Performance Modeling of Database and Simulation Protocols: Design Choices for Query Driven Simulation

John A. Miller* and Nancy D. Griffeth†

*Department of Computer Science, University of Georgia
†Bellcore

Abstract

Query driven simulation is an approach to simulation modeling and analysis that attempts to automate the storage and retrieval of simulation data and simulation models. Object-oriented database systems provide sufficient data and behavioral structuring mechanisms to support demanding applications like query driven simulation. This paper represents a preliminary modeling study of some of the design choices related to the operation of active objects in an object-oriented database system. We wish to select a collection of database and simulation protocols that will support a highly concurrent, high performance query driven simulation system.

1. Introduction

Because of the difficulty of simulating large complex systems with traditional tools, new approaches have and are being developed. One group of interrelated approaches attempts to simultaneously make simulation modeling and analysis easier while at the same time providing enough power to handle more complex problems. This group includes the following important (overlapping) approaches: integrated simulation support environments, object-oriented simulation, and knowledge-based simulation. Query driven simulation fits somewhere in the middle of these three approaches ([21], [23], [24], [31], [32]). Active KDL which is a functional object-oriented database system (FOODS) supports query driven simulation by providing access to integrated data, knowledge, and model bases. Active KDL is an enhancement of KDL ([33], [34], [35], [36], [37]). The three sublanguages of Active KDL provide strong support for simulation modeling and analysis: Complex simulations may be structured using the schema definition language (SDL); results may be retrieved using the query language (QL); and models (which may include active objects) may be implemented using the database programming language (DBPL).

Active KDL is designed to support the complex information needs of engineering databases and expert databases. The fact that Active KDL is built from solid theoretical foundations (i.e., object-oriented programming, functional programming, and hyper-semantic data modeling) allows Active KDL to meet the needs of demanding applications (e.g., simulation, model management, CAD/CAM, and intelligent database applications such as a university data/knowledge base capable of advising students).

A single user uniprocessor prototype of KDL is nearly complete. We are now in the process of designing a multi-user multiprocessor prototype of Active KDL. In this design, databases will consist of a large number of active objects. Active objects provide a nice abstraction for parallel and distributed object-oriented database systems. In Active KDL, each active object has its own thread of control, thereby allowing concurrent execution of active objects ([21], [23]). The use of threads (lightweight processes) facilitates high concurrency because of their low overhead. Active objects communicate with each other by sending messages. They are also able to interact with ordinary passive objects by invoking the...
passive objects' functions (attributes, heuristics, or methods).

2. Active Objects

In this paper, we examine performance issues related to the use of active objects in object-oriented database systems such as Active KDL. Because of the enormous complexity of such systems and the many possible design choices, the first step is to model the problem. Some of the important design alternatives are listed below:

1. Should all of the objects be active or should the database support both active and passive objects? The database would be more elegant if we only had active objects, but this might lead to too much overhead.

2. Should objects consist of a single version or should multiple versions be supported? Having multiple versions increases the amount of storage needed, but also helps facilitate complex engineering applications [7].

3. For highly concurrent systems, is better to use protocols that rely primarily on blocking or on aborting/undoing?

Our initial modeling study focuses on what we believe to be a good set of design choices. This study compares the performance of multiversion protocols for databases consisting of entirely active objects. Furthermore, we believe that the use of blocking should be kept to a minimum, so that aborting/undoing is used as the primary means of handling conflicts. Some support for this belief is given in [28]. Later studies that try other design choices must be performed to test these beliefs.

To achieve high performance, object-oriented database systems must be designed to exploit the concurrent capabilities of multiprocessors. Suitable architectures include tightly coupled multiprocessor, loosely coupled multiprocessors, and reliable high-speed LAN’s. Our interest is on parallelism and high performance and not on distribution and fault-tolerance. The architecture must support the assignment of active objects to a processor such that the sharing of the processor does not become a major bottleneck. Preferring aborting/undoing over blocking seems more appropriate for these architectures, since their main disadvantage is that they require more work to be done which is not so bad if there are plenty of processors around to do the work.

For the sake of consistency, the execution of active objects must be controlled. The active objects will periodically perform a sub-transaction and must be ready to handle requests from other active objects. Left uncontrolled, this concurrent behavior leads to inconsistency. Therefore protocols must be used to ensure that active objects do not violate certain consistency conditions. A strong consistency condition for the execution of a set of concurrent sub-transactions is serializability. An execution of concurrent sub-transactions is said to be serializable, if their effects are equivalent to some serial execution. Protocols that enforce serializability are called concurrency control protocols ([1], [4]). Closely related are the recovery protocols that ensure that a database can recover from non-catastrophic failures and aborts of sub-transactions ([5], [12], [17]). Also, for parallel and distributed simulation, causal events must be executed before their resultant events ([9], [18], [29]).

Interestingly enough, there are similarities between protocols for database concurrency control and protocols for simulation. One protocol having versions that work for both applications (database and simulation) is the Time Warp protocol ([18], [19]). A similar protocol, that has been found to have good performance [28] enforcing serializability in highly concurrent systems, is the Multiversion Timestamp Ordering protocol [38]. Since query driven simulation requires both types of protocols, we are performing preliminary studies to find suitable protocols for inclusion into a query driven simulation system/environment.

In this paper we analyze the performance of protocols that minimally use blocking. In particular, we compare the performance of the Multiversion Timestamp Ordering and Time Warp protocols. Additionally, a recovery protocol will need to be incorporated to fully ensure consistency. Protocols such as the Multiversion Timestamp Ordering and Time Warp protocols have the following advantages:

1. High concurrency level,
2. No need for deadlock detection, and
3. Highly efficient operation unless there is a conflict.

These types of protocols resolve conflicts by undoing operations or aborting sub-transactions within active objects. The fact that active objects are not blocked leads to higher effective concurrency levels. Thus, if aborts or undo operations do not occur with great frequency, these types of protocols may exhibit excellent performance. The analysis of performance is carried out by elements of analytic modeling (Markov models) and simulation modeling (SIMODULA* programs). Both throughput and delay time are studied via these models.

* SIMODULA is a simple yet powerful process-oriented simulation system written in Modula-2 ([25], [26], [27]).
3. Concurrency Control and Recovery

The performance of a database system, especially one that supports a multitude of concurrent users, is greatly affected by the protocols used by the system ([14], [15], [16], [28]). Indeed, in future systems where high concurrency and parallelism are essential, the design of efficient protocols will be critical. As a simple example, suppose that a system has $M$ transactions executing concurrently, each requiring 10 seconds of service to complete, of which 10% must be spent in mutual exclusion. On a system with sufficient parallelism, the time for a transaction to complete (response time) is $10(1 + 0.1(M - 1))$ seconds. For $M = 6$ the response time is 15 seconds, which is a tolerable increase in response time. However, for $M = 101$ the response time is 100 seconds, which is an intolerable (order of magnitude) increase.

Transactions are the basic units of work in database systems. The transactions perform a sequence of read and write operations on objects.$^\dagger$ They then finish by issuing either a commit (normal completion) or abort (abnormal termination) operation. A commit guarantees the permanence of the effects of a transaction, while an abort removes the partial effects of a transaction. Database protocols are used to constrain the sequencing of database operations submitted by the transactions. This is to ensure that the transactions do not violate certain correctness conditions.

The two primary correctness conditions to maintain in a database are serializability and recoverability. Serializability guarantees the consistency of data, by guaranteeing that if the individual transactions are correct then their combined effect will be correct ([30], [43]). Serializability is maintained by constraining the interleaving of operations so that their effect is equivalent to a serial execution of the transactions. Recoverability guarantees that after a failure condition (transaction failure or system crash) the database can be recovered to a consistent state at which point things can carry on as usual [17]. Recoverability is maintained by not allowing a transaction to commit until all the data it has read is committed.

In this paper we model transactions ([41], [42]) as Markov chains ([14], [15], [28]) where each state represents a database operation such as reading or writing. The models are used to analyze the performance of database systems. In particular, we study the effect that database protocols have on performance measures such as throughput and response time. In [28] these Markov models are derived from an intractable queuing network ([13], [22], [40]) by applying a mean substitution approximation. In [14] the Markov models are applied to four recovery protocols (realistic, optimistic, pessimistic, and paranoid). The most important result from this study was that the realistic and optimistic protocols performed significantly better than the pessimistic and paranoid protocols. Additionally, because of the problem of cascaded aborts, the optimistic protocol can suffer greater degradation in some circumstances.

Since any of the four recovery protocols could be used with the concurrency control protocols, we briefly review them. Since the realistic recovery protocol appears to be a good choice for a multiversion database, we will develop the model for it in detail. This development will show how transactions (or more specifically sub-transactions of active objects) can be modeled. Furthermore, because of the simplicity of the protocol and consequently the model for it, we are able to validate the simulation results with analytic results. For the two concurrency control protocols only simulation results were produced.

In this paper, we consider only two-stage transactions which do all their reading before writing. In [28] non two-stage transactions where reads and writes are intermixed are considered. Two-stage transactions appear to be very good from a performance standpoint, deadlock recovery is greatly simplified (indeed deadlocks cannot occur for several of the protocols), and the throughput for two-stage transactions is significantly better than the throughput for non two-stage transactions. Furthermore, any transaction can be turned into a two-stage transaction by a simple change to the execution of the write operations: The data to be written is held in a local work area until the transaction is complete, then all writes are output. Note that this requires that we also postpone scheduling activities (e.g., setting locks, timestamps, etc.) associated with writes.

4. A Model of Sub-Transaction Behavior

The concurrent behavior of multiversion object-oriented databases can be modeled as follows: The database consists of a set of $D$ active objects. At any time, $M$ of these active objects are requesting operations to be performed by other active objects. This is accomplished by sending messages between active objects. For the sake of consistency a sequence of interrelated operations are grouped together to form a sub-transaction. These $M$ active objects are said to be transacting. The other $D - M$ active objects sleep until the next operation request message comes in.

In the models, transacting active objects perform a sequence of sub-transactions. Furthermore, in this modeling study only read and write operations are considered.

$^\dagger$ Object-oriented database systems provide additional types of operations such as increment, which may facilitate better performance.
The flux (rate of flow) out of state \( j \), while the right hand side represents the flux into state \( j \). Hence, it is referred to as the balance equation, while the second equation is referred as the normalization equation. The system can then be viewed as \( M \) parallel Markov chains (the mean substitution approximation takes care of the interaction). Hence the number of sub-transactions at node \( j \) is simply

\[
N_j = \pi_j M
\]

### 4.2. SIMODULA Simulation Models

The active objects in the database are modeled as SIMODULA ([25], [26], [27]) processes. In the script procedure used for these processes, a transacting active object repeatedly performs sub-transactions until the simulation is over. Within a sub-transaction, \( k \) (= Objects) read/write operations are performed followed by a commit. This usual behavior is modified if the sub-transaction is aborted or partially undone. [A PreCommit operation is added for the Time Warp protocol to allow a sub-transaction to be reversed up until the last moment of its execution.] An abbreviated version of the ActiveObject script for the Time Warp protocol is given below. The script for the other protocols are very similar. The main differences occur in the logic of the database operations (Read, Write, PreCommit, Commit, Undo, and Abort). [Each of these operations WORK Uniform (0.0, Tmax, Stream) units of time where Tmax for PreCommits, Commits, Undos, and Aborts is a few times greater than that of Reads and Writes.]

```plaintext
PROCEDURE ActiveObject;
VAR
old: CARDINAL;
attr: ObjectAttributes;
arrivalTime, timestamp, delay: LONGREAL;
aborted: BOOLEAN;
BEGIN
Attributes (CurrentProcess (), attr);
WITH attr DO
    IF Transacting THEN
        /* Perform sequence of sub-transactions */
        WHILE Completions <= NumStop DO
            INC (SubTransaction);
            arrivalTime := Time ();
            aborted := TRUE;
            WHILE aborted DO
                aborted := FALSE;
                timestamp := Time ();
                /* Start/restart sub-transaction */
                WORK (0.1); /* Start time delay */
                OpNum := 1;
                LOOP
                    IF aborted THEN EXIT; END;
                    IF abort THEN EXIT; END;
                    IF Select Read/Write Operation *)
                    ICON (OpNum) <=
                    ReadProb * FLOAT (Objects)
                    aborted := Read (old, timestamp);
                    ELSE
                        aborted := Write (old, timestamp);
                    END (* IF *);
```
5. Recovery Protocol Models

In this section, we address the performance of various protocols that enforce the recoverability condition. As mentioned earlier, this condition states that a transaction cannot commit until all the data it has read is committed. Most of the protocols we consider also enforce a dual condition known as restorability [13] which prevents a transaction from aborting before any transaction that reads its data. Although it is not a critical correctness condition, its violation will require more complex aborting procedures. In summary, recoverability constrains the order in which commits can occur while restorability constrains the order in which aborts can occur.

In [14], we developed analytic models to predict the performance of four database recovery protocols described in a paper by Graham, Griffeth and Smith-Thomas [11]. The [11] paper establishes the correctness of the protocols and the preservation of serializability by the protocols. In general, a recovery scheduler has two tools that it can use to resolve conflicts, blocking and aborting¹. For example, if a later transaction attempts to access an object that is being used by an earlier transaction, then the recovery scheduler can either block the later transaction until the earlier one is finished with the object, or abort the later transaction forcing it to start over. It can also defer its decision to a later operation (e.g., commit) hoping the conflict will go away.

All but one of the protocols work by not allowing transactions to read uncommitted data, thereby guaranteeing recoverability. The exception allows such, giving potentially more concurrency, but risking difficulties such as cascaded aborts, and group commits or aborts. In order to simplify descriptions and facilitate analysis, we will use a locking abstraction. Thus, for all of the recovery protocols considered, the transactions acquire write locks on objects before writing the object. [We use locks in a generic sense to mean actual locks or something functionally equivalent.] The locks are then held until the transaction either commits or aborts. The protocols differ only in the semantics of the locks. The four recovery protocols can now be succinctly described as follows:

1. **Realistic** - transactions attempting to read locked objects are blocked;
2. **Optimistic** - transactions that have read locked objects that have as yet not been unlocked are blocked at the commit phase;
3. **Pessimistic** - transactions attempting to read or write locked objects are blocked;
4. **Paranoid** - transactions attempting to read locked objects are aborted.

The **Pessimistic Recovery protocol** is probably the most straightforward way to implement a recovery protocol. If an object has been written to by an uncommitted transaction, then no other transaction is allowed to access that object until the original transaction commits. Data that has been written by transactions that have yet to commit is termed dirty data to convey the idea that until the transaction certifies the data by committing it may or may not be good. Thus, recoverability is enforced by this protocol by not allowing transactions to read dirty data.

If we look more closely at the pessimistic protocol we will note that we are blocking writes which has nothing directly to do with reading dirty data. We must not overwrite a object that has pending reads, since this would destroy serializability. However, if we allow multiple versions of the object to exist and keep track of which version a particular read is to see, then we can allow the write to proceed unblocked. This is the idea behind the **Realistic Recovery protocol** which trades off more bookkeeping and space for more concurrency. The protocol must keep all uncommitted versions and a few of the latest committed versions of the object in the database (in the list of versions). [Of course in a multiversion database, one would keep object versions around longer than this.]

A convenient way to examine the behavior of the Realistic Recovery protocol is to look at an Op-List (list of operations) for an object (this information is easily extracted from the list of versions). Basically, committed writes must be left on the Op-List until they are immediately followed by another committed write. Reads are removed once they have executed. Hence, an Op-List for object O₂₀ might look something like the following, where committed writes are boldfaced.

```
DESTROY;
HANDlEMess ages;
WHORK(1.0);
IF OpNum = Objects THEN PreCommit; END;
INC(OpNum);
IF OpNum = Objects + 1 THEN EXIT; END;
END (* LOOP *);
IF * aborted THEN Commit; END;
END (* WHILE *);
delay := Time() - arrivalTime;
Tally [0, delay];
INC (Completion);
END (* WHILE *);
DEC (NumExe);
END (* IF *);
END (* WITH *);

/* Handle messages until simulation is over */
WHILE NumExe > 0 DO
HandleMessages;
IF NumExe > 0 THEN SUSPEND;
ELSE WORK (1.0); END;
END (* WHILE *);
DESTROY;
END ActiveObject;
```
While the above protocols rely on blocking (save in the event of deadlock), the protocols below make extensive use of aborting. Another way to allow writes to proceed and prevent the reading of dirty data is just to abort a transaction if it attempts to read dirty data. This is done by the Paranoid Recovery protocol. This protocol has the advantage of being simple to implement (e.g., complex data structures need not be maintained, and deadlocks need not be checked). We should note that if transactions are two-stage, then aborts for this protocol are cheap as there are no writes to be undone. Still, it seems that choosing to abort every time there is a conflict is a bit excessive.

A quite different approach is taken by the Optimistic Recovery protocol. It allows reads to go ahead and look at dirty data, thus allowing a transaction to possibly perform operations using erroneous data. However, before this transaction can commit, the data must be cleaned up (committed). Thus, this transaction must wait for all the transactions on which it depends to commit before it can commit. If any one of those transactions aborts, then it must abort, and any transaction depending on it must also abort. Thus, just like an optimist, when things go well this protocol does well, but it has the potentially very bad downside of cascaded aborts.

For each protocol, a Markov model can be derived using the following steps.

**Modeling Procedure**

1. Determine the state transitions for each state of a transaction;
2. Solve the balance equations in terms of:
   - \( k \) - the number of read and write operations in a transaction,
   - \( r \) - the number of read operations in a transaction,
   - \( c \) - the expected time required to commit a transaction which has done all its read and write operations,
   - \( p \) - the probability that an operation conflicts with an uncommitted write,
   - \( w \) - the expected time spent in blocked state each time a transaction is blocked;
3. Determine \( p \) & \( w \) - these constants depend on the properties of the system as a whole;
4. Compute the dependent variables: throughput, delay and number executing.

The key variables in analyzing these models are \( p \) (probability of conflict) and \( w \) (mean waiting time).

These variables can be expressed as complex functions of the state of an intractable queueing network [28]. Using a mean substitution approximation, which was shown to be good by simulation, we are able to solve for \( p \) and \( w \) in terms of the \( \pi_i \)'s. We have assumed that the access pattern to the objects is uniform, implying that the probability that an access results in a conflict is simply the ratio of the number of locks held to the total number of objects, \( L/D \). The total number of locks held by all the transactions is computed by tallying \( l_i \) locks for each transaction in state \( j \).

\[
L = \sum_j l_j N_j
\]

The mean waiting time can be computed by a similar formula. Let \( W_j \) be the expected time for a transaction to complete from state \( j \). The probability that a blocked transaction is waiting for a lock held by a transaction in state \( j \) is simply \( l_j N_j / L \). Hence, the mean waiting time is given by the weighted sum.

\[
w = L^{-1} \sum_j l_j N_j W_j
\]

**5.1. Realistic Recovery Protocol**

For the reasons mentioned earlier, we now present the model for the realistic recovery protocol. The behavior of a typical sub-transaction following this protocol is described by the Markov chain represented by the state transition diagram in Figure 1. The states in the diagram have the following meanings: State 0 is the start state, states 1 to \( r \) are read states, states \( r+1 \) to \( k \) are write states, state \( k+1 \) is the commit state, while states \( k+2 \) to \( k+r+1 \) are blocked states.

The transitions between states represent requests. For example, a transition out of states 0 to \( r-1 \) represents a read request; with probability \( q = 1 - p \) the request will be granted, so the next read state will be entered; and with probability \( p \) the request will be denied, so that a blocked state will be entered. The basic rate at which operations are requested (state transitions) is given to be 1. It could as well have been given by \( \mu \) (just multiply all of the rates by \( \mu \), which would then factor out).

To find the equilibrium solution, we apply the balance equations as follows. The transition rate into a read state is \( q \) from the previous state and \( 1/w \) from the preceding blocked read state, while the rate out is 1. The rates into and out of a write state are simply equal to 1. The rate into the start state is \( 1/c \) from the commit state, while the rate out is 1. Finally, the rate into a blocked read state is \( p \) from a read (or the start) state, while the output rate is \( 1/w \).
Figure 1: Realistic Recovery Protocol.

\[
\begin{align*}
\text{Input} & = \text{Output} \\
q \pi_{j-1} + \frac{1}{\omega} \pi_{k+1+j} & = \pi_j, \quad j = 1 \cdots r \\
\pi_{j-1} & = \pi_j, \quad j = r+1 \cdots k \\
\frac{1}{c} \pi_{k+1} & = \pi_0 \\
p \pi_{j-1} & = \frac{1}{\omega} \pi_{k+1+j}, \quad j = 1 \cdots r
\end{align*}
\]

Hence solving for \( \pi_j \) we get

\[
\begin{align*}
\pi_j & = \pi_0, \quad j = 0 \cdots k \\
& = \frac{1}{\omega} \pi_0, \quad j = k+1 \cdots k+r+1 \\
\sum_j \pi_j & = \pi_0 (k+1+c+rpw) = \pi_0 (t+rpw) = 1
\end{align*}
\]

where \( t = k+1+c \).

We have now solved for the state probabilities, the \( \pi_j \)'s, in terms of several constants, where \( k, c, \) and \( r \) are input parameters, and \( p \) and \( \omega \) can be derived from the other parameters including \( \lambda = M/D \), the congestion ratio. The probability of conflict \( p \) equals the ratio of the mean number of locks held \( L \) to the total number of objects \( D \). Applying the formula \( \lambda = L/D = \sum_j l_j N_j \) where \( l_{r+j} = j \) for \( j = 1 \cdots s \) (write), \( l_{k+1} = s \) (commit), and \( l_j = 0 \) otherwise, and recalling that \( N_j = \pi_j M \), we get

\[
p = \frac{L}{D} = \frac{\sum_j l_j \pi_j}{\lambda}
\]

\[
= \lambda \left[ \sum_j \pi_{r+j} + s \pi_{k+1} \right] = \lambda \pi_0 \left[ \frac{s}{2} + \frac{s(s+1)}{2} \right]
\]

\[
= \lambda \pi_0 (s+1) = \lambda \pi_0 (s+1+2c)/2
\]

We are now left with the problem of determining the mean time \( w \) that a sub-transaction is in a blocked state. A sub-transaction can wait on only one other sub-transaction which must be in either its write or commit phase. The mean times for a sub-transaction to complete from one of these states are given by the following two formulae (the first for writes and the second for commits).

\[
W_{r+j} = c + (s-j) + 2/3 \\
W_{k+1} = 2c / 3
\]

Note that the mean time it takes a sub-transaction to leave its current state can be determined using residual-life theory [39]. In this theory, if \( Y \) is the holding time in a state, then the mean residual-life is \( z E(Y) \) where \( z = E(Y^2)/2E^2(Y) \). Since in our simulations we used the uniform distribution on \([0,b]\), we will use the \( z \) for this distribution which is \( z = 2/3 \). Therefore, applying the formula for \( w \) we get

\[
w = \sum_j l_j W_j / \sum_j l_j \pi_j
\]

\[
= \frac{\sum_j (c + s + 2j - j) + sc (2c/3)}{\sum_j j + sc}
\]

\[
= \frac{(s+2/3) s(s+1)/2 + sc (2c/3) - s(s+1)(2s+1)/6}{s(s+1)/2 + sc}
\]

\[
= \frac{(s+1+3c)(s+1) + 4c^2}{3(s+1+2c)}
\]

which can be directly computed. We may now find \( p \) by solving the following quadratic equation

\[
rvp^2 + \lambda v - \lambda v ((s+1)/2 + c) = 0
\]

so that choosing the correct root, the one in \((0, 1)\), we get
\[ p = \frac{-u + \sqrt{u^2 + 4wV}}{2w} \]

For this protocol the equations turn out to be particularly simple. In general, we will have two simultaneous nonlinear equations for the unknowns \( p \) and \( w \). There are several good numerical algorithms (e.g., Newton's method) for solving such equations.

Having determined the values for \( p \) and \( w \), we can now compute the performance measures of primary interest. The probability a sub-transaction is in state 0 is

\[ \pi_0 = (t + rwp)^{-1} \]

The mean number of sub-transactions executing may be computed by summing over all nonblocked nodes.

\[ N_0 = \sum_{i=0}^{\infty} N_f = 1,000M \]

The throughput in committed sub-transactions per unit time is simply the mean rate at which sub-transactions leave node \( k + 1 \). Since in node \( k + 1 \) we have \( N_{k+1} \) sub-transactions each completing at rate \( \mu_{k+1} = 1/c \) we get

\[ \theta = \sum_{k=1}^{\infty} \mu_{k+1} = \pi_0 M \]

Finally, the mean total sojourn time \( T \) (from start to completion) for sub-transactions can be found by using Little's law

\[ T = M/\theta \]

6. Concurrency Control Protocol Models

Both the Multiversion Timestamp Ordering and Time Warp protocols are based upon the use of timestamps to order the execution of sub-transactions. They both attempt to achieve high performance by exploiting the fact that there are multiple versions of objects.

6.1. Multiversion Timestamp Ordering Protocol

Under the Multiversion Timestamp Ordering protocol, each sub-transaction is timestamped upon initiation. In the simulation this is simply done by calling the Time () function. In a parallel or distributed database, the generation of good unique timestamps is a bit more challenging (see [6] for techniques to do so). As discussed previously, the versions are also timestamped with the timestamp of the sub-transaction writing it. Additionally, it is important to keep track of the latest reader of each version. Hence, the following information needs to be maintained for the successful operation of the protocol.

- \( ts(T_i) \) = timestamp of sub-transaction \( T_i \)
- \( ws(v_k) \) = write-stamp of version \( v_k \)
- \( rs(v_k) \) = read-stamp of version \( v_k \)

The Multiversion Timestamp Ordering protocol supports high concurrency since Reads can be executed in a relatively unrestricted way. Except for the concern for recoverability, there are no restrictions at all (simply the appropriate version is chosen and read). Writes, however, may cause a sub-transaction to be aborted. In particular, if sub-transaction \( T_i \) attempts to write a version of object \( O_j \) that interferes with an existing information transfer, then this write cannot be allowed to proceed. This occurs under the following conditions

\[ ws(v_k) < ts(T_i) < rs(v_k) \]

where \( v_k \) is the version that immediately precedes the version \( T_i \) is attempting to write. In other words, \( T_i \) is attempting to insert a new version immediately after \( v_k \), but since the read-stamp on \( v_k \) is greater than the timestamp of \( T_i \), this cannot be allowed. The problem is that a sub-transaction reading version \( v_k \) should have really read the version that sub-transaction \( T_i \) is attempting to write. For example, consider the following schedule.

\[ W_2(O_10) R_4(O_10) W_5(O_20) R_3(O_20) R_5(O_20) \]

If sub-transaction 8 attempts to write a version of \( O_20 \), it must be aborted since sub-transaction 9 has already read the version written by sub-transaction 5.

Since writes may be aborted, some type of recovery protocol is necessary to maintain consistency. One that appears particularly well suited is the realistic recovery protocol. As mentioned before, it will block reads until the versions they are attempting to read are committed. This will introduce some extra delay into the system, but the amount of blocking will be rather small in comparison to the pessimistic recovery protocol or the commonly used two-phase locking concurrency control protocol [28]. Furthermore, so long as the transactions are two-phase no deadlocks can occur.

6.2. Time Warp Protocol

The Time Warp protocol has been studied extensively for distributed simulation. It allows active objects (processes) to advance their state forward as rapidly as possible, until a message (from some other active object) is received whose receive time is in the past. When this happens, the active object must be rolled back to this time so that the event specified in the message can be executed. This same Time Warp protocol can be adapted slightly for use as a database concurrency control protocol ([18], [19]).

In the schedule given above, sub-transaction 7 also read the version written by sub-transaction 5. Unlike the read by sub-transaction 9, this presents no problem for sub-transaction 8 in its write attempt. To maintain serializability, it suffices to have sub-transaction 9 change its read from version 5 to version 8. Hence, we need to be able to partially undo a sub-transaction. The simplest approach is to back up sub-transaction 9 to the point of
the read of $O_{20}$ (i.e., the read in question) and then begin redoing the sub-transaction. A more detailed analysis shows that the following would suffice:

1. Undo and then redo reads that are \textit{conditionally dependent} on the read in question. A read of object $O_j$ is conditionally dependent, if there is conditional logic in the sub-transaction that uses the value of the read in question to decide whether or not to read object $O_j$.

2. Undo and then redo writes that are either \textit{conditionally} or \textit{value dependent} on the read in question. A write is value dependent, if the value obtained by the read in question is used in the write's calculation.

Choosing to undo rather than abort as is done with the Multiversion Timestamp Ordering protocol, has appeal since aborting is a more drastic operation. However, when the writes of a sub-transaction are undone, they will cause other sub-transactions to be partially undone leading to cascaded undo's. This would suggest that as the rate of conflict gets high, the performance of this protocol would degrade rapidly. A further difficulty is that some of the sub-transactions that need to be undone cannot, since they have already committed. The are two remedies to this problem. The first remedy assumes that transactions can be reversed through the use of anti-transactions. The second is to use a hybrid protocol. Use Time Warp so long as none of the reads in question belong to committed transactions and use Multiversion Timestamp Ordering otherwise.

In our modeling study, we consider two scenarios for the Time Warp protocol. Under scenario 1, we assume that all operations that come after the read in question are dependent on it, and hence need to be undone. Under scenario 2, we assume no conditional dependencies exist. Furthermore, we assume for the sake of simplicity (both in modeling and in actual protocol implementations) that all of the writes are value dependent. Thus, sub-transactions will only need to be backed up to the beginning of their write phase. More optimistic scenarios are also possible. Since scenario 2 represents the middle ground, in this paper we present the results from this scenario.

7. Results and Conclusions

With $D$ fixed at 500 active objects and the object access time distributed uniformly from 0 to 20 milliseconds, two experiments were conducted. In the first experiment reads and writes were equally likely (ReadProb = 0.50, $k = 12$, $r = 6$), while in the second experiment reads were more common (ReadProb = 0.75, $k = 12$, $r = 9$). The simulation results for the first experiment are summarized in Figure 2. This figure displays the throughput in committed sub-transactions per second (tps). In particular, the throughput is plotted versus the level of concurrency ($M$). $M$ ranges from 10 (low concurrency) to 70 (high concurrency).

In previous studies [28], the Multiversion Timestamp Ordering (mvts) protocol was found to perform significantly better than the standard two-phase locking protocol, if the system had the resources necessary for high concurrency. From the results given in this paper, it appears that the Time Warp (tw) protocol's overall performance is even better. For the equiprobable case, the peak throughput improves from 94 tps for the mvts protocol to 122 tps for the tw protocol.

Similarly, for the case in which reads are more common, the peak throughput improves from 125 tps for the mvts protocol to 144 tps for the tw protocol. [Note, the realistic recovery protocol exhibits much greater throughput (peaking above 300 tps) which is expected since recoverability is a much easier condition to enforce than serializability.]

Although the number of undo's performed by the tw protocol is greater than the sum of the aborts and blocks (blocks are caused by the embedded realistic recovery protocol) performed by the mvts protocol, tw exhibits better performance since undoing is a less costly operation. On the negative side, however, when the concurrency level is very high, the rate of undoing begins to explode leading to a sharp downturn in throughput.

Much work remains to be done in the future. The most important next step is to deal with the situation in which the use of anti-transactions is not feasible. We plan to study the performance of a hybrid of the mvts and tw protocols to handle this situation. Then the three protocols, mvts, tw, and hybrid, need to be analyzed under a wider range of scenarios than was done in this preliminary study. [Although it may introduce significant blocking, for completeness we will also analyze the version of tw that avoids the necessity of anti-transactions by delaying the commitment of transactions [19].]

As mentioned in the introduction, this study focused on protocols that we believed best suited our needs. Clearly, alternative protocols (e.g., two-phase locking, multiversion locking, K-R optimism [6], [8]) must also be analyzed. Finally, the models need to be refined in order to study the performance that can be gained by exploiting the object-oriented-ness of the database (e.g., operations such as increment [10]).
Figure 2: Throughput vs. M.
8. References


