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 papers 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 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 [Mill89, Mill90a, Mill90b, Pott90, Mill91, Koch91]. Its fundamental tenant is that simulationists or even naive users should see a system based upon query driven simulation as a sophisticated information system. System based upon query driven simulation are able to store information about or to generate information about the behavior of systems that users wish to study.

Object-oriented database systems provide data and behavioral structuring mechanisms powerful enough to support demanding applications like query driven simulation. This papers 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 system.

We are currently in the process of designing and implementing a prototype of such a system. The foundation of this prototype is Active KDL (Knowledge/Data Language) [Mill90a, Pott90, Mill91, Pott91, Koch91]. 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 [Pott86, Pott87, Pott88a, Pott88b, Pott89]. 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 [Kess89, Kesk90, Ucar90]. 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 [Mill91, Koch91]. 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 [***].
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 [Mill86]. 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*. Closely related are the *recovery protocols* that ensure that a database can recover from non-catastrophic failures and aborts of sub-transactions. Also, for parallel and distributed simulation causal events must be executed before their resultant events [Chan79, Jeff85, Misra86].

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* [Jeff85]. A similar protocol, that has been found to have good performance [Mill86] enforcing serializability in highly concurrent systems, is the *multi-version timestamp protocol* [Reed78]. 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 multi-version 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.

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 [Grif84a, Grif84b, Grif85, Mill86]. 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 $10(1 + .1(M - 1))$ seconds. For $M = 6$ the response time is 15 seconds, a tolerable increase. However for $M = 101$, the response time is 100 seconds, an 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‡. 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 [Papa77, Ullm82]. 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 [Hadz83]. 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 [Tay84, Tay85] as Markov chains [Grif84a, Grif85, Mill86] 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 [Mill86] these Markov models are derived from an intractable queueing network by applying a

† SIMODULA is a simple yet powerful process-oriented simulation system written in Modula-2 [Mill88, Mill89, Mill90b].

‡ Object-oriented database systems provide additional types of operations such as increment, which may facilitate better performance.

mean substitution approximation. In [Grif85] 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 [Mill86] 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. [Note, some operations such as increment allow very general interleavings and therefore contribute to high concurrency [Allc83].] A write operation will simply write a new version of an object. The different versions are distinguished via the unique timestamps of the writing sub-transactions ($ts(T_i)$). Formally, a write by sub-transaction T_i to object O_j has the following effect:

$$v_k = W_i(O_j)$$

$$O_j = O_j \cup \{ v_k \}$$

The write $W_i(O_j)$ produces a new version v_k with write-stamp $ws(v_k) = ts(T_i)$. This version is inserted into the ordered list O_j according to its write-stamp. Sub-transactions will read the most recent version of an object. Specifically, the most recent from its time frame, versions from the future may not be read. Consequently, when sub-transaction T_i reads from object O_j the version with the largest write-stamp not exceeding the timestamp of T_i is returned.

$$R_i(O_j) = \max_{ws} \{ v_k \mid v_k \in O_j \text{ and } ws(v_k) \le ts(T_i) \}$$

Each sub-transaction goes through a sequence of J stages each corresponding to an operation (i.e., after completing an operation a sub-transaction goes to the next stage to perform its next operation).

4.1. Analytic Markov Models

From an analytic modeling point of view, we can model each of the stages as a node in a queueing network. Therefore, the system will be composed of M concurrent transacting active objects traversing J nodes of the queueing network, reading and writing versions of D active objects. In [Mill86] we developed the queueing network models in detail. After applying a mean substitution approximation we were able to derive simple continuous-time Markov chain models, where $X(t)$ represents the state of a sub-transaction at time t. [In this derivation we showed that any service time distribution will work; it does not have to be exponential.]

We are interested in the steady-state behavior (equilibrium distribution -- where $X(t) \rightarrow X$ in distribution) of such chains, which may be determined from the following set of linear equations [Klei75, Ross83]

$$\pi_j \sum_{k \ne j} q_{jk} = \sum_{k \ne j} \pi_k q_{kj} \qquad j = 0 \cdots J-1$$

$$\sum_j \pi_j = 1$$

where q_{jk} is the transition rate from state j to state k, and

$$\pi_j = P\{X = j\}$$

is the equilibrium probability of a sub-transaction being in state j. The left hand side of the first equation represents 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 [Mill88, Mill89, Mill90b] 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/writes 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.]

```
PROCEDURE ActiveObject;
     (*********************************************************************
     Script for a typical ActiveObject.
     *********************************************************************)
VAR
     oid: CARDINAL;
     attr: ObjectAttributes;
     arrivalTime, timeStamp, delay: LONGREAL;
     aborted: BOOLEAN;
BEGIN
     Attributes (CurrentProcess (), attr);
     WITH attr^ DO
          IF Transacting THEN

               (* Carry out a 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;
                              INC (Requests);
                              oid := Randi (1, NumObjects, Stream);

                              (* Select Read/Write Operation *)
                              IF FLOAT (OpNum) <= ReadProb * FLOAT (Objects)
                                   aborted := Read (oid, timeStamp);
```

```
                                        ELSE
                                             aborted := Write (oid, timeStamp);
                                        END (* IF *);

                                        IF ˜ aborted AND (Random (Stream) < SAProb) THEN
                                             aborted := TRUE;
                                             INC (SAborts);      (* Spontaneous abort *)
                                             mvts.Abort (CurrentProcess ());
                                        END (* IF *);
                                        HandleMessages;
                                        WORK (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 (Completions);
                         END (* WHILE *);
                         DEC (NumExe);
                    END (* IF *);
               END (* WITH *);

               (* Handle messages until the simulation is over *)
               WHILE NumExe > 0 DO
                    HandleMessages;
                    IF NumExe > 0 THEN SUSPEND; ELSE WORK (1.0); END;
               END (* WHILE *);
               DESTROY;
          END  ActiveObject;
```

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* [Grif86], 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 [Grif85], we developed analytic models to predict the performance of four database recovery protocols described in a paper by Graham, Griffeth and Smith-Thomas [Grah84]. The [Grah84] 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 *aborting*†. 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 versions around longer than this.] A convenient way to examine the behavior of this 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

† Additionally, some protocols *undo* operations.

followed by another committed write. Reads are removed once they have executed. Hence an Op-List for object O_{20} might look something like the following, where committed writes are boldfaced.

$$\mathbf{W}_5 W_4 R_7 R_6 \mathbf{W}_8 W_9 R_{12} R_{11} R_{13}$$

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. [One transaction depends on another if it has read data from that transaction.] 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 [Mill 86]. 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 π_j'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_j 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 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 μ (just multiply all of the rates by μ, which would then factor out).

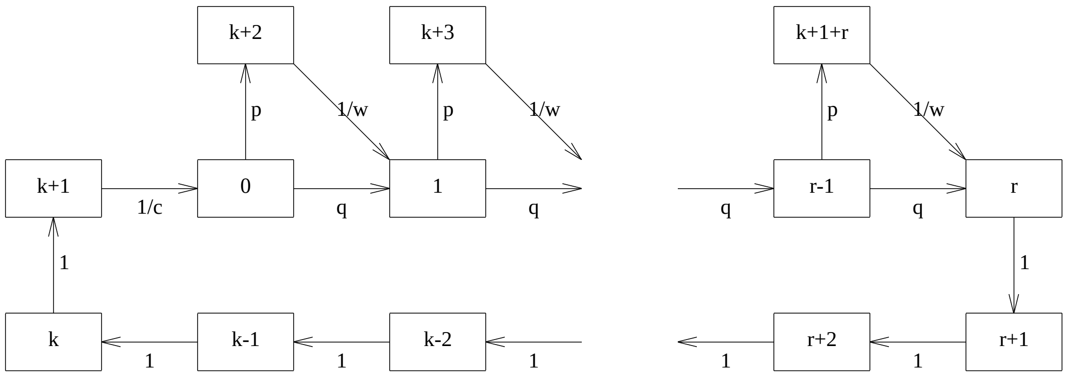

Figure 1. Realistic Recovery Protocol

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 $1/w$.

Input	=	**Output**	
$q\pi_{j-1} + 1/w\,\pi_{k+1+j}$	=	π_j	$j = 1 \cdots r$
π_{j-1}	=	π_j	$j = r+1 \cdots k$
$1/c\,\pi_{k+1}$	=	π_0	
$p\,\pi_{j-1}$	=	$1/w\,\pi_{k+1+j}$	$j = 1 \cdots r$

Hence solving for π_j we get

$$
\begin{aligned}
\pi_j &= \pi_0 && j = 0 \cdots k \\
&= c\,\pi_0 && j = k+1 \\
&= pw\,\pi_0 && j = k+2 \cdots k+r+1 \\
\sum_j \pi_j &= \pi_0(k + 1 + c + rpw) && = \pi_0(t + rpw) = 1
\end{aligned}
$$

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 w 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 $L = \sum_j l_j N_j$ 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

$$p = L/D = \lambda \sum_j l_j \pi_j$$

$$= \lambda [\sum_{j=1}^{s} j \pi_{r+j} + s \pi_{k+1]} = \lambda \pi_0 [\sum_{j=1}^{s} j + sc]$$

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

$$= \frac{\lambda s(s+1+2c)/2}{t + rpw}$$

We are now left with the problem of determining the mean time w that a sub-transaction waits 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 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* [Ross83]. 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

$$w = \sum_j l_j \pi_j W_j \Big/ \sum_j l_j \pi_j$$

$$= \frac{\sum_{j=1}^{s} j(c+s+2/3-j) + sc(2c/3)}{\sum_{j=1}^{s} j + sc}$$

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

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

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

$$rwp^2 + tp - \lambda s((s+1)/2 + c) = 0$$

so that choosing the correct root, the one in (0, 1), we get

$$p = \frac{-t + [t^2 + 4rws\lambda((s+1)/2 + c]^{1/2}}{2rw}$$

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 non-blocked nodes.

$$N_E = \sum_{j=0}^{k+1} N_j = t\pi_0 M$$

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

$$\theta = N_{k+1}\mu_{k+1} = \pi_0 M$$

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 timewarp protocols are based upon 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 [***] 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.

$$W_2(O_{10})\,R_4(O_{10})\,W_5(O_{20})\,R_7(O_{20})\,R_9(O_{20})$$

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 [Mill86]. Furthermore, so long as the transactions are two-phase no deadlocks can occur.

6.2. Timewarp Protocol

The timewarp 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 timewarp protocol can be adapted slightly for use as a database concurrency control protocol [Jeff85].

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 *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 conditionally or *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 timewarp 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.

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 = 0.05, $k = 12$, $r = 6$), while in the second experiment reads were more common (ReadProb = 0.75, $k = 12$, $r = 9$). The simulation results are presented in the appendix. For several levels of concurrency (M), nine performance measures are computed. For each entry, the top number is the grand mean, while the bottom number is the standard deviation of the batch means (5 batches each of size 100 with the first batch thrown out because of its transient nature). The most important performance measure is Throughput in committed sub-transactions per second (tps). Next in importance are the counts (per 100 commits) of responses to conflicts (Aborts, Blocks, and Anti-transactions).

In previous studies [Mill86], the multiversion timestamp ordering (mvtso) 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 timewarp (tw) protocol's overall performance is even better. For the equiprobable case, the peak throughput improves from 94 tps for the mvtso 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 mvtso protocol to 144 tps for the tw protocol. [Note, the realistic recovery protocol exhibits much greater throughput 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 (caused by the embedded realistic recovery protocol) performed by the mvtso 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 aborting 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 the situation in which the use of anti-transactions is not feasible. We plan to study the performance of a hybrid of the mvtso and tw protocols to handle this situation. Then the three protocols, mvtso, tw, and hybrid need to be analyzed under a wider range of scenarios than was done in this preliminary study. As mentioned in the introduction, this study focussed on protocols that we believed best suited our needs. Clearly, alternative protocols (e.g., two-phase locking, multiversion locking, K-R optimism [Cell88]) 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-orientedness of the database (e.g., operations such as increment).

8. REFERENCES

[Agra84] R. Agrawal, M. Carey and M. Livny, *Models for Studying Concurrency Control Performance: Alternatives and Implications*, Technical Report 567, University of Wisconsin (Dec 1984).

[Agra85] R. Agrawal and D. DeWitt, *Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation*, Transactions on Database Systems, 10, 4 (Dec 1985) 529-564.

[Allc83] J. Allchin, An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, Georgia Tech (Sept 1983).

[Bask75] F. Baskett, K. Chandy, R. Muntz and F. Palacios, *Open, Closed, and Mixed Networks of Queues with Different Classes of Customers*, Journal of the ACM 22, 2 (April 1975) 248-260.

[Bern81] P. Bernstein and N. Goodman, *Concurrency Control in Distributed Database Systems*, ACM Computing Surveys 13, 2 (June 1981) 185-222.

[Bern83] P. Bernstein, N. Goodman and V. Hadzilacos, *Recovery Algorithms for Database Systems*, Technical

Report-10-83, Harvard University (March 1983).

[Care83] M. Carey, Modeling and Evaluation of Database Concurrency Control Algorithms, Ph.D., Thesis, UCB (Sept 1983).

[Cell88] W. Cellary, E. Gelenbe and T. Morzy, **Concurrency Control in Distributed Database Systems**, North-Holland, Amsterdam (1988).

[Chan79] K. Chandy and J. Misra, *Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, SE-5, 5 (May 1979) 440-452.

[Ches83] A. Chesnais, E. Gelenbe and I. Mitrani, *On the Modeling of Parallel Access to Shared Data*, Communications of the ACM 26, 3 (March 1983) 196-202.

[Eswa76] K. Eswaran, J. Gray, R. Lorie and I. Traiger, *The Notions of Consistency and predicate Locks in a Database System*, Communications of the ACM 19, 11 (Nov 1976).

[Gall82] B. Galler, Concurrency Control Performance Issues, Ph.D. Thesis, University of Toronto (Sept 1982).

[Gall83] B. Galler and L. Bos, *A Model of transaction Blocking in Databases*, Performance Evaluation 3 (1983) 95-122.

[Garc79] H. Garcia-Molina, Performance of Update Algorithms for Replicated Data in a Distributed Database, Ph.D. Thesis, Stanford University (June 1979).

[Garc83] H. Garcia-Molina, *Using semantic Knowledge for Transaction Processing in a Distributed System*, Transactions on Database Systems 8, 2 (June 1983) 186-213.

[Grah84] M.H. Graham, N.D. Griffeth and B. Smith-Thomas, *Reliable Scheduling of Transactions on Unreliable Systems*, Proceedings of the 1984 Conference on Principles of Database System, Waterloo, Ont. (1984).

[Gray78] J. Gray, *Notes on Database Operating Systems*, **Operating Systems: An Advanced Course**, R. Bayer, et al. (Eds.), Springer-Verlag, Inc., N.Y. (1978) 393-481.

[Gray81] J. Gray, et al., *The Recovery Manager of the System R Database Manager*, ACM Computing Surveys 13, 2 (June 1981) 223-242.

[Grif86] N.D. Griffeth, *Reliability Issues in Data Engineering*, Proceedings of the Second International Conference on Data Engineering, (1986).

[Grif85] N.D. Griffeth and J.A. Miller, *Performance Modeling of Database Recovery Protocols*, IEEE Transactions on Software Engineering SE-11, 6 (June 1985) 564-572.

[Grif84a] N.D. Griffeth and J.A. Miller, *Performance Modeling of Database Recovery Protocols*, Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems, (Oct 1984) 75-83.

[Grif84b] N.D. Griffeth and M. Morsi, *Simulation of Concurrency Control and Recovery Protocols for Distributed Database Systems*, Proceedings of the 1984 IEEE InfoCon, San Francisco, CA (April 1984) 200-209.

[Hadz83] V. Hadzilacos, *An Operational Model for Database System Reliability*, Proceedings of the 1983 Conference on Principles of Distributed Computing, (1983) 244-257.

[Haer83] T. Haerder and A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys 15, 4 (Dec 1983).

[Kent84] J. Kent and H. Garcia-Molina, *Performance Evaluation of Crash Recovery Mechanisms*, Technical Report

329, Princeton University (Nov 1984).

[Kies83] W. Kiessling and G. Landherr, *A Quantitative Comparison of Lock Protocols for Centralized Databases*, Proceedings of the 9th Int. Conference on Very Large Data Bases (Nov 1983).

[Klei75] L. Kleinrock, **Queuing Systems, Vol I: Theory**, John Wiley & Sons, Inc., N.Y. (1975).

[Klei76] L. Kleinrock, **Queuing Systems, Vol II: Computer Applications**, John Wiley & Sons, Inc., N.Y. (1976).

[Koch91] K.J. Kochut, J.A. Miller and W.D. Potter, *Design of a CLOS Version of Active KDL: A Knowledge/Data Base System Capable of Query Driven Simulation*, Proceedings of the 1991 AI and Simulation International Conference (to appear), New Orleans, LA (April 1991).

[Kung81] H. Kung and J. Robinson, *On Optimistic Methods for Concurrency Control*, Transactions on Database Systems 6, 2 (June 1981) 213-226.

[Lang82] A. Langer and A. Shum, *The Distribution of Granule Accesses Made by Database Transactions*, Communications of the ACM 25, 11 (Nov 1982) 831-832.

[Lazo84] E. Lazowska, J. Zahorjan, G. Graham and K. Sevcik, **Quantitative System Performance -- Computer System Analysis Using Queueing Network Models**, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1984).

[Lin83] W. Lin and J. Nolte, *Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking*, Proceedings of the 9th Int. Conference on Very Large Data Bases (Nov 1983).

[Mill91] J.A. Miller, K.J. Kochut, W.D. Potter, E. Ucar and A.A. Keskin, *Query Driven Simulation In Active KDL: A Functional Object-Oriented Database System (FOODS)*, International Journal in Computer Simulation (to appear), (1991).

[Mill90a] J.A. Miller, W.D. Potter, K.J. Kochut and O.R. Weyrich, Jr., *Model Instantiation for Query Driven Simulation In Active KDL*, Proceedings of the 23rd Annual Simulation Symposium, Nashville, TN (April 1990).

[Mill90b] J.A. Miller, O.R. Weyrich, Jr., W.D. Potter and V. Kessler *The SIMODULA/OBJECTR Query Driven Simulation Support Environment*, in **Progress in Simulation**, J. Leonard and G. Zobrist (Eds.) (1990).

[Mill89] J.A. Miller and O.R. Weyrich, Jr., *Query Driven Simulation Using SIMODULA*, Proceedings of the 22nd Annual Simulation Symposium, Tampa, FL (March 1989).

[Mill88] J.A. Miller, O.R. Weyrich, Jr., and D. Suen, *A Software Engineering Oriented Comparison of Simulation Languages*, Proceedings of the 1988 Eastern Simulation Conferences: Tools for the Simulationists, Orlando, FL (April 1988).

[Mill86] J.A. Miller, Markovian Analysis and Optimization of Database Recovery Protocols, Ph. D. Thesis, Georgia Tech (Aug 1986).

[Misr86] J. Misra, *Distributed Discrete-Event Simulation*, ACM Computing Surveys, 18, 1 (March 1986) 39-65.

[Papa77] C. Papadimitriou, P. Bernstein and J. Rothnie, *Computational Problems Related to Database Concurrency Control*, Proceedings of the Conference on Theoretical Computer Science, Waterloo, Ont. (1977).

[Pein83] P. Peinl and A. Reuter, *Empirical Comparison of Database Concurrency Control Schemes*, Proceedings of the 9th Int. Conference on Very Large Data Bases (Nov. 1983).

[Poti80] D. Potier and P. Leblanc, *Analysis of Locking Policies in Database Management Systems*, Communications of the ACM 23, 10 (Oct 1980) 584-593.

[Pott91] W.D. Potter, K.J. Kochut, J.A. Miller, A.A. Keskin and E. Ucar, *Intensional and Extensional Effects of Data Modeling and Querying in KDL*, Submitted to the ACM-SIGMOD 1991 International Conference on Management of Data, Denver, CO (May 1991).

[Pott90] W.D. Potter, J.A. Miller and K.J. Kochut, *Supporting an Intelligent Simulation/Modeling Environment Using the Active KDL Object-Oriented Database Programming Language*, Proceedings of the 21st Annual Pittsburgh Conference on Simulation and Modeling, Pittsburgh, PA (May 1990).

[Pott89] W.D. Potter, R.P. Trueblood and C.M. Eastman, *Hyper-Semantic Data Modeling*, Data & Knowledge Engineering (1989).

[Pott88a] W.D. Potter, *KDL-ADVISOR: A Knowledge/Data Based System Written in KDL*, Proceedings of the 21st Annual Hawaii International Conference on System Science, Kailua-Kona, Hawaii (Jan 1988).

[Pott88b] W.D. Potter and R.P. Trueblood, *Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling*, IEEE Computer (June 1988).

[Pott87] W.D. Potter, R.P. Trueblood, C.M. Eastman, and M.M. Mathews, *KDL: A Hyper-Semantic Data Model Specification Language*, Proceeding of the 2nd International Symposium on Methodologies for Intelligent Systems, Colloquia Program, Charlotte, NC (Oct 1987).

[Pott86] W.D. Potter and L. Kerschberg, *A Unified Approach to Modeling Knowledge and Data*, Proceedings of the IFIP TC2 Conference on Knowledge and Data (DS-2), Algarve, Portugal, (Nov 1986).

[Reut84] A. Reuter, *Performance Analysis of Recovery Techniques*, Research Report, IBM Research Laboratory (June 1984).

[Ross83] S. Ross, **Stochastic Processes**, John Wiley & Sons, Inc., N.Y. (1983).

[Saue81] C. Sauer and K. Chandy, **Computer Systems Performance Modeling**, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1981).

[Ston86] M. Stonebraker and L. Rowe, *The Design of POSTGRES*, Proceedings of the ACM-SIGMOD 1986 International Conference on Management of Data, Washington, DC (June 1986).

[Tay84] Y. Tay, A Mean Value Performance Model for Locking in Databases, Ph.D. Thesis, Harvard University (Feb 1984).

[Tay85] Y. Tay, N. Goodman and R. Suri, *Locking Performance in Centralized Databases*, Transactions on Database Systems 10, 4 (Dec 1985) 415-462.

[Ullm82] J. Ullman, **Principles of Database Systems**, Computer Science Press, Rockville Md. (1982).

9. APPENDIX

Realistic Recovery Protocol Simulation (12, 0.50)

Concurrncy	Requests	Aborts	SAborts	Blocks
10	1215.600	0.000	0.000	13.000
510	35.970	0.000	0.000	2.098
Delay	NumExe	WBlock	Thruput	Conflict
159.334	9.678	39.629	62.209	0.011
2.891	0.065	2.630	2.176	0.002

Concurrncy	Requests	Aborts	SAborts	Blocks
20	1231.000	0.000	0.000	25.800
520	48.270	0.000	0.000	4.534
Delay	NumExe	WBlock	Thruput	Conflict
163.570	18.817	38.876	116.595	0.021
4.174	0.327	7.694	7.759	0.004

Concurrncy	Requests	Aborts	SAborts	Blocks
30	1243.400	0.000	0.000	40.800
530	85.692	0.000	0.000	6.554
Delay	NumExe	WBlock	Thruput	Conflict
168.449	27.101	37.350	164.832	0.033
3.581	0.315	3.156	23.497	0.006

Concurrncy	Requests	Aborts	SAborts	Blocks
40	1250.200	0.000	0.000	53.200
540	118.146	0.000	0.000	7.222
Delay	NumExe	WBlock	Thruput	Conflict
172.400	35.653	38.404	215.153	0.043
2.459	0.231	4.094	34.494	0.006

Concurrncy	Requests	Aborts	SAborts	Blocks
50	1263.200	0.000	0.000	61.800
550	138.921	0.000	0.000	6.794
Delay	NumExe	WBlock	Thruput	Conflict
176.209	43.096	39.701	255.391	0.050
4.558	0.844	1.814	50.542	0.008

Concurrncy	Requests	Aborts	SAborts	Blocks
60	1276.600	0.000	0.000	70.600
560	140.997	0.000	0.000	8.405
Delay	NumExe	WBlock	Thruput	Conflict
177.224	51.112	38.167	298.858	0.056
9.006	1.799	4.678	62.294	0.010

Concurrncy	Requests	Aborts	SAborts	Blocks
70	1289.200	0.000	0.000	79.600
570	185.070	0.000	0.000	6.053
Delay	NumExe	WBlock	Thruput	Conflict
181.919	58.736	38.218	340.540	0.063
5.158	1.946	2.885	81.690	0.011

Multiversion Timestamp Ordering Simulation (12, 0.50)

Concurrncy	Requests	Aborts	SAborts	Blocks
10	1499.600	29.000	0.000	16.400
510	30.605	4.817	0.000	4.317

Delay	NumExe	WBlock	Thruput	Conflict
193.230	9.811	22.323	51.244	0.030
7.504	0.060	3.043	1.699	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
20	1923.200	72.400	0.000	38.600
520	138.290	13.200	0.000	3.666

Delay	NumExe	WBlock	Thruput	Conflict
239.076	19.130	23.714	78.821	0.058
22.340	0.227	1.742	8.125	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
30	2382.400	122.200	0.000	72.600
530	204.565	16.987	0.000	9.604

Delay	NumExe	WBlock	Thruput	Conflict
297.002	28.323	21.607	93.423	0.082
19.433	0.275	1.385	12.363	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
40	3176.600	209.000	0.000	105.800
540	368.974	41.144	0.000	7.730

Delay	NumExe	WBlock	Thruput	Conflict
387.243	37.751	21.339	94.268	0.099
48.063	0.224	1.149	15.109	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
50	4191.000	321.400	0.000	161.600
550	74.536	17.188	0.000	5.463

Delay	NumExe	WBlock	Thruput	Conflict
498.275	46.759	20.412	87.113	0.115
59.152	0.487	1.429	10.435	0.003

Concurrncy	Requests	Aborts	SAborts	Blocks
60	5444.000	465.000	0.000	224.400
560	714.305	74.003	0.000	19.407

Delay	NumExe	WBlock	Thruput	Conflict
646.647	55.957	19.445	83.609	0.127
107.534	0.472	0.787	16.813	0.003

Concurrncy	Requests	Aborts	SAborts	Blocks
70	6731.600	618.200	0.000	293.600
570	601.704	64.676	0.000	19.855

Delay	NumExe	WBlock	Thruput	Conflict
761.463	65.222	18.882	76.336	0.135
70.508	0.573	1.594	15.676	0.003

TimeWarp Concurrency Control Simulation (12, 0.50)

Concurrncy	Requests	Undo's	SAborts	Anti's
10	1263.400	57.200	0.000	0.800
510	38.888	9.826	0.000	1.166
Delay	NumExe	WBlock	Thruput	Conflict
173.307	10.000	0.000	56.599	0.046
2.307	0.000	0.000	1.726	0.007

Concurrncy	Requests	Undo's	SAborts	Anti's
20	1437.200	172.800	0.000	5.600
519	93.307	6.997	0.000	2.871
Delay	NumExe	WBlock	Thruput	Conflict
194.880	20.000	0.000	94.638	0.124
5.460	0.000	0.000	5.345	0.006

Concurrncy	Requests	Undo's	SAborts	Anti's
30	1635.800	321.600	0.000	10.800
529	172.560	97.559	0.000	3.059
Delay	NumExe	WBlock	Thruput	Conflict
219.638	30.000	0.000	120.077	0.200
9.509	0.000	0.000	12.813	0.041

Concurrncy	Requests	Undo's	SAborts	Anti's
40	2011.600	542.200	0.000	24.200
533	127.462	28.610	0.000	2.786
Delay	NumExe	WBlock	Thruput	Conflict
252.299	40.000	0.000	122.338	0.283
10.584	0.000	0.000	6.543	0.023

Concurrncy	Requests	Undo's	SAborts	Anti's
50	2996.200	1086.200	0.000	65.800
541	447.201	199.720	0.000	21.423
Delay	NumExe	WBlock	Thruput	Conflict
299.973	50.000	0.000	98.854	0.383
13.588	0.000	0.000	13.746	0.025

Concurrncy	Requests	Undo's	SAborts	Anti's
60	4250.000	1847.200	0.000	113.000
539	527.915	296.059	0.000	27.452
Delay	NumExe	WBlock	Thruput	Conflict
350.448	60.000	0.000	78.462	0.460
15.734	0.000	0.000	7.983	0.025

Concurrncy	Requests	Undo's	SAborts	Anti's
70	4976.400	2292.600	0.000	141.400
549	1109.843	672.701	0.000	53.414
Delay	NumExe	WBlock	Thruput	Conflict
365.441	70.000	0.000	74.807	0.482
29.857	0.000	0.000	14.854	0.039

Realistic Recovery Protocol Simulation (12, 0.75)

Concurrncy	Requests	Aborts	SAborts	Blocks
10	1212.600	0.000	0.000	6.000
510	34.133	0.000	0.000	3.464
Delay	NumExe	WBlock	Thruput	Conflict
156.443	9.860	37.693	63.444	0.005
2.176	0.085	10.412	1.895	0.003

Concurrncy	Requests	Aborts	SAborts	Blocks
20	1227.000	0.000	0.000	14.400
520	45.830	0.000	0.000	5.004
Delay	NumExe	WBlock	Thruput	Conflict
158.274	19.429	32.909	120.413	0.012
3.011	0.226	6.478	9.361	0.004

Concurrncy	Requests	Aborts	SAborts	Blocks
30	1246.800	0.000	0.000	28.000
530	85.340	0.000	0.000	1.095
Delay	NumExe	WBlock	Thruput	Conflict
161.916	28.230	31.849	172.408	0.023
3.195	0.439	4.775	25.139	0.002

Concurrncy	Requests	Aborts	SAborts	Blocks
40	1256.800	0.000	0.000	35.000
540	136.217	0.000	0.000	8.877
Delay	NumExe	WBlock	Thruput	Conflict
163.614	37.567	31.127	226.006	0.028
5.058	0.746	2.356	39.260	0.007

Concurrncy	Requests	Aborts	SAborts	Blocks
50	1278.600	0.000	0.000	44.800
550	128.539	0.000	0.000	2.040
Delay	NumExe	WBlock	Thruput	Conflict
166.009	45.958	29.652	271.187	0.035
1.758	0.450	1.778	53.076	0.004

Concurrncy	Requests	Aborts	SAborts	Blocks
60	1286.200	0.000	0.000	45.600
560	151.807	0.000	0.000	3.382
Delay	NumExe	WBlock	Thruput	Conflict
163.971	54.961	30.390	322.086	0.036
2.102	0.447	2.714	73.898	0.006

Concurrncy	Requests	Aborts	SAborts	Blocks
70	1293.600	0.000	0.000	54.800
570	218.074	0.000	0.000	5.706
Delay	NumExe	WBlock	Thruput	Conflict
168.014	63.833	29.495	378.001	0.043
2.293	0.678	2.938	91.752	0.004

Multiversion Timestamp Ordering Simulation (12, 0.75)

Concurrncy	Requests	Aborts	SAborts	Blocks
10	1457.600	21.600	0.000	6.400
510	90.440	7.310	0.000	2.871
Delay	NumExe	WBlock	Thruput	Conflict
181.989	9.957	11.381	54.616	0.019
7.140	0.022	3.273	3.066	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
20	1778.800	50.400	0.000	14.200
520	142.532	8.499	0.000	2.135
Delay	NumExe	WBlock	Thruput	Conflict
215.364	19.827	13.656	89.243	0.036
8.459	0.031	1.111	9.634	0.002

Concurrncy	Requests	Aborts	SAborts	Blocks
30	2099.200	78.200	0.000	26.800
530	140.618	10.534	0.000	2.482
Delay	NumExe	WBlock	Thruput	Conflict
252.450	29.641	12.430	109.505	0.050
17.065	0.066	1.416	15.331	0.004

Concurrncy	Requests	Aborts	SAborts	Blocks
40	2626.800	125.800	0.000	34.200
540	237.185	15.992	0.000	8.976
Delay	NumExe	WBlock	Thruput	Conflict
303.407	39.422	13.061	119.219	0.061
20.044	0.098	1.465	20.523	0.005

Concurrncy	Requests	Aborts	SAborts	Blocks
50	3141.400	172.800	0.000	51.600
550	70.019	9.867	0.000	6.888
Delay	NumExe	WBlock	Thruput	Conflict
366.124	48.913	12.333	122.601	0.071
46.198	0.551	1.580	15.254	0.003

Concurrncy	Requests	Aborts	SAborts	Blocks
60	3739.000	228.000	0.000	56.000
560	422.665	31.330	0.000	7.720
Delay	NumExe	WBlock	Thruput	Conflict
405.311	58.727	12.157	125.770	0.076
27.879	0.580	1.562	28.887	0.004

Concurrncy	Requests	Aborts	SAborts	Blocks
70	4977.600	342.000	0.000	78.400
570	443.783	37.240	0.000	6.248
Delay	NumExe	WBlock	Thruput	Conflict
544.708	68.666	10.603	111.764	0.084
43.300	0.646	1.134	23.397	0.003

TimeWarp Concurrency Control Simulation (12, 0.75)

Concurrncy	Requests	Undo’s	SAborts	Anti’s
10	1278.200	32.800	0.000	1.200
510	50.253	11.374	0.000	0.748

Delay	NumExe	WBlock	Thruput	Conflict
172.153	10.000	0.000	56.536	0.026
3.518	0.000	0.000	2.250	0.008

Concurrncy	Requests	Undo’s	SAborts	Anti’s
20	1377.200	70.000	0.000	2.800
520	92.989	19.142	0.000	0.748

Delay	NumExe	WBlock	Thruput	Conflict
189.396	20.000	0.000	100.861	0.053
7.829	0.000	0.000	6.810	0.012

Concurrncy	Requests	Undo’s	SAborts	Anti’s
30	1599.600	133.600	0.000	9.600
530	151.849	25.476	0.000	2.332

Delay	NumExe	WBlock	Thruput	Conflict
207.636	30.000	0.000	127.198	0.089
6.773	0.000	0.000	9.696	0.010

Concurrncy	Requests	Undo’s	SAborts	Anti’s
40	1900.000	213.200	0.000	20.200
535	170.720	32.841	0.000	4.308

Delay	NumExe	WBlock	Thruput	Conflict
228.966	40.000	0.000	140.099	0.122
8.243	0.000	0.000	14.209	0.013

Concurrncy	Requests	Undo’s	SAborts	Anti’s
50	2223.600	294.600	0.000	34.400
540	181.217	45.227	0.000	9.625

Delay	NumExe	WBlock	Thruput	Conflict
243.575	50.000	0.000	144.293	0.148
10.629	0.000	0.000	14.465	0.017

Concurrncy	Requests	Undo’s	SAborts	Anti’s
60	3068.000	528.600	0.000	63.200
545	626.894	178.659	0.000	26.240

Delay	NumExe	WBlock	Thruput	Conflict
283.592	60.000	0.000	126.835	0.188
19.333	0.000	0.000	25.427	0.025

Concurrncy	Requests	Undo’s	SAborts	Anti’s
70	4855.600	923.000	0.000	143.400
545	1246.467	277.474	0.000	65.344

Delay	NumExe	WBlock	Thruput	Conflict
308.364	70.000	0.000	86.580	0.217
18.703	0.000	0.000	18.402	0.019