules out concurrent execution of transactions that have temporal dependencies.

Many time-independent correctness criteria have been proposed in the literature to assure the correctness of the execution of concurrent transactions. While transaction management models based on such criteria successfully determine and resolve value dependencies between transactions, they do not capture temporal transaction dependencies. Time-independent correctness criteria guarantee consistency by (1) defining what constitutes a conflict between transactions and (2) specifying how conflicts should be resolved. Conflicts are defined differently, depending on the semantic information captured by each transaction management model. Serializability (Papadimitriou, 1986) allows transactions to issue only read and write operations and assumes that two transactions conflict if they issue operations on the same data item and at least one of these operations is a write. If a transaction reads a data item a and later writes b, a value dependency between a and b is assumed. Hence, serializability does not take into account any application-specific information. Semantic transaction models have been proposed to take into account the semantics of the operations in defining and resolving value dependencies (Garcia-Molina, 1983; Schwarz, 1984; Weihl, 1988, 1989; Korth and Speegle, 1988; Kumar and Stonebraker, 1988; Farrag and Ozsu, 1989; Herlihy, 1990). These models assume that transactions can issue operations semantically richer than reads and writes. Conflicts are usually defined by providing an operation compatibility table that specifies all possible value dependencies between transactions.

One of the basic reasons why most of the known correctness criteria cannot capture temporal dependencies is that temporal transaction dependencies are not always related to value dependencies. For example, consider two transactions T_1 and T_2 that have no value dependencies and do not access common data items. If, due to a consistency constraint, the result of T_2 is not useful unless T_1 is successfully completed before T_2, such transactions have a temporal dependency but no value dependencies (see the example in Section 2.1).

Another reason why most of the proposed correctness criteria cannot satisfy temporal dependencies is due to the way they resolve conflicts. Most models require that, for each pair of conflicting transactions, all operations issued by the one transaction should precede the *conflicting* operations issued by the other transaction, or vice versa. This is insufficient to satisfy temporal dependencies. For example, if a temporal dependency requires a transaction T_2 to be executed after T_1, *all* operations of T_1 must precede the operations of T_2. Schedules in which the operations of T_2 precede the operation of T_1 should not be allowed.

Some issues related to temporal dependencies have been discussed in the literature. The synchronization of site clocks and ordering of events in distributed systems has been addressed by Lamport (1978). Rusinkiewicz et al. (1990), Leu et al. (1990), and Ngu (1990) proposed approaches for the specification of temporal transaction dependencies. Both of these methods construct a precedence graph that reflects the ordering of transactions as defined by their temporal dependencies and then determine the execution order of the depended transactions. Dayal et al. (1990) proposed a transaction model that uses rules to specify the ordering of transactions. For example, if a transaction S must be executed after the end of a transaction T, this is specified by a rule that explicitly delays the execution of S until the end of T. The basic limitation of the above methods is that all related transactions must be known in advance. Furthermore, the problem of the concurrent execution of transactions that have temporal dependencies was not addressed.

In this article, we introduce a new paradigm for transaction management that captures both temporal dependencies and value dependencies between concurrent transactions. To relax the requirement that temporally dependent transactions must be known in advance, we associate with each transaction T a real time value that indicates its order in time, called the *value date* of T. Because time provides an external ordering mechanism, the transaction value dates specify the relative order of the transactions in time. As long as new transactions are assigned value dates that are later than the current time, they can be submitted even after the execution of transactions with which they have temporal dependencies has begun. A practical scheduler is proposed to guarantee that transaction executions satisfy their temporal dependencies. Transaction value dates are generated by the applications (i.e., externally to the scheduler). Temporally dependent transactions are assigned value dates that reflect the temporal semantics of the application. On the other hand, the value dates of transactions that have no temporal dependencies are selected to optimize response times and scheduler throughput.

The proposed paradigm has many advantages over other traditional transaction management approaches, even in applications in which transactions have no temporal dependencies. In addition to the absence of deadlocks and transaction restarts, the chronological scheduling paradigm allows transaction *cooperation* and can cope with *long-lived* transactions. Because these are basic requirements for supporting many non-traditional applications, the chronological scheduling paradigm can be used to support such applications in database systems (DBSs) and distributed object management systems (DOMSs) (Manola et al., 1992). In heterogeneous multidatabase systems (MDBSs) (Litwin, 1988) and DOMSs that integrate multiple autonomous systems by representing their data and functionality as objects, chronological scheduling does not violate the autonomy of the participating systems and ensures consistency in many applications.

This article is organized as follows: *Temporal transaction dependencies* are classified and discussed in Section 2. In Section 3 we define the class of *succession dependencies* and provide a theoretical framework to describe the concurrent execution of transactions having such dependencies. *Chronological dependencies* and *value dates* are defined in Section 4. In Section 5 we describe the *chronological* *scheduler*, a mechanism that satisfies application-defined succession and chronological dependencies between concurrent transactions. In Section 6 we discuss the implementation and performance of the chronological scheduler. In Section 7 we discuss the advantages of the chronological scheduling paradigm over other traditional scheduling methods in the context of various applications.

2. Temporal dependencies

Temporal transaction dependencies are constraints on the execution of transactions with respect to time or to the order in time. If temporal dependencies impose constraints that explicitly refer to time, they are *real-time dependencies*. Temporal dependencies of a single transaction can be only real-time dependencies. For example, a constraint to start the execution of a transaction at 10:15, is a real-time dependency. On the other hand, *succession dependencies* are temporal dependencies that do not impose real-time constraints on the execution of transactions. They are constraints on the ordering of transactions. In this sense, succession dependencies only implicitly refer to time.

2.1 Succession Dependencies

Transactions that have succession dependencies may or may not access common data items. For example, consider a situation in which a travel agent submits two transactions (possibly to different reservation databases) to make flight and hotel reservations for a customer. Such transactions have no value dependencies. However, a succession dependency exists between them, because the hotel reservation is useless if the flight reservation is not made. Dependent transactions in this category usually carry out successive steps that accomplish a single logical task that, for various reasons, cannot be executed as a single transaction. Succession dependencies ensure correctness by preventing an out-of-order execution of these steps. A succession dependency between two value-dependent transactions exists if one of the transactions needs to use the effects of the other transaction to produce correct results. For example, the transactions in a banking system that set a new interest rate, compute the earned interest, and deposit it to the bank accounts must be executed in this order to accomplish their objectives correctly. While succession dependencies can impose an arbitrary collection of non-conflicting constraints on the ordering of transactions, in a particular application they are defined by the application semantics. If a succession dependency between transactions is not satisfied, dependent transactions may produce invalid results in the same way that the violation of serializability can produce inconsistent retrievals (e.g., an invalid interest rate may be used to compute and display the interest earned by an account). Furthermore, it is also possible that the violation of a succession dependency may introduce inconsistencies in the database (e.g., the earned interest deposited in a bank account may be based on an invalid interest rate).

We can argue that succession dependencies can be satisfied if all succession-dependent transactions are grouped together in a single transaction or if we require the user to submit them serially. These solutions are not always adequate for the following reasons:

- Succession-dependent transactions may be submitted by different users. For example, the transaction that sets the interest rate and the transactions that compute and deposit the earned interest in bank accounts are usually submitted by different departments of a banking organization.

- Succession-dependent transactions may be submitted to different database systems. For example, if the access to the flight and hotel reservation systems is controlled by independent database systems that do not participate in the same MDBS, the transactions that make hotel and flight reservations cannot be grouped in a single transaction. If the reservation databases participate in the same MDBS, the MDBS has to submit the flight and hotel reservations as different subtransactions and must deal with their temporal dependencies.

- Predefined sets of fixed transactions may have succession dependencies. To increase performance and security, many systems (e.g., automatic teller machines, military systems, service ordering and provisioning systems) allow the user to execute a limited number of predefined transactions. In such systems, logical units of work have to be composed by successive executions of temporally dependent transactions.

- The composition of succession-dependent transactions may result in long-lived transactions (Gray et al., 1981; Garcia-Molina and Salem, 1987). Long-lived transactions are undesirable because of their poor completion rate, their large deadlock probability and the increased number of abortions and restarts they cause to other transactions. Therefore, instead of grouping transactions with succession dependencies into large long-lived transactions, it is more likely that long-lived transactions will be broken down to a number of transactions with succession dependencies (e.g., sagas; Garcia-Molina and Salem, 1987) or activities of causally dependent, detached transactions (Dayal et al, 1990).

- In many cases, succession-dependent transactions do not need to be executed serially. For example, the transactions that perform flight and hotel reservations can be interleaved. The only restriction is that the hotel reservation transaction should be committed after (and only if) the flight reservation transaction is successfully committed. Similarly, the transaction that computes the earned interest of bank accounts can be interleaved with the transaction that sets the interest rate. To guarantee correctness in this case we only need to make sure that the transaction that computes the earned interest sees the interest rate produced by the transaction that sets it.

2.2 Chronological dependencies

Because time provides an external ordering mechanism, succession dependencies can be always replaced by real-time dependencies. For example, consider the transactions that set the interest rate and compute the earned interest of bank accounts as described (Section 2.1). To satisfy the succession dependency of these transactions it is sufficient to ensure that the transaction that sets the interest rate finishes its execution at time t, and the transaction that computes the interest begins at time $t' > t$. Based on this approach, succession dependencies can be effectively transformed into real-time dependencies. However, such real-time dependencies are not always easy to enforce. In particular, while it is relatively easy to ensure that transactions are started on time, there are serious theoretical and practical problems in ensuring that transactions meet their real-time deadlines. Deadline scheduling problems have been investigated, by Abbott and Garcia-Molina (1992), among others, and we do not address them here. The difficulties in scheduling real-time transactions make it desirable to define a new subclass of real-time dependencies that do not impose *hard deadline* constraints on the transaction execution, and in addition can replace succession dependencies. We call such dependencies *chronological dependencies*.

A chronological dependency is defined by specifying the start time and the expected completion time of a transaction. For example, let us assume that the bank interest changes every week at time t. The chronological dependency of the transaction that sets the interest rate is defined by specifying t as its expected completion time. The start time of the transaction depends on its expected duration. In a chronological dependency, the expected completion time of a transaction determines its relative (*chronological*) order with respect to other transactions having chronological dependencies. Thus, chronological dependencies define a succession dependency. Succession dependencies defined by chronological dependencies are called *chronological succession dependencies* and they are satisfied only if transactions produce their effects in the chronological order specified by their expected completion time. Therefore, chronological dependencies are satisfied even if transactions do not complete by their expected completion time.

Succession dependencies between transactions can be reduced to a chronological dependency. Consider again the transactions that set and compute the interest of bank accounts. By specifying t as the expected completion time of the transaction that sets the interest rate and any time after t as the expected completion time of the transactions that must use the new interest rate, we reduce the succession dependency between these transactions to a chronological dependency. If the chronological dependencies are satisfied, the transactions that compute the interest use the new interest rate and the succession dependency is also satisfied.

In addition to replacing succession dependencies, chronological dependencies specify constraints on the start time of transactions and can be extended to capture any other real-time dependency that does not impose hard deadline constraints on the transaction execution. With the exception of real-time control systems, it is possible to reduce most temporal dependencies to chronological dependencies, particularly in applications where deadline constraints are less stringent.

3. Theory of Concurrent Successions

A theory to analyze the concurrent execution of transactions with succession dependencies can be formulated similarly to the serializability theory (Papadimitriou, 1986). A succession dependency *SD*: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$ imposes a *total order* on the execution of the transactions $T_1, \ldots, T_k$.

Definition 1. Transaction T_i succeeds transaction T_{i-1} in *SD* ($T_{i-1} \xrightarrow{s} T_i$, $1 < i \leq k$), if T_{i-1} performs its write operations earlier in time than T_i. Furthermore, if T_i reads data items which T_{i-1} writes, T_i succeeds T_{i-1} only if T_i reads all such data items after T_{i-1} writes them.

A schedule satisfies a succession dependency *SD* if it preserves the "succeeds" relationship among the transactions in *SD*. Schedules that satisfy *SD* are called *successions* with respect to *SD*.

Definition 2. A succession (S^{SD}) with respect to a succession dependency *SD*: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$ is a schedule that preserves the "succeeds" relationship among the transactions in *SD*.

A serial schedule that is also a succession is called a *serial succession*.

Definition 3. A serial succession (SS^{SD}) with respect to a succession dependency *SD*: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$ is a serial schedule in which the transactions in *SD* issue their operations as follows: T_1 issues its operations earlier in time than T_2, T_2 issues its operations earlier in time than $T_3, \ldots, T_{k-1}$ issues its operations earlier in time than T_k.

To ensure the "succeeds" relationship when the transactions in a succession dependency *SD* are interleaved, a correctness criterion *stronger than serializability* is required for the following reasons:

- not all serial schedules of the transactions in *SD* are serial successions with respect to *SD*; and

- concurrent schedules exist that are conflict equivalent[1] (Bernstein et al., 1987) to a serial succession with respect to *SD*, but they do not satisfy *SD* (i.e., they are not successions with respect to *SD*).

To illustrate these problems consider the following serial succession with respect to a succession dependency $T_1 \xrightarrow{s} T_2$:

$SS^{T_1 \xrightarrow{s} T_2}$: $w_{T_1}(a)\ w_{T_1}(b)\ w_{T_2}(c)$

Consider also the following schedules of the transactions in $SS^{T_1 \xrightarrow{s} T_2}$:

H_1: $w_{T_2}(c)\ w_{T_1}(a)\ w_{T_1}(b)$

H_2: $w_{T_1}(a)\ w_{T_2}(c)\ w_{T_1}(b)$

H_1 is serial. However, since T_2 performs its operations before T_1, H_1 is not a succession with respect to $T_1 \xrightarrow{s} T_2$. H_2 is conflict equivalent to $SS^{T_1 \xrightarrow{s}} T_2$. However, T_2 writes before T_1. Therefore, the succession dependency $T_1 \xrightarrow{s} T_2$ is not satisfied. The basic problem here is that the conflict equivalence relation as defined in serializability theory is not sufficient to ensure that a succession dependency is satisfied even if a schedule is equivalent to a serial succession. Therefore, we introduce a stronger equivalence relationship:

Definition 4. We define two schedules H and H' to be s-equivalent (succession equivalent) if:

1. They are defined over the same set of transactions that have issued the same operations.

2. The order of conflicting operations issued by non-aborted transactions is the same in both schedules. That is, for any pair of conflicting operations o_i and o_j issued by non-aborted transactions T_i and T_j respectively, if o_i precedes o_j in H then o_i precedes o_j in H'.

3. The order of the non-conflicting write operations issued by non-aborted transactions is the same in both schedules. That is, for any pair of such operations o_i and o_j issued by non-aborted transactions T_i and T_j respectively, if o_i precedes o_j in H then o_i precedes o_j in H'.

In the definition above, condition 2 ensures serializability. Condition 3 is needed to preserve the "succeeds" relationship among the transactions in a succession that issue non-conflicting write operations. Using the definition of succession equivalence we can now state the requirements to satisfy a succession dependency by defining a succession as follows:

1. As defined by the serializability theory.

Definition 5. A concurrent schedule is a succession S^{SD} with respect to a succession dependency SD: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$, if its committed projection[2] is s-equivalent to a serial succession SS^{SD} with respect to SD.

4. Chronological Dependencies, Successions, Value Dates

In Section 2.2 we described chronological dependencies as a class of real-time dependencies that do not place hard deadline constraints on transaction execution. In this section, we formally define chronological dependencies using value dates. We also discuss how value dates reduce arbitrary non-conflicting succession dependencies to a chronological succession dependency.

A chronological dependency CD_i of a transaction T_i is defined by a pair of real-time values (d_i, vd_i). The first of these values specifies the *expected duration* d_i of T_i. A good estimate of d_i, is the time required to execute T_i in the absence of any concurrency control. The second value CD_i associates with T_i is called the *value date* vd_i (Litwin and Tirri, 1988, 1989; Litwin and Shan, 1991) of T_i and indicates the expected completion time of T_i. Given a transaction T_i with value date vd_i there is no guarantee that T_i will complete its execution before vd_i. The only information vd_i provides is that the execution of T_i is likely to last until vd_i. Given the *start time* s_i of T_i (i.e., the time on which T_i should start its execution), the simplest way to determine vd_i is to compute $s_i + d_i$.[3]

In addition to placing the above constraints on transactions, chronological dependencies $CD_1, CD_2, \ldots, CD_k$ of transactions $T_1, T_2, \ldots, T_k$ also define a succession dependency among these transactions as follows:

Definition 6. Let $vd(T_1) < vd(T_2) < \ldots < vd(T_k), k > 1$. To satisfy the chronological dependencies $CD_1, CD_2, \ldots, CD_k$ we must also satisfy the succession dependency CSD: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$. We call CSD the chronological succession dependency of $T_1, T_2, \ldots, T_k$ with respect to $CD_1, CD_2, \ldots, CD_k$.

Hence, the value date of each chronologically dependent transaction determines its order in the chronological succession dependency. This is similar to the way timestamps specify the serialization order of the transaction in timestamp ordering (Bernstein et al., 1987). However, value dates are different from timestamps in the following aspects:

1. they define the order in which transactions should be completed;

2. Given a schedule H, its committed projection is the schedule obtained from H by deleting all operations that do not belong to transactions committed in H (Bernstein et al., 1987).

3. An alternative way to define CD_i is by specifying (s_i, vd_i). Given vd_i and s_i, we can easily determine $d_i = vd_i - s_i$. Therefore, either d_i or s_i can be used to define CD_i and both definitions are equivalent.

2. they are specified by the applications, not by the scheduler;

3. they carry additional information about the expected duration of each transaction.

The basic difference between a succession dependency *SD*: $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$ and a chronological succession dependency *CSD* is in the way they are specified. Succession dependencies can be specified by explicitly providing the precedence graph of the dependent transactions (Leu et al., 1990; Ngu, 1990). The main disadvantage of having to provide a precedence graph of the transactions with succession dependencies is that all such transactions must be known in advance. On the other hand, a chronological succession dependency is specified by simply providing the value dates of the transactions. Because chronological successions are specified by referring to the order that is defined externally by time, there is no need to know all dependent transactions in advance. A chronological succession dependency can be extended at any time by specifying new dependent transactions. The only requirement is that transactions cannot be assigned value dates that are earlier than the current time.

Given an acyclic precedence graph of transactions with succession dependencies, it is always possible to derive a transaction execution sequence that satisfies them. This is essentially a problem of topological sorting and it can be solved in several ways, including:

- Leu et al. (1990) proposed a method that uses Petri-nets to determine the execution order of transactions.

- Ngu (1990) proposed a synthesis procedure, called the tableau method. This method generates a dependency graph that gives all possible execution sequences of transactions.

- Dynamic programming methods, such as PERT charts, can be also used.

Given a transaction execution sequence, it is relatively simple to ensure that the value dates of the transactions are generated according to it. Therefore, *arbitrary non-conflicting succession dependencies can always be expressed as a chronological succession dependency.*

Because succession dependencies can be reduced to a chronological succession dependency, we investigate how chronological succession dependencies can be satisfied in the following discussion. In particular, we describe a transaction scheduler based on the correctness criterion proposed in Section 3. The basic requirements for such a scheduler are the following:

1. Because transactions in a chronological succession may be started before the previous transactions in the succession are completed, the transactions in a chronological succession have to be interleaved.

2. Independently of their actual start and completion times, concurrent transactions in a chronological succession must perform their conflicting operations and their non-conflicting write operations in the order determined by their value dates.

In Section 1, we discussed that most correctness criteria used to provide concurrency control do not satisfy the above requirements. Therefore, none of the traditional schedulers (Eswaran et al., 1976; Bernstein et al., 1987; Kung and Robinson, 1981) can satisfy chronological succession dependencies.

5. The Chronological Scheduler

The *chronological scheduler* or *chrono-scheduler* is a transaction management mechanism capable of enforcing application-defined chronological dependencies between concurrent transactions. While there may be transactions that do not actually have any such dependencies, the chrono-scheduler views a priori all transactions as being part of a "global" chronological succession. This is similar to the ordering of elementary events according to their occurrence in time, even if in some cases the order of a particular event in time is not important. The order of each transaction in the "global" chronological succession dependency assumed by the chrono-scheduler is specified by its value date. Clearly, if all transactions issue their conflicting operations and their non-conflicting write operations in value date order, the "global" chronological succession assumed by the chrono-scheduler and the application-defined chronological succession dependencies are all satisfied (i.e., correctness is assured). However, there is no guarantee that transactions will issue their operations in value date order and transactions are not required to predeclare the data items they access. Thus, there is no way to determine if a particular operation can be processed next. To deal with this problem the chrono-scheduler uses the same transaction synchronization approach as the time warp mechanism (TWM) (Jefferson and Motro, 1986).

TWM has been proposed first for controlling the sequence of events in distributed discrete event simulations (Jefferson, 1985*a*, 1985*b*) and was later extended for operating systems (Jefferson et al., 1987), process synchronization, and concurrency control in database systems (Jefferson and Motro, 1986). The database variation of TWM views transactions and data items as objects that encapsulate sequential processes and communicate through timestamped messages. A data object in TWM is a simple program that processes read and write messages concurrently. Transaction objects have a much more complex behavior. They execute arbitrary transaction programs, accept messages from the application and send read and write messages to data objects. TWM uses partial transaction and data object rollback for the synchronization of concurrent transactions. To implement partial rollback, objects in TWM maintain an input queue, an output queue, and a state queue. Messages received by an object are inserted in its input queue. Objects process input messages in timestamp order. When a message changes the state[4] of an object, the new state is recorded in the object state queue. In addition to changing the state of the object, the processing of an input message may cause one or more messages to be sent to other objects. Before sending such a message, the object creates and inserts a corresponding *antimessage* (Jefferson and Motro, 1986) in its output queue. Antimessages are sent only as a part of an object rollback. Whenever an object receives the antimessage of a message *m* it undoes the effects of *m*.

A transaction execution begins when a transaction object receives an invocation message from the application. TWM stamps the message with a timestamp that becomes the timestamp of the transaction. Subsequent messages originating from the same transaction carry the transaction timestamp. Objects process messages they have received in timestamp order. A synchronization conflict in TWM arises when a new message arrives out of order, that is, has a timestamp *ts* that is less than the timestamp of the last message the object has already processed. TWM neither delays read and write operations so they can be processed in timestamp order, nor aborts and restarts transactions when they issue out-of-order operations. Whenever a new message with timestamp *ts* arrives at a data object out of order, TWM performs a *time warp* that consists of the following actions: First, the object that received the out-of-order message finds the state associated with the largest timestamp that is less than *ts* and discards all states with greater timestamp from its state queue. This action rolls back the object to a time just before *ts*. Next, the object sends all antimessages that have been recorded earlier in its output queue and have timestamps greater than *ts*. These antimessages cancel the effects of messages incorrectly sent to other objects. Canceling the effects of an incorrect message at some object may also require a partial object rollback there. Finally, the object processes the new message and starts re-executing messages with timestamps greater than *ts*, again in timestamp order. When transactions complete all their operations they are committed in timestamp order.

The primary concern about TWM is that the number of rollbacks may be large and very costly. In particular, situations where TWM will allow a large number of operations to be executed and then re-execute them, in effect making ten steps forward and nine backward, are possible in principle. However, such situations can be predicted and the circumstances under which they occur will cause performance problems to most concurrency control mechanisms. For example, a large number of rollbacks may occur in TWM if many short transactions conflict with a long transaction. Under the same circumstances a 2PL scheduler will either force short transactions to wait until the long transaction finishes its execution or the long transaction will get into deadlock problems. On the other hand, schedulers that use timestamp ordering or optimistic concurrency control will repeatedly abort and restart the conflicting transactions until the long transaction is able to commit.

4. The state of a transaction object is the state of its process, while the state of a data object is its value.

Because TWM and most other schedulers do not take into account the transaction duration, they cannot deal with these problems. These concerns have been addressed in the design and implementation of the chrono-scheduler.

The main difference between the chrono-scheduler and TWM is that in the chrono-scheduler transactions are assigned value dates by the application, while TWM transactions are assigned timestamps that are generated by the scheduler. This allows the chrono-scheduler to satisfy chronological transaction dependencies and to take into account the expected duration of transactions to avoid unnecessary rollbacks. (The use of value dates in reducing the number of rollbacks in the chrono-scheduler is discussed further in Section 6.) Another difference is that the chrono-scheduler does not assume that transactions and data items are objects that encapsulate processes. The chrono-scheduler and TWM also differ in several implementation aspects. The chrono-scheduler does not keep a record of all input and output messages, and requires simpler data structures than TWM to support partial rollback. To minimize the cost of partial transaction rollbacks, the chrono-scheduler implementation takes into account the process requirements of the transaction programs. (Optimizations of checkpointing are discussed in Section 6.) However, despite their differences, both schedulers use the basic time warp principle to resolve synchronization conflicts, the same way different locking schedulers use blocking to assure database consistency.

5.1 Chrono-scheduler model

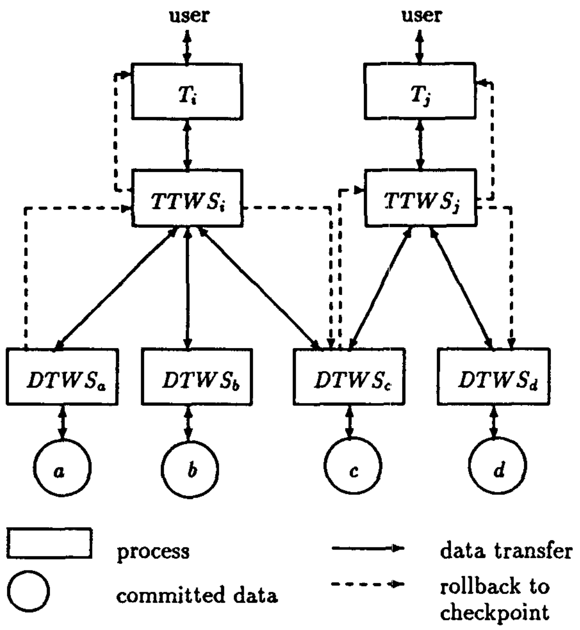
To support time warps, the chrono-scheduler uses *time warp stub* processes. For simplicity we assume that a *transaction time warp stub* (TTWS) is associated with every uncommitted transaction and that a *data item time warp stub* (DTWS) is associated with each data item.[5] The chrono-scheduler does not allow transactions to read and write data items directly. Instead, transactions issue their operations on their corresponding TTWSs, the TTWSs forward the operations to the appropriate DTWSs and only DTWSs access the data. The transaction processing model of the chrono-scheduler is shown in Figure 1.

A data item time warp stub which is associated with a data item a (DTWS$_a$) maintains the following data structures:

- an *operation queue* to record the read and write operations on a that have been issued by uncommitted transactions,
- a *version queue* to store uncommitted versions of a, and
- a *time warp vector* to keep track of the transactions that must be rolled back in the event of a time warp.

5. For efficiency reasons, several transactions or several data items can be associated with a single time warp stub.

Figure 1. Chrono-scheduler transaction processing model



The operation queue and the version queue of the data stubs are similar to the input and state queue of the data objects in TWM. Unlike objects in TWM, data stubs do not keep track of output messages and do not maintain output queues. The time warp vector is used to record the IDs of transactions that must be rolled back during a time warp. All data stub queues are maintained in ascending value date order.

A transaction time warp stub associated with an uncommitted transaction T_i (TTWS$_i$) maintains the following information:

- an *operation log* to keep a record of the operations issued by T_i,
- a *state queue* to record checkpoints of the transaction process, and
- a *time warp vector* to keep track of all data items that must be rolled back in the event of a time warp.

Transaction stubs append checkpoints of the transaction process in the state queue in the order they take them. TTWSs maintain no input queue; unlike TWM, the chrono-scheduler assumes that it has no control over transaction input. Operations are inserted in the operation log according to the order they are issued by the transaction.

5.2 Processing On-Time Operations

To illustrate the processing of an operation, suppose that a transaction T_i issues an operation o_i on a data item a. To process o_i, the transaction time warp stub TTWS$_i$ of T_i sends a message containing o_i to the time warp stub $DTWS_a$ that is associated with a. On receiving the message, $DTWS_a$ inserts o_i in its operation queue in a position determined by the value date vd_i of the transaction T_i that issued it. Because we assume here that T_i issued o_i in value date order, $DTWS_a$ appends o_i at the end of the queue. If o_i is a read operation, $DTWS_a$ simply returns the latest version of a to T_i. If o_i is a write operation, it causes the creation of a new version of a. Because all messages that arrive at $DTWS_a$ carry operations on a, we use r_{T_i} and $w_{T_i}(va_i)$ to denote read and write operations issued by a transaction T_i. The new version of a that is created by a write operation of T_i is denoted by va_i.

Figure 2 shows the effects of a write operation $w_{T_8}(va_8)$ which is issued by transaction T_8 and arrives at $DTWS_a$ in a correct value date order. We assume that transactions T_1, T_2, T_3, T_5 and T_7 with value dates $vd_1 < vd_2 < vd_3 < vd_5 < vd_7$ have performed their operations and have created versions va_1, va_3 and va_7 before the arrival of $w_{T_8}(va_8)$. If T_8 issues a read operation instead of $w_{T_8}(va_8)$, then va_8 is not created and va_7, the latest version of a, is returned to T_8.

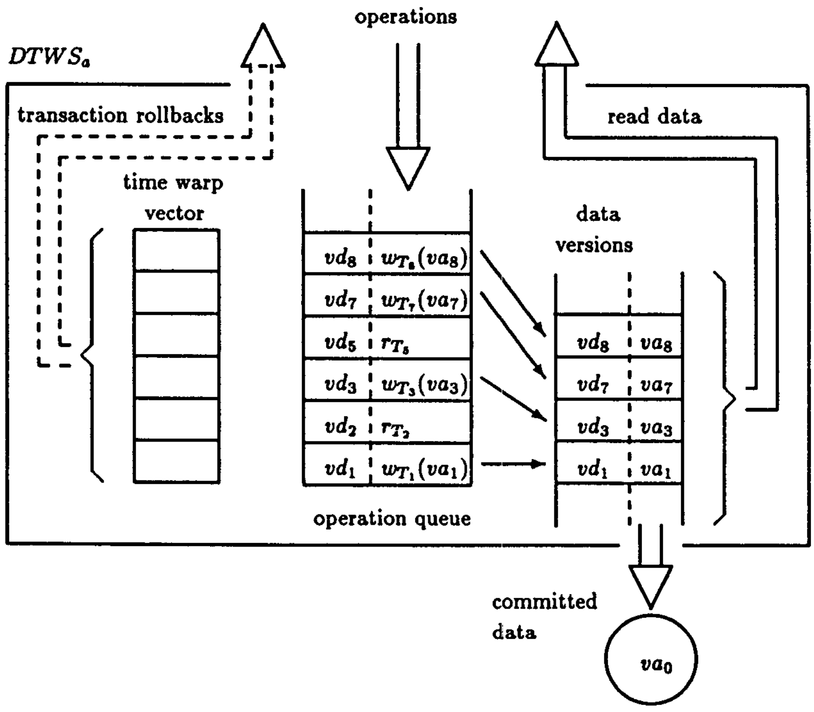
5.3 Processing Late Operations

Let us assume that a transaction T_i with value date vd_i issues an operation o_i on a data item a that arrives out of value date order at $DTWS_a$. Depending on whether o_i is a read or a write operation there are two possible cases. If o_i is a read operation, $DTWS_a$ returns the latest version of a with value date earlier than vd_i. This is similar to the multiversion concurrency control (Reed, 1983). For example, if a read operation r_{T_4} with value date vd_4 arrives at the data stub depicted in Figure 2, $DTWS_a$ returns va_3 to T_4.

If o_i arrives late at $DTWS_a$ and it is a write operation $w_{T_i}(va_i)$, it triggers a time warp. A time warp in the chrono-scheduler consists of partial rollbacks of data items and transactions. Data items are rolled back by their data stubs while transaction rollbacks are performed by the transaction stubs. To carry out the time warp that is caused by a late write operation $w_{T_i}(va_i)$, the data and transaction stubs perform the following actions:

1. *Partial data item rollback*: $DTWS_a$ rolls back the data item a to its latest version with value date earlier than vd_i. Next, $DTWS_a$ performs $w_{T_i}(va_i)$ and resumes the processing of new operations on a in value date order. If any other transaction T_j with value date $vd_j > vd_i$ has read or created a version of a that was discarded during the the rollback of a, T_j must be rolled back. To initiate the rollback of T_j, $DTWS_a$ sends a *rollback* message to $TTWS_j$. The data item stub actions that rollback a data item are described in Section 5.3.1.

Figure 2. Operation $w_{T_8}(va_8)$ in correct value date order at $DTWS_a$

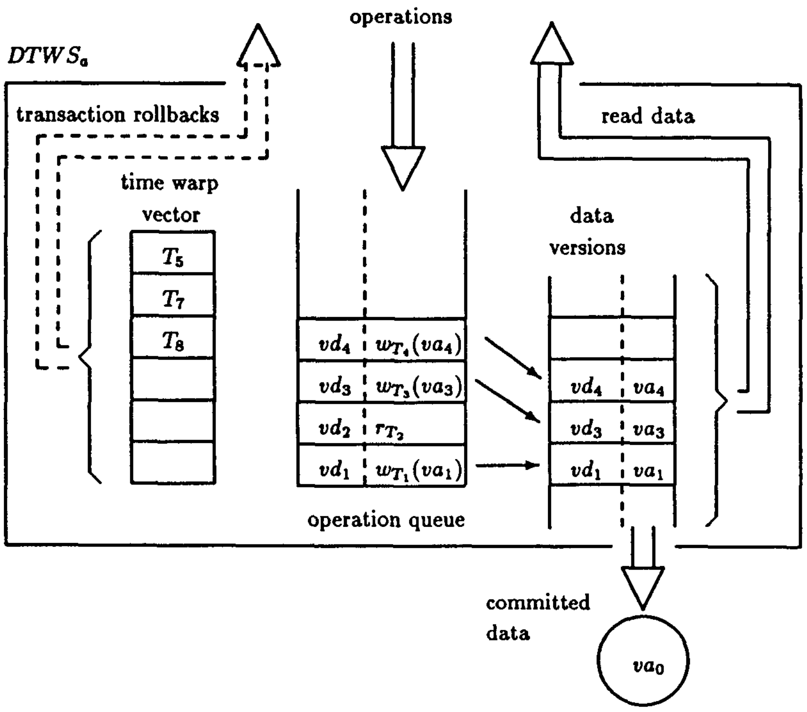


2. *Partial transaction rollback*: On receiving a rollback message from $DTWS_a$, $TTWS_j$ rolls back the process of transaction T_j to its latest state immediately before it issued its *first* operation on *a*. If T_j has performed a *write* operation on another data item *b* after performing its operation on *a*, *b* must be also rolled back to its latest version immediately before vd_j (i.e., its value before it received the write operation from T_j). To initiate the rollback of *b*, $TTWS_j$ sends a *rollback* message to $DTWS_b$. The rollback of a transaction by its transaction stub is explained further in Section 5.3.2.

3. *Cascading partial rollbacks*: Transaction stubs that receive rollback messages behave like $TTWS_j$ in item 2, while the data item stubs behave like $TTWS_a$ in item 1.

A clarification must be made here. When a message arrives at a data- or a transaction time warp stub it causes an interrupt. If no other operation is currently executed there, the newly arrived message is processed immediately. Otherwise, the value date of the operation currently being serviced is compared with the value date of the operation in the message. If the value date of the new operation is

Figure 3. Operation $w_{T_4}(va_4)$ Arrives Out of Value Date Order at $DTWS_a$



found to be later, or the new operation is a read, the interrupted execution resumes. Otherwise, the new operation is a late write and a time warp begins immediately. This is required to avoid waiting for the completion of unnecessary computations.

5.3.1 Partial Rollback of Data Items. To describe in detail how $DTWS_a$ rolls back data item a to its latest version with value date earlier than vd_i, let us assume that the version queue of $DTWS_a$ contains versions va_1,va_2, . . ., va_m, with value dates $vd_1 < vd_2 \ldots < vd_m$, $1 \leq m$. Whenever a message containing a write operation $w_{T_i}(va_i)$ with value date vd_i arrives at $DTWS_a$ and $vd_1 <$ rti vd$_2 \ldots < vd_l < vd_i < vd_{l+1} \ldots < vd_m$, $1 \leq l < m$, $DTWS_a$ discards all versions of a with value dates greater than vd_i, i.e., va_{l+1}, va_{l+2}, . . ., va_m are removed from the version queue. This makes a_l (the version of a with the latest value date that is earlier than vd_i) the current version of a. Similarly, all operations with value dates greater than vd_i are removed from the operation queue. Next, $DTWS_a$ performs $w_{T_i}(va_i)$ on a and resumes the processing of new operations in value date order. Figure 3 illustrates the effects of a write operation $w_{T_4}(va_4)$ after it has been processed by the $DTWS_a$ ($DTWS_a$ before the processing of $w_{T_4}(va_4)$ is depicted in Figure 2).

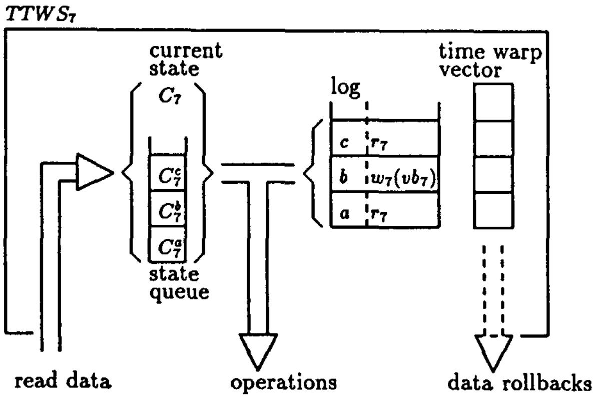
Now assume that transactions T_1, T_2, . . ., T_n, with value dates later than T_i's, have performed operations on *a* before T_i issues $w_{T_i}(va_i)$. To undo the performed computation, all these transactions must be rolled back to their state immediately before they issued their first operation on *a*. To initiate the rollback, $DTWS_a$ places the identifiers of the transactions T_1, T_2, . . ., T_n in its time warp vector, discards their operations from the operation queue and then sends *rollback* messages to their time warp stubs to inform them about the expected rollback. For example, after the data stub depicted in Figure 3 rolls back *a* and processes $w_{T_4}(va_4)$, $DTWS_a$ places the identifiers of the transactions T_5, T_7, and T_8 in its time warp vector and sends rollback messages to their time warp stubs. The actual rollback of each of these transactions is accomplished by its time warp stub.

5.3.2 Partial Rollback of Transactions

To explain the actions of transaction stubs in the event they receive a rollback message from a data stub, let us assume that $TTWS_j$ (the transaction stub of T_j) receives a rollback message from $DTWS_a$. To support a unit of rollback smaller than a transaction, the time warp stub of each transaction takes a checkpoint of the transaction process every time the transaction issues an operation on a data item for the *first* time. More specifically, when a transaction T_j issues its first operation o_j on a data item *a*, $TTWS_j$ takes a checkpoint C_j^a of T_j and then forwards o_j to the $DTWS_a$. Hence, the number of checkpoints taken per transaction is equal to the number of data items it accessed. In the event $TTWS_j$ receives a rollback message from $DTWS_a$, $TTWS_j$ finds C_j^a in its state queue and restores it as the current state of the transaction process (the state of a transaction process is discussed further in Section 6.2). This rolls back T_j to its state immediately before it accessed *a*. However, rolling back the transaction process is not sufficient to undo the effects that the write operation T_j has performed on other data items from the time it issued its first operation on *a* until the time $TTWS_j$ received a rollback message from $DTWS_a$. To determine all such write operations, $TTWS_j$ keeps a log of the operations issued by T_j. To undo their effects, $TTWS_j$ sends *rollback* messages to all data stubs that have processed them. Rollback messages carry the value date vd_j of T_j. On receiving a rollback message from the transaction stub of T_j, data stubs roll back the data items to their version immediately before they perform any write operation issued by T_j, that is, to their latest version with value date earlier than vd_j. The behavior of data stubs that receive a rollback message is similar to the behavior of $DTWS_a$ when it receives a late write operation.

To illustrate the behavior of a transaction stub in the event of a time warp, consider a transaction T_7 that reads the value of the data item *a* and then writes *b* and reads *c*. Before processing each of these operations the transaction time warp stub of T_7 ($TTWS_7$), takes a checkpoint of the process of T_7 and records it in its state queue. In addition, $TTWS_7$ records all operations of T_7 in its operation log. Figure 4 illustrates $TTWS_7$ after the completion of these actions. On receiving the rollback message from $DTWS_a$ (Figure 3), $TTWS_7$ searches its state queue and finds

Figure 4. ***TTWS*$_7$ After Processing Operations on Data Items *a*, *b* and *c*.**



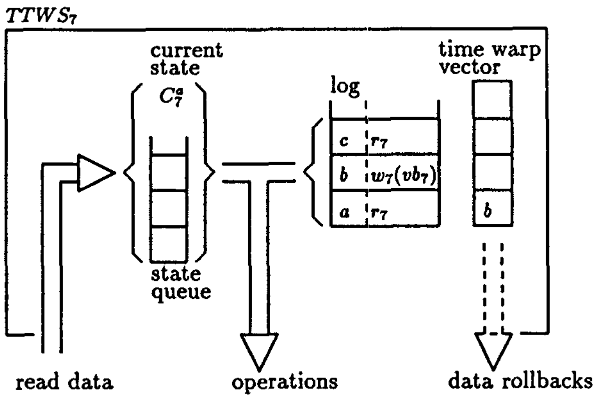
C_7^a, the checkpoint of T_7 taken immediately before T_7 issued its first operation on a. $TTWS_7$ discards all checkpoints taken after C_7^a and restores C_7^a as the current state of T_7. This rolls back T_7 to its state immediately before it accessed a. Before resuming the processing of T_7, $TTWS_7$ searches its log to determine whether T_7 has performed any write operations after reading a. It discovers the write on b, places the ID of b in its time warp vector, and sends a rollback message to $DTWS_b$ with value date vd_7. Figure 5 illustrates $TTWS_7$ after the completion of these actions.

While undoing write operations may require additional data and transaction rollbacks, undoing read operations is accomplished by the rollback of the transaction process. If T_j has performed read operations after it issued its first operation on a, it "forgets" the values of data items it read when its process is rolled back to checkpoint C_j^a. If, after resuming its execution from C_j^a, T_j needs the values of data items that were discarded during the rollback to C_j^a, T_j must issue new read operations.

5.4 Implicit Commitment and Correctness

Transactions in the chrono-scheduler are *implicitly committed* as they get older without the explicit negotiation and agreement of the participants that is required by 2PC (Gray, 1978) and other traditional commitment protocols (Skeen, 1982*a*, 1982*b*). The commitment time of transactions in the chrono-scheduler depends on the *global value date* (GVD). If the chronological scheduler uses a single real-time clock (e.g., in a centralized system), then the GVD is its current value. In distributed systems in which all site clocks are kept precisely synchronized, the GVD can be defined as the current value of any site clock. One way to keep the site clocks synchronized is to use a variant of Lamport's scheme (providing that value dates are not used in any way to set the values of the local site clocks) (Lamport, 1978). Alternatively, the

Figure 5. ***TTWS*$_7$ Immediately After a Rollback by *DTWS*$_a$**



site clocks can be kept aligned by using the clock synchronization utilities available in many operating systems. In distributed environments in which clocks cannot be synchronized (because of site autonomy or geographical time difference), the GVD is the earliest of the values of all the real-time clocks used by the chrono-scheduler to determine whether transactions have reached their value dates. The GVD becomes similar to the *global virtual time* in TWM (Jefferson and Motro, 1986) if it is defined as the instance of real time that is the minimum of:

- The earliest value date assigned to an uncommitted transaction.
- The earliest possible value of all the real time clocks that can be used as transaction value dates.

Because no operation can cause a rollback to a time earlier than its value date and no new transaction can have an earlier value date than GVD, it is guaranteed that no data item or transaction will ever be rolled back to a value date earlier than the earliest value date assigned to an active transaction. Therefore, assuming that active transactions terminate normally, the chrono-scheduler makes progress. A similar claim was proven by Jefferson (1985*b*) and Jefferson and Motro (1986).

The chrono-scheduler commits a transaction T_i with value date vd_i if it satisfies the following requirements:

- T_i has finished its execution,
- GVD $\geq vd_i$, and
- all transactions with value dates earlier than vd_i are already committed.

To support implicit commitment, the chrono-scheduler must keep track of all active transactions. When a transaction T_i is committed, the data item versions it produces overwrite the previously committed values and become visible to the outside world. Furthermore, all records related to T_i are no longer needed and can be discarded. As long as T_i is uncommitted its effects are buffered in the version queues of the appropriate time warp stubs and are not visible to the outside world.

The chrono-scheduler satisfies chronological and succession dependencies for the following reasons:

- Conflicting operations are executed and re-executed by the time warp stubs until they are performed in value date order.

- Although non-conflicting write operations may be performed in an order different than their value date order, their results do not become visible before the transactions that issued them commit. Thus, the implicit commitment of transactions in value-date order guarantees that non-conflicting write operations appear to the outside world as if they have been performed in value date order.

6. Implementation and Performance of Chrono-scheduler

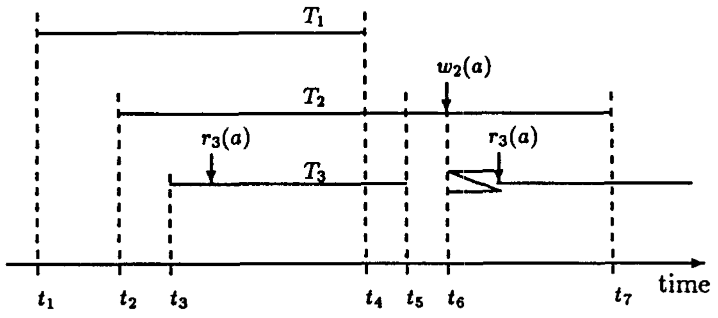
A prototype of a distributed chrono-scheduler has been developed as an object-based component of a DOMS at GTE Laboratories Incorporated. The chrono-scheduler prototype runs on several UNIX platforms. Transactions and stubs run as UNIX processes in a network of workstations and communicate by exchanging TCP/IP messages. Below, we will discuss the main issues affecting the performance of the chrono-scheduler, namely the choice of the value dates for transactions and the checkpointing of the transaction processes.

6.1 Choice of Transaction Value Dates

A performance analysis of TWM is presented by Berry (1986). Assuming that transactions have no chronological dependencies and their value dates are selected the same way as timestamps, the behavior and performance characteristics of TWM and the chrono-scheduler are identical.

While the correctness of the chrono-scheduler is not affected if timestamps are used as value dates of transactions that have no chronological dependencies, the commitment of younger, shorter transactions must wait until older, longer transactions have committed. Furthermore, younger, shorter transactions are subject to rollbacks triggered by conflicting older, longer transactions. To reduce rollbacks and transaction waiting in these situations, value dates should *reflect the duration* of the transactions. For example, if transactions have no chronological dependencies and the value date of each transaction is generated by adding a relatively accurate

Figure 6. Effects of Value Date Assignments on Transaction Execution



estimate of its duration to the time it begins its execution, the chrono-scheduler performs better than TWM.

To illustrate this, consider the concurrent execution that is depicted in Figure 6. Transaction T_1 starts its execution at time t_1 and finishes all its operations at time t_4. Transactions T_2 and T_3 start their execution at time t_2 and t_3, respectively. At time t_5, T_3 finishes all its operations while T_2 continues its execution. Assuming that the transaction value dates are selected the same way as timestamps (i.e., $vd_1 = t_1$, $vd_2 = t_2$ and $vd_3 = t_3$), the chrono-scheduler behaves like TWM. In particular, T_3 cannot commit until T_2 commits. Furthermore, any write operation that is issued by T_2 after t_5 and conflicts with an operation performed earlier by T_3, causes a transaction time warp that partially rolls back T_3. For example, if T_3 performs $r_3(a)$ before t_5, and T_2 issues $w_2(a)$ at time $t_6 > t_5$, the write operation triggers a time warp and the chrono-scheduler partially rolls back T_3 to its state immediately before it issued $r_3(a)$ for the first time. When the time warp is completed, T_3 starts re-executing from the point it issues $r_3(a)$.

Suppose now that the value dates of the transactions reflect their duration (i.e., $vd_1 = t_4$, $vd_2 = t_7$, and $vd_3 = t_5$). In this case, $r_3(a)$ and $w_2(a)$ are executed in value date order, there is no time warp, and T_3 commits at time t_5. Therefore, if the value date of each transaction reflects its duration, the chrono-scheduler never rolls back transactions after they finish all their operations. Furthermore, there is no transaction waiting. Since the chrono-scheduler requires fewer rollbacks and less transaction waiting than TWM, its throughput and response times are better than TWM.

Further improvements in the throughput and response times of the chrono-scheduler can be achieved if transactions that have chronological dependencies are also assigned value dates that reflect their duration. This basic principle is satisfied if the application generates unique transaction value dates as follows:

- For transactions having an application-defined chronological succession dependency $T_1 \xrightarrow{s} T_2 \xrightarrow{s} \ldots \xrightarrow{s} T_k$, the value date vd_i of each transaction T_i,

$1 < i \leq k$, must be set as follows. If T_i can be started at time s_i and $s_i + d_i > vd_{i-1}$ then $vd_i = s_i + d_i$. Otherwise, $vd_i = vd_{i-1} + dt$, where dt is a small quantity of time.

- For a transaction T_i that has no chronological dependencies, a good choice of value date is $vd_i = s_i + d_i$.

6.2 Optimization of Checkpointing

The performance of the chrono-scheduler in the UNIX environment can be substantially improved by reducing the information that must be saved when the chrono-scheduler takes a checkpoint of a transaction process. Transaction checkpoints are expensive because the transaction stub must stop the transaction process and make a copy of its process address space. The address space of a process comprises a data part and a process control information part. The data part consists of five separate areas: the text area that stores the executable code of the process, the initialized and uninitialized global variables, the data portion of the stack that contains parameters for function calls and the local variables of each function, the heap that contains the dynamic data structures, and the general-purpose registers. The process control information consists of the contents of the control registers, for example, program counter, stack pointer, and the return address of the process which is recorded in the first stack frame.

Clearly, the checkpoint of a process must always contain the process control part. However, there is no need to save the entire data part of the process address space. For example, because the transaction program does not change in the various states of its execution, there is no need to save the text area of the process address space. The space allocated for global variables is relatively small and it is not expensive to keep a copy of it. Furthermore, individual variables can be located and saved only if their value has been changed. The most expensive task in taking a process checkpoint is saving the process heap.

Unlike the area for global variables, the heap can be very large, and locating, saving, and restoring dynamic data structures can be very expensive. The latter is due to the space allocation policy of the dynamic memory allocation procedure. When a dynamic data structure is extended, the dynamic memory allocation procedure allocates the first available slot in the heap without any consideration to where the other components of the data structure reside. Thus, locating the components of a dynamic data structure requires a traversal. Furthermore, in most cases the restoration of a dynamic data structure in the heap requires its reconstruction.

One solution that avoids saving and restoring the entire heap area is to restrict transaction programs from using dynamic data structures. Since there are cases in which this may be too restrictive, another solution is to implement a new dynamic memory allocation procedure that uses a policy that reduces the cost of locating, storing, and restoring updated data structures, (e.g., one that keeps the components of dynamic data structures together in clearly identifiable and continuous areas of the heap).

Stack recovery is another area in which it is possible to reduce the cost of checkpoints. In the event of a checkpoint the process stack contains the parameters and local variables of all functions/procedures currently being executed. Since checkpoints are taken before a transaction performs it first operation on a data item, we can eliminate the need to save the stack by requiring transactions to issue all data manipulation operations from their main programs. By taking the issues above into account, we have introduced several classes of transactions that require increasingly less information to be saved when the chrono-scheduler takes their checkpoint. Maximum checkpointing efficiency is achieved for the class of transactions that do not require the chrono-scheduler to save the stack and heap of their process.

7. Conclusion

In addition to satisfying succession and chronological dependencies, the chronological scheduling paradigm has many advantages over traditional transaction management approaches. Compared to two-phase locking, timestamp ordering, and the optimistic concurrency control schemes used in traditional DBSs, the chrono-scheduler neither blocks nor aborts transactions to resolve synchronization conflicts. Thus, chronological scheduling does not suffer from deadlocks and starvation. Unlike most traditional schedulers the chrono-scheduler takes into account the duration of the transactions and can deal more effectively with problems caused by long-lived transactions. Another significant advantage of the chrono-scheduler over traditional schedulers is that it *does not enforce transaction isolation.* Since transactions are allowed to observe the effects of uncommitted transactions concurrency is increased and becomes possible to support applications that require close cooperation of transactions accessing common data items. The chrono-scheduler effectively deals with cascading aborts by allowing transactions to commit only in value date order. These are highly desirable for non-traditional applications, such as CAD/CAM, where long-lived transactions manipulate a relatively small number of complex data objects. By viewing interdependent design steps as a collection of transactions that have chronological succession dependencies, the chronological scheduling paradigm allows design steps to share uncommitted data (i.e., the drafts of the objects being designed, without violating serializability). Because the chronological scheduling paradigm copes with long-lived transactions and allows transaction cooperation, it can be used in DBSs and DOMSs to provide transaction support for non-traditional applications.

Another significant application of the chronological scheduling paradigm is the synchronization of multidatabase transactions in heterogeneous MDBSs (Breitbart and Silberschatz, 1988; Breitbart et al., 1991; Georgakopolous et al., 1991) and DOMSs (Manola et al., 1992). In such environments, multidatabase transactions access data items stored in multiple autonomous local database systems. Whenever a multidatabase transaction issues an operation on a data item for the first time, and no other uncommitted multidatabase transaction has accessed it before, the chrono-scheduler associates a time warp stub to the data item and the copies its value from the appropriate local database into the version queue of the stub. Operations performed by uncommitted multidatabase transactions read and create uncommitted versions of data items under the control of the chrono-scheduler. When a multidatabase transaction commits, the chrono-scheduler overwrites the values of the data items in the local databases with their new versions that have been created by the transaction. Chronological scheduling in a multidatabase environment assures that the subtransactions of all multidatabase transactions satisfy the same chronological succession dependency in every participating database system. Thus, it guarantees quasi-serializability (Du and Elmagarmid, 1989) and global committativity (Elmagarmid et al., 1991). Furthermore, if we assume that all participating local database system use rigorous schedulers (Breitbart et al., 1991), chronological scheduling enforces global serializability.

In distributed environments, such as distributed DBSs, MDBSs, and DOMSs, the chronological scheduler needs no additional mechanisms to deal with the distribution of the scheduler. Like timestamps in timestamp ordering, value dates in the chrono-scheduler carry all information necessary to perform concurrency control in a distributed environment. Furthermore, chronological scheduling allows the individual sites of a distributed system to decide independently whether to commit the subtransactions of a distributed transaction, by comparing their value dates to the value of the local site clock. This significantly reduces the explicit negotiation and agreement of the participants required to reach a decision about the outcome of each transaction by commitment protocols such as 2PC (Gray, 1978), 3PC (Skeen, 1982*a*, 1982*b*) and their variations.

Acknowledgments

We thank Raj Batra, Mark Hornick, Ole Anfindsen, and Piotr Krychniak for their helpful comments and reviews.

References

Abbott, R. and Garcia-Molina, H. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513-560, 1992.

Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

Berry, O. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism*. PhD thesis, University of Southern California, 1986.

Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M., and Silberschatz, A. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954-960, 1991.

Breitbart, Y. and Silberschatz, A. Multidatabase update issues. *Proceedings of ACM SIGMOD International Conference on Management of Data*, Chicago, 1988.

Dayal, U., Hsu, U., and Ladin, R. Organizing long-running activities with triggers and transactions. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Atlantic City, NJ, 1990.

Du, W. and Elmagarmid, A. QSR: A correctness criterion for global concurrency control in InterBase. *Proceedings of the 15th International Conference on Very Large Databases*, Amsterdam, 1989.

Elmagarmid, A., Jing, J., and Kim, W. Global commitment in multidatabase systems. Technical Report CSD-TR-91-017, Department of Computer Science, Purdue University, 1991.

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624-633, 1976.

Farrag, A. and Ozsu, T. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503-525, 1989.

Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186-213, 1983.

Garcia-Molina, H. and Salem, K. SAGAS. *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Francisco, 1987.

Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A. On serializability of multidatabase transactions through forced localL conflicts. *IEEE Proceedings of the 7th International Conference on Data Engineering*, Kobe, Japan, 1991.

Gray, J.N. *Operating Systems: An Advanced Course, Lecture Notes in Computer Science.* New York: Springer-Verlag, 1978.

Gray, J.N. The transaction concept: Virtues and limitations. *Proceedings of the 7th International Conference on VLDB*, Cannes, France, 1981.

Gray, J., Homan, P., Korth, H., and Obermark, R. A straw man analysis of the probability of waiting and deadlock in a database system. Technical Report RJ3066, IBM Research, San Jose, February, 1981.

Herlihy, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96-124, 1990.

Jefferson, D. Fast concurrent simulation using the time warp mechanism. *SCS Simulation*, 15(2):96-124, 1985*a*.

Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):96-124, 1985*b*.

Jefferson et al. Distributed simulation and time warp operating system. *Operating Systems Review*, 21(5):96-124, 1987.

Jefferson, D. and Motro, A. The time warp mechanism for database concurrency control. *Proceeding of the IEEE International Conference on Data Engineering*, Los Angeles, 1986.

Korth, H.F. and Speegle, G.D. Formal model of correctness without serializability. *Proceedings of ACM SIGMOD International Conference on Management of Data*, Chicago, 1988.

Kumar, A. and Stonebraker, M. Semantics based transaction management techniques for replicated data. *Proceedings of ACM SIGMOD International Conference on Management of Data*, Chicago, 1988.

Kung, H.T. and Robinson, J.T. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213-226, June 1981.

Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558-565, July 1978.

Leu, Y., Elmagarmid, A., Rusinkiewicz, M., and Litwin, W. Exterding the transaction model in a multidatabse environment. *Proceedings of the 16th International Conference on Very Large Databases*, Brisbane, Australia, 1990.

Litwin, W. From database systems to multidatabase systems: Why and how. *British National Conference on Databases*. Cambridge Press, 1988.

Litwin, W. and Shan, M.C. Value dates for concurrency control and transaction management in interoperable systems. *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

Litwin, W. and Tirri, H. Flexible concurrency control using value dates. Technical Report 845, INRIA, May 1988.

Litwin, W. and Tirri, H. Flexible concurrency control using value dates. In: Gupta, A., ed., *Integration of Information Systems—Bridging Heterogeneous Databases*. Washington, DC: IEEE Press, 1989.

Manola, F., Heiler, S., Georgakopoulos, D., Hornick, M., and Brodie, M. Distributed object management. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):5-42, March 1992.

Ngu, A.H.H. Specification and verification of temporal relationships in transaction modeling. *Information Systems*, 15(2):257-267, 1990.

Papadimitriou, C.H. *The Theory of Concurrency Control.* Rockville, MD: Computer Science Press, 1986.

Reed, D.P. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1) 1983.

Rusinkiewicz, M., Elmagarmid, A., Leu, Y., and Litwin, W. Extending the transaction model to capture more meaning. *SIGMOD record*, 19(1):3-7, 1990.

Schwarz, P.M. *Transactions on Typed Objects*. PhD thesis, Carnegie-Mellon University, December 1984.

Skeen, D. Nonblocking commit protocols. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Orlando, FL, 1982*a*.

Skeen, D. A quorum based commit protocol. *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, CA, 1982*b*.

Weihl, W. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):205-218, 1988.

Weihl, W. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249-282, 1989.