Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

SIMULATION OF DATABASE TRANSACTION MANAGEMENT PROTOCOLS: HYBRIDS AND VARIANTS OF TIME WARP

John A. Miller

Department of Computer Science
415 Graduate Studies
University of Georgia
Athens, Georgia 30602, U.S.A

ABSTRACT

The Time Warp protocol has been used successfully for parallel and distributed simulation. This paper considers several variants of Time Warp adapted for database transaction management, and also hybrids of Time Warp with traditional transaction management (concurrency control and recovery) protocols. Simulation models are constructed to simulate the behavior of these protocols, and an analysis of their relative performance is given. The results indicate that hybrids and variants of the Time Warp protocol can exhibit excellent performance.

1. INTRODUCTION

The combination of newer more demanding database applications, along with the increasing use of multiprocessor machines (both loosely and tightly coupled) produce a need and an opportunity for improved database performance. As levels of concurrency get higher and higher the impact on performance of transaction management protocols becomes more and more important. The use of common concurrency control protocols such as Two-Phase Locking may lead to an unacceptable performance bottleneck (Kim 1990), particularly for long-duration transactions. Possible solutions include: (1) finding a useful weaker correctness condition than serializability, (2) finding higher performance protocols to enforce serializability, and (3) exploiting special semantics of operations. Approaches (1) and (3) are relatively complicated for a naive user to work with, so a higher performance protocol for enforcing serializability is an attractive alternative.

One way to achieve acceptable response time and throughput is to avoid protocols that block processors, whether they are protocols for synchronization of simulation events or protocols for concurrency control. This claim is based on the intuition that it is better to use available processors, even if the work using those processors turns out to be incorrect and must be thrown away, than to block processing when processors are available. This argument assumes that the useful work done by transactions in the forward direction outweighs the non-productive work that has to be reversed. The simulation results presented in this paper and in (Miller 1986, 1992; Miller and Griffeth 1992) indicate that this assumption is justified. A second way to improve performance is to have multiple versions of objects (rather than updating an object, new versions of the object are created). This reduces the kinds of conflicts that can occur (e.g., write-write conflicts are no longer a problem). Many object-oriented database systems provide versioning capabilities (Kim 1990) for application reasons (Biliris 1989).

A secondary motivation for this work is to select protocols suitable for concurrency control in the query driven simulation system that is being developed at the University of Georgia (Miller and Weyrich 1989; Miller et al. 1990, 1991a, 1991b, 1992b; Kochut, Miller and Potter 1991). Query driven simulation is an approach to simulation modeling and analysis that uses database technology to automate the storage and retrieval of simulation data and simulation models. The fundamental tenet of query driven simulation is that simulationists or even naive users should see a simulation system as a sophisticated information system. This system should be able to store or (just as easily) generate information about the behavior of systems that users are studying. Some of the application areas to which we are applying query driven simulation include model management (Potter et al. 1990), decision support systems (Potter et al. 1992), and most recently genome mapping (Miller et al. 1992a). Since object-oriented database systems have such powerful data and behavioral structuring mechanisms, they provide an ideal foundation to support demanding applications like query driven simulation. On such a system, the concurrency control protocols used by the object-oriented database will have a major impact on performance (Griffeth and Miller 1985; Miller 1986). They will affect the performance of both ordinary database operations (queries and updates) and the execution of simulation models.

Interestingly enough, there are similarities between protocols for database concurrency control and protocols for synchronization of simulation events (Chandy and Misra 1979; Misra 1986). Richard Fujimoto has pointed out in (Fujimoto 1990) that the protocols developed for the difficult problem of parallel and distributed simulation can be adapted to other parallel/distributed applications. In the mid 1980's David Jefferson (Jefferson 1985; Jefferson and Motro 1986) adapted his *Time Warp* protocol to serve as a concurrency control protocol for database transactions. Time Warp is one of the more successful optimistic protocols used for parallel and distributed simulation. It is convenient that there are protocols that apply to both problems, since the approach used for the concurrency control protocol needs to be compatible with the approach used for distributed event synchronization. If one protocol blocks while the other attempts to proceed, we will negate the advantages of both protocols.

A similar protocol, which has been found to have good performance (Miller 1986) enforcing serializability in highly concurrent systems, is the *Multiversion Timestamp Ordering* protocol (Reed 1978). Protocols such as the Multiversion Timestamp Ordering and Time Warp protocols have the following advantages: (1) high effective concurrency levels, (2) deadlock free operation (because there is no or limited blocking), and (3) efficient operation unless there is a conflict. These types of protocols resolve conflicts by partially rolling back (undoing some operations) or completely aborting transactions. The fact that transactions are not blocked leads to higher effective concurrency levels. Thus, if aborts or partial rollbacks do not occur with great frequency, these protocols should exhibit excellent performance.

We extend Jefferson's work on the Time Warp protocol, developing several other variants of Jefferson's Time Warp protocol (Miller and Griffeth 1991, 1992; Miller 1992). In addition, we develop a Hybrid protocol that combines the best features from the Time Warp and Multiversion Timestamp Ordering protocols. The performance of these protocols is compared by simulating their execution on a multiprocessor.

2. DATABASE CORRECTNESS CONDITIONS

The two primary correctness conditions to maintain in a database are serializability and recoverability. *Serializability* maintains the consistency of data, by guaranteeing that if the individual transactions are correct then their combined effect will be correct (Papadimitriou, Bernstein and Rothnie 1977; Ullman 1982). Serializability is maintained by constraining the ordering of operations so that their effect is equivalent to a serial execution of the transactions. We also need to guarantee that the effects of successfully completed work are permanent. To this end, we provide a *commit* operation for a transaction to signal its successful completion and require that after it has committed, its effects will be part of the database forever.

Recoverability guarantees that after a failure (either transaction failure or system crash) the database can be recovered to a consistent state, at which point things can carry on as usual (Hadzilacos 1983; Gray et al. 1981). Recoverability is maintained by not allowing a transaction to commit until all the data it has read is committed. After a failure, we recover the database to a state including the effects of exactly those transactions that committed before the failure.

As a baseline for examining the performance of the Time Warp hybrids and variants, a simulation model was constructed for the Two-Phase Locking protocol (Two-Phase Locking is by far the most commonly used protocol in today's database systems). Under this protocol, objects to be read are read-locked, objects to be written are write-locked, and read-locks can be upgraded to write-locks. The transactions acquire their locks just before they are needed, and release them at commit time. Deadlocks are checked by using a WaitsFor graph, and are broken by aborting a transaction.

3. MULTIVERSION TIMESTAMP ORDERING PROTOCOL

Under the Multiversion Timestamp Ordering protocol, each 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 (Bernstein, Hadzilacos and Goodman 1987) for techniques to do so). Each version of an object is also timestamped with the timestamp of the 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,

$$\begin{aligned} ts(T_i) &= \text{timestamp of transaction } T_i \\ ws(v_k) &= \text{write-stamp of version } v_k \\ rs(v_k) &= \text{read-stamp of version } v_k \end{aligned}$$

where T_i is the i^{th} transaction to initiate. A write operation will simply write a new version of an object. The write $W_i(O_j)$ by transaction T_i 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. 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 transaction T_i reads from object O_j, the version with the largest write-stamp not exceeding the timestamp of T_i is returned.

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 transaction to be aborted. In particular, if 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 transaction reading version v_k should have really read the version that transaction T_i is attempting to write. For example, consider the following schedule,

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

where $R_i(o)/W_i(o)$ denotes a read/write of object o by transaction i. If transaction 8 attempts to write a version of O_{20}, it must be aborted since transaction 9 has already read the version written by 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. 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 (Griffeth and Miller 1985) or the commonly used Two-Phase Locking concurrency control protocol. Furthermore, so long as the transactions are two-stage (see Section 6) no deadlocks can occur. (Another possibility is to use the Optimistic Recovery protocol (Miller and Griffeth 1992). This approach was used in the study by (Liu, Miller and Parate 1992) with the result being that the overall performance is marginally worse than using the Realistic Recovery protocol.)

4. TIME WARP PROTOCOL

The Time Warp protocol has been studied extensively for parallel and 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 receiver 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 for use as a database concurrency control protocol (Jefferson 1985; Jefferson and Motro 1986).

In the schedule given in Section 3, transaction 7 also reads the version written by transaction 5. Unlike the read by transaction 9, this presents no problem for transaction 8 in its write attempt. To maintain serializability, it suffices to have transaction 9 change its read from transaction 5's version to transaction 8's version. Hence, we need to be able to partially rollback a transaction. The simplest approach is to back up transaction 9 to the point of the read of O_{20} (i.e., the ***read-in-question***) and then begin redoing the transaction. A more detailed analysis shows that the following would suffice:

1. "Redo" the read-in-question. In addition, handle erroneously included/excluded reads: (a) "undo" any reads that were previously performed, but should not have been as their guard condition now evaluates to false; and (b) "do for the first time" any reads that were previously skipped because a condition that uses the value of the read-in-question evaluated to false. We say such reads are conditionally dependent on the read-in-question.
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.

Note that in Time Warp, since transactions are only partially rolled back, their internal structure (including if statements) becomes important. Choosing to partially roll back 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 transaction are undone, they may cause other transactions to be partially rolled back leading to *cascaded rollbacks*. This would suggest that as the rate of conflict gets high, the performance of this protocol would degrade rapidly.

4.1. Jefferson's Original Protocol

The adaptation of the Time Warp algorithm, as originally done by Jefferson (Jefferson 1985; Jefferson and Motro 1986) is similar to the deferred commitment variant discussed below. However, Jefferson requires that all messages be processed in timestamp order. Thus the relative order in which reads are processed is significant. This requirement is above and beyond what is necessary to enforce serializability, so one would expect a tradeoff of possibly higher consistency for lesser performance. However, optimizations can be used to enhance performance.

4.2. Anti-Transaction Variant

The thorniest problem in using the Time Warp protocol is that some of the transactions that need to be partially rolled back cannot, since they have already committed. A possible remedy for this assumes that transactions can be effectively reversed through the use of anti-transactions. An anti-transaction is a system spawned transaction that reverses or counterbalances the effects of some transaction. Therefore, under this variant, conflicts are handled by rollbacks for uncommitted transactions, and by spawned anti-transactions for committed transactions.

4.3. Deferred Commitment Variant

In some applications, the use of anti-transactions (or compensating transactions) is infeasible. For example, if money is dispensed from an automated teller machine, reclamation of this money is not possible. The deferred commitment variant of the Time Warp protocol avoids the necessity of anti-transactions by delaying the commitment of transactions. Using this variant, transactions are committed in timestamp order. (Actually, commitment occurs when the system's estimate of Global Virtual Time (GVT) exceeds the timestamp of the transaction (Jefferson and Motro 1986). At this time, external outputs may be physically performed and older versions may be purged.) Although this may introduce a significant amount of blocking, it should not dramatically degrade throughput (Jefferson and Motro 1986). The main effect would be to increase the transaction completion time. However, it is possible that a few very long duration transactions could increase the delay encountered by average transactions to an unacceptable level.

5. HYBRID PROTOCOL

An alternative to these non-ideal possibilities is to use a Hybrid protocol. The Hybrid would use Time Warp so long as none of the reads-in-question belong to committed transactions, and use Multiversion Timestamp Ordering otherwise. Thus, when a write of a new version is attempted by T_i three possible actions may ensue: (1) The first possibility is to ***continue*** normally -- the read-stamp is smaller than $ts(T_i)$ so there is no conflict. (2) The second possibility is to ***partially roll back*** the transactions with questionable reads -- there is a conflict and all of the reads-in-question belong to uncommitted transactions. Since rollbacks are less costly than aborts, this is the preferred corrective action. (3) The final possibility is to ***abort*** transaction T_i -- there is a conflict and at least one of the reads-in-question belongs to a committed transaction. Unless one uses anti-transactions or substantially delays the commitment of transactions, rolling back transactions will not always suffice. If the new write destroys one of the information transfers where the reader has already committed, then the write will be prevented from occurring by aborting the transaction and making it start over with a new, larger timestamp.

5.1. Anti-Transaction Variant

Avoiding anti-transactions by choosing to abort, leads to additional complexities. To prevent aborts from corrupting the correctness of the database either a recovery protocol is needed, or again anti-transactions must be used (although less frequently). To see why this is so, consider the following case: Transaction T_2 reads a version written by transaction T_1, after which T_1 aborts. It is possible, even though T_1 started before T_2, that T_2 commits before T_1 finishes (or in this case aborts). Thus, T_2 has committed having read uncommitted (i.e. dirty) data, violating the recoverability condition.

5.2. Realistic Recovery Variant

Coupling a recovery protocol with the Hybrid protocol will solve this problem. Since we desire to keep blocking to a minimum, the Realistic Recovery (or Optimistic Recovery) protocol would make a good candidate. If a recovery protocol is not used then anti-transactions must be used to clean up the damage.

The *Realistic Recovery* protocol described in (Hadzilacos 1983) is a good choice for a multiversion database. To describe the Realistic Recovery protocol, let us consider how recovery protocols must operate. Data that has been written by transactions that have yet to commit is termed *dirty data*. From the definition of recoverability, we can see immediately that recoverability must be enforced by not allowing commits of transactions that have read dirty data. To do this, we can block either reads of dirty data or we can block commits of transactions that have read dirty data. In either case we have to block; from simulation results (Griffeth and Miller 1985; Miller 1986), the better choice appears to be to block reading of dirty data. The Realistic Recovery protocol simply blocks transactions attempting to read dirty data, until that data is committed. Writes, however, are not blocked since they simply produce a new version. (See the Appendix for a more detailed specification of this protocol.) We will now specify in detail how the Realistic Recovery protocol can be combined with the Hybrid protocol.

Read. When a read operation is processed, the most recent (from the time frame of the reading transaction) version is selected. If this version is committed, then the read is dispatched; otherwise the transaction is blocked until the version is committed. (If we allow parallelism within transactions, then much of this blocking can be reduced.) Since the read-stamp indicates the last transaction to have read a version it will only be updated once a read is actually dispatched, i.e., waiting readers do not effect the read-stamp of a version.

Write. When a write operation is processed, it will succeed without conflict or corrective action if it does not interfere with an information transfer that has already occurred. This will be the case if the read-stamp of the previous version is less than or equal to the timestamp of the writing transaction. If there is a conflict, then the action to be taken is as specified in the previous sub-section (i.e., abort the transaction, or roll back any transactions which should have read this new version). Notice that waiting reads do not produce conflicts. If a read by transaction 9 was waiting for transaction 5 to commit, while in the meantime transaction 8 writes a new version of the same object, then transaction 9 now simply waits for transaction 8 to commit. The write will be carried out by inserting (in timestamp order) the new version into the list of versions for the specified object and marking it uncommitted.

Commit. When a transaction commits, the version of each object it has written will be marked as committed. Any reads that were waiting on these versions will now be dispatched. Even though commitment is not delayed, it is still necessary to keep track of Global Virtual Time (GVT) for the purpose of purging older committed versions. A committed version may be purged when it can be assured that no additional reads will need this version (i.e., when GVT has exceeded this and a subsequent versions' timestamp).

Abort. When a transactions aborts, all of the versions which it has written will be removed from relevant version lists. Any reads that were waiting for the transaction to commit will now be assigned to the immediately preceding version. If this version is already committed, then these reads will be dispatched. Notice that cascaded aborts are not possible, since transactions are not allowed to read uncommitted (dirty) data.

Rollback. When a transaction is to be rolled back, the read-in-question will need to be redone, and a determination will be made as to what other read operations need to be undone or done for the first time. Similarly, writes that are conditionally dependent on the read-in-question will need to be either undone or done for the first time. Finally, any writes that are value dependent on the read-in-question will need to be undone and then redone (both actions can be handled by one anti-message if lazy cancellation is used). Notice that since reading of uncommitted versions is prohibited by the realistic recovery protocol, rollbacks will *never cascade* to other transactions.

6. SIMULATION MODELS

In this paper, we consider only *two-stage* transactions which do all their reading before writing. In (Miller 1986), non two-stage transactions where reads and writes are intermixed are considered. Two-stage transactions are very good from a performance standpoint, as *deadlock recovery* is greatly simplified. Indeed deadlocks cannot occur for several of the protocols, including the Realistic Recovery protocol. For this protocol, only reads can be blocked; they must wait for writes to be committed; since transactions are two-stage, once into their write stage they cannot be blocked; therefore it is impossible for a circular wait condition to develop. In addition, the throughput for two-stage transactions is significantly better than the throughput for non two-stage transactions (Miller 1986). 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.

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 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 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 (Allchin 1983; Garcia-Molina 1983).) Each transaction goes through a sequence of J stages each corresponding to an operation (i.e., after completing an operation a transaction goes to the next stage to perform its next operation). 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 (Miller 1986), 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 transaction at time t. In the derivation we showed that any service time distribution will work; it does not have to be exponential.

From a simulation modeling point of view, active objects in the database can be modeled as SIMODULA processes (Miller et al. 1990; Miller and Griffeth 1991, 1992). (SIMODULA is a simple process-oriented simulation system coded in Modula-2 (Miller, Weyrich and Suen 1988; Miller and Weyrich 1989; Miller et al. 1992b).) In the script procedure used for these processes, a transacting active object repeatedly performs transactions until the simulation is over. Within a transaction, k read/write operations are performed followed by a commit ($k = r + s$, where r = number of reads and s = number of writes). This usual behavior is modified if the transaction is aborted or partially rolled back. (A PreCommit operation is added for the Time Warp protocol to allow a 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 in (Miller and Griffeth 1992).

In our modeling study, we consider two possible dependency scenarios for the anti-transaction variant of the Time Warp protocol.

1. High Dependency Scenario. We assume for the sake of simplicity (both in modeling and in actual protocol implementations) that all of the writes are value dependent, and that no conditionally dependent reads need to be performed. Thus, a transaction in its write phase will only need to be backed up to the state preceding the write phase. If the transaction is still in its read phase, the situation is even better. The transaction simply needs to do its read-in-question again.
2. Low Dependency Scenario. More optimistic scenarios are certainly possible. For example, one or a few writes could be dependent on the read-in-question, while the other operations are independent.

In this paper, we present results for the low dependency scenario (see (Miller and Griffeth 1992) for results assuming high dependency). Specifically, to test the Time Warp protocol under an optimistic, yet realistic scenario, we examine the case where only one write is dependent on a given read.

7. SIMULATION ANALYSIS

With D fixed at 500 active objects and the object access time distributed uniformly from 0 to 20 milliseconds (expected value of 10 milliseconds), several experiments were conducted. Each experiment generated simulation results for the Two-Phase Locking (2PL), Multiversion Timestamp Ordering (MVTO), Time Warp (TW) (anti-transaction variant), and Hybrid (HYB) (anti-transaction variant) protocols. These results are summarized in Tables 1-3. These tables show the throughput (tps), and the number of rollbacks, anti-transactions, aborts, and blocks per 100 committed transactions. The method of batch means was used to control the simulation; 10 batches each representing 100 committed transactions were produced; the first batch was thrown out to reduce transient effects; the performance measures were computed on this basis.

Table 1:	$k = 12$,	$r = 3$,	$c = 3.5$,	$a = 2.5$
Table 2:	$k = 12$,	$r = 6$,	$c = 3.5$,	$a = 2.5$
Table 3:	$k = 12$,	$r = 9$,	$c = 3.5$,	$a = 2.5$

Note, $c = 3.5$ means that the expected time to commit is 35 milliseconds (or 3.5 times the read time), while $a = 2.5$ means that the expected time to abort is 25 milliseconds. The results given in Table 2 are summarized graphically in Figure 1. This figure displays the *throughput* in committed transactions per second (tps) of the four protocols (HYB: dash-dot, TW: dot, MVTO: dash, 2PL: solid). In particular, the throughput is plotted versus the concurrency level (M), which ranges from 10 (low concurrency) to 70 (high concurrency). Finally, Figure 2 shows a three dimensional plot of the throughput for the HYB protocol (x-axis: concurrency level (M), y-axis: number of reads (r), z-axis: tps).

7.1. Low Read-Write Ratio

The low read-write ratio scenario represents the case where writes are more common than reads (read probability = 0.25). For 2PL, this represents the worst scenario. The throughput at low concurrency is 43 tps, and proceeds to steadily drop from there. For the other protocols, however, their performance under this scenario is nearly as good as their performance when the read-write ratio is high. The explanation for this dichotomy is that the last three protocols (MVTO, TW and HYB) are all multiversion protocols which do not suffer from write-write conflicts, as 2PL does. Comparing the peak performance of the protocols, it is evident that the multiversion protocols perform 3 to 5 times better than 2PL. This is the case because the experiments were designed to test the protocols under conditions that are non-ideal for 2PL, namely high concurrency (and hardware support for it), and high levels of conflict (a database

Table 1: Summary of Simulation Results for k = 12 and r = 3

	2PL			MVTO			TW			HYB		
M	*blocks*	*aborts*	*tps*	*blocks*	*aborts*	*tps*	*rbacks*	*antis*	*tps*	*rbacks*	*aborts*	*tps*
10	93.78	6.33	43.43	16.00	18.78	55.97	36.00	0.33	58.04	36.67	0.44	58.85
20	218.33	34.78	35.27	43.33	55.44	88.32	75.33	1.00	108.26	84.78	1.22	109.09
30	344.00	77.00	26.62	69.22	91.56	13.81	130.44	3.22	145.46	146.89	2.78	148.02
40	563.22	167.00	18.34	98.11	128.00	129.48	203.00	6.22	172.44	223.44	4.22	179.59
50	793.00	272.67	13.44	136.89	183.22	135.52	259.00	7.00	199.76	325.11	8.33	199.47
60	1125.00	421.56	10.19	197.67	284.33	126.50	407.89	18.78	191.73	400.89	10.11	223.15
70	1636.36	668.67	7.53	266.67	379.78	122.45	485.11	22.56	203.81	584.11	16.67	223.07

Table 2: Summary of Simulation Results for k = 12 and r = 6

	2PL			MVTO			TW			HYB		
M	*blocks*	*aborts*	*tps*	*blocks*	*aborts*	*tps*	*rbacks*	*antis*	*tps*	*rbacks*	*aborts*	*tps*
10	65.33	4.22	50.75	15.22	29.33	52.21	38.67	0.78	57.46	41.56	0.44	57.74
20	161.11	23.44	52.26	37.78	74.56	81.26	97.11	1.89	104.20	103.89	2.78	104.06
30	273.22	60.22	37.44	67.56	144.11	91.43	162.56	5.89	134.99	189.44	5.33	137.55
40	413.00	126.44	27.80	109.33	198.56	101.91	263.89	13.44	152.69	265.33	8.22	166.54
50	579.00	212.44	21.31	159.00	329.78	93.12	339.22	18.56	171.12	414.56	15.89	176.86
60	863.78	370.11	15.83	227.67	456.44	89.97	644.00	43.33	142.68	546.00	20.67	187.64
70	1293.44	604.78	11.20	273.78	605.89	84.62	860.11	52.22	143.81	738.89	29.22	185.70

Table 3: Summary of Simulation Results for k = 12 and r = 9

	2PL			MVTO			TW			HYB		
M	*blocks*	*aborts*	*tps*	*blocks*	*aborts*	*tps*	*rbacks*	*antis*	*tps*	*rbacks*	*aborts*	*tps*
10	33.78	1.44	57.57	7.11	21.67	54.97	24.00	0.67	58.70	25.11	1.11	58.72
20	72.44	7.00	90.38	13.89	46.78	93.38	59.00	3.22	107.29	59.00	2.44	1109.12
30	124.00	19.11	83.10	25.33	87.57	111.89	100.00	5.89	145.42	91.00	4.22	151.21
40	168.11	35.00	80.47	35.67	134.00	123.40	147.67	11.00	173.64	139.22	7.78	184.61
50	236.22	78.33	52.86	49.33	177.22	131.79	182.33	14.00	200.16	194.78	12.22	205.85
60	281.00	101.33	52.77	63.00	249.33	129.07	258.67	21.67	207.75	232.67	13.78	230.22
70	359.22	173.11	40.48	79.22	331.22	122.65	386.78	35.67	194.90	305.89	19.22	239.27

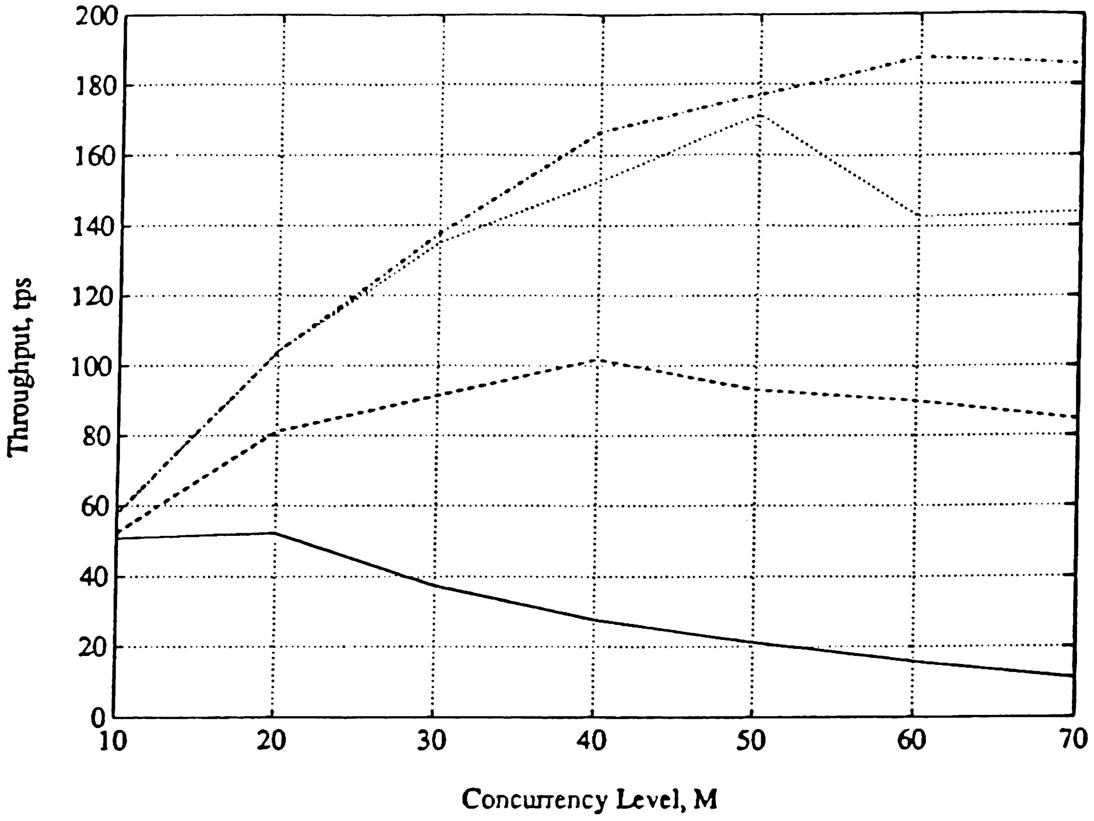

Figure 1: Protocol Performance

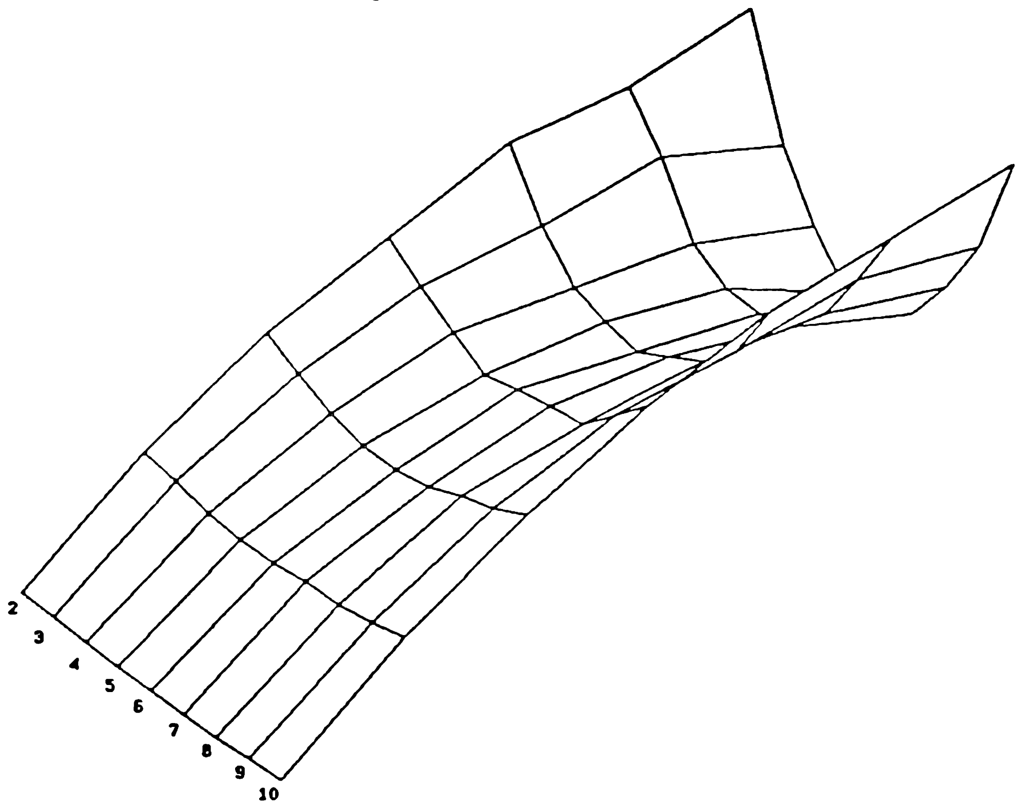

Figure 2: Surface for Hybrid Protocol

size of 500 objects is relatively small). Consequently, the rate of blocking and even aborting (due to deadlocks) becomes very high, resulting in poor performance. It is interesting to note that, although the abort rate for MVTO begins much higher than that of 2PL, as the concurrency level climbs this situation reverses. Comparing the multiversion protocols among themselves on the basis of peak performance, we find that TW (at 204 tps) performs 50% better than MVTO (at 136 tps), while HYB (at 223 tps) performs 64% better than MVTO.

7.2. Medium Read-Write Ratio

The medium read-write ratio scenario represents the case where reads and writes are equally likely (read probability = 0.50). Under this scenario, the performance of 2PL improves over the previous scenario, starting at 51 tps and increasing to 52 tps at a concurrency level of 20 before dropping. Although the multiversion protocols all had reduced performance under this scenario, they still outperformed 2PL by a factor of 2 to 4. Again, comparing the multiversion protocols among themselves on the basis of peak performance, we find that TW (at 171 tps) performs 68% better than MVTO (at 102 tps), while HYB (at 188 tps) performs 84% better than MVTO.

7.3. High Read-Write Ratio

The high read-write ratio scenario represents the case where reads are more common than writes (read probability = 0.75). Under this scenario, all of the protocols exhibit their best performance. This is fortunate since the high read-write ratio is the most likely scenario to occur in practice. The performance improvement under this scenario is particularly noticeable for 2PL. Its throughput begins at 57 tps, and then increases and stays in the range of 80 to 90 tps for the mid levels of concurrency. Still, the multiversion protocols exhibit substantially higher throughput. Once again, comparing the multiversion protocols among themselves on the basis of peak performance, we find that TW (at 208 tps) performs 58% better than MVTO (at 132 tps), while HYB (at 239 tps) performs 81% better than MVTO.

In general, for all of the multiversion protocols we find that the throughput drops off as the number of reads versus writes becomes balanced. At either extreme (mainly reads or mainly writes) the performance level is the greatest. This phenomenon is clearly indicated by the trough pattern exhibited in Figure 2 (i.e., for all levels of concurrency throughput is at a minimum when r is equal to 6).

8. CONCLUSIONS

Overall, we can conclude that the multiversion protocols provide ever increasing performance relative to Two-Phase Locking (2PL) as concurrency levels increase. Of the multiversion protocols, the newer ones, Time Warp (TW) and Hybrid (HYB) exhibit better performance than Multiversion Timestamp Ordering (MVTO). In comparing TW with MVTO, there is a substantial performance improvement with TW. Although the rate of corrective action is roughly the same for the two protocols, with MVTO relying upon blocks (due to the embedded Realistic Recovery protocol) and aborts, and TW relying upon rollbacks and anti-transactions, TW exhibits better performance since its corrective actions are less costly. Anti-transactions are costly corrective actions, but fortunately, the frequency of their occurrence within TW is 1 to 2 orders of magnitude less than the occurrence of rollbacks. Because rollbacks are relatively cheap in comparison to aborts, the primary corrective action used by MVTO, TW spends less time fixing problems and more time progressing the states of transactions forward. Finally, the HYB protocol was developed to reduce or eliminate the need for anti-transactions. Usually, HYB will respond to conflicts in the same way that TW would, by rolling back transactions. However, the protocol at times behaves like the MVTO protocol by aborting transactions to avoid using an anti-transaction. The HYB protocol performs even better than TW, with higher throughput, fewer rollbacks and fewer aborts (than TW's anti-transactions). Although the difference is not as great as the difference between TW and MVTO, it appears to be consistent. When a write produces a conflict, a decision must be made to either rollback the reader(s) or abort the writer. Evidently, HYB makes the right choice more often than either TW or MVTO, thus accounting for its better performance.

On the negative side, TW is susceptible to a type of data contention thrashing. When the concurrency level and probability of conflict are very high, the frequency of rollbacks begins to explode leading to a sharp downturn in throughput. The problem of cascaded rollbacks escalates to the point where more work is done in reversing time than in progressing time. This is particularly evident for the longer transactions, see (Miller 1992). Once the throughput reaches its peak, it begins an ever more precipitous drop. The throughput curve seems to drop dramatically when the rate at which anti-transactions execute begins to exceed the rate at which transactions commit. By throttling the concurrency level, though, high performance for TW can be maintained. The data we have collected so far, indicates that this explosive performance degradation is much more minor for HYB and has a later onset.

Some interesting related performance results may be found in our previous work. In the paper by (Liu, Miller and Parate 1992), it was found that multiversion protocols performed substantially better than their single version equivalents, if the system had the resources necessary for high concurrency (e.g, a multiprocessor systems with plenty of processors around to advance the states of transactions). The combination of the results from (Miller 1992; Miller and Griffeth 1992) and this paper indicate that TW exhibits performance superior to that of MVTO. Furthermore, as the degree of dependency within transactions is reduced, the cost of aborting a transaction becomes much greater than the cost of partially rolling back a transaction, thereby giving TW an even bigger advantage over traditional protocols.

In the field of simulation, the Time Warp protocol has been shown to be one of the better protocols for parallel and distributed simulation (Fujimoto 1990; Jefferson and Reiher 1991). In this paper, we have used simulation models to demonstrate that adaptations and hybrids of the Time Warp protocol can provide superior database transaction throughput. Our major findings may be summarized as follows: (1) So long as the probability of conflict does not get too high, the Time

Warp protocol has an excellent potential for high performance. Its peak throughput is higher than that of any traditional protocol we have studied. (2) The Hybrid protocol reduces and can even eliminate the need for anti-transactions, without resorting to lengthy blocking delays, and according to our simulation results provides even higher performance than Time Warp. Furthermore, the severity of performance degradation as concurrency levels become very high is less with the Hybrid protocol.

ACKNOWLEDGMENTS

The author would like to thank Dr. Nancy D. Griffeth of Bellcore, Dr. Krys J. Kochut of the University of Georgia, and Mr. Nilesh R. Parate of Oracle Corp. for their comments and suggestions related to this work. This research was partially supported by a University of Georgia Faculty Research Grant.

APPENDIX

This appendix explains in more detail how the Realistic Recovery (Hadzilacos 1983; Graham, Griffeth and Smith-Thomas 1984) protocol works. The Realistic Recovery protocol requires that each object maintain a list of operations that have been requested (some of which may have also been executed) and that each transaction maintain a list of objects at which it has requested write operations. When a read or write operation is requested of an object, the operation is added to the object's list of operations. When a transaction requests a commit or abort, it notifies each object at which it has requested a write. The object may then remove some writes from its list of operations or dispatch some reads. Committed writes must be left on the operation list until they are immediately followed by another committed write. Reads are removed once they have been dispatched. The following summarizes how each operation is handled:

1. Read. Dispatch the read if it is immediately preceded by a committed write operation. Otherwise, add the read to the end of the operation list.
2. Write. Add the write operation (including the value to be written) to the end of the operation list. Mark the write operation as uncommitted.
3. Commit. Notify the objects to mark all write operations of the transaction as committed. If any of these writes is immediately preceded by a committed write, the earlier committed write is removed from the operation list. If any of these writes is immediately followed by one or more read operations, the reads are dispatched.
4. Abort. Notify the objects to remove all write operations of the transaction. If after removal, any committed write is immediately preceded by another committed write, the earlier one may be removed. Again, if any committed write is immediately followed by one or more reads, the reads may be dispatched.

REFERENCES

Allchin, J.E. 1983. *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, Information and Computer Science, Georgia Tech.

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

Biliris, A. 1989. A data model for engineering design objects. *Proceedings of the IEEE Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, pp. 49-58.

Chandy, K.M., and J. Misra. 1979. Distributed simulation: A case study in the design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5:5, pp. 440-452.

Fujimoto, R.M. 1990. Optimistic approaches to parallel discrete event simulation. *Transactions of the Society for Computer Simulation*, 7:2, pp. 153-191.

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

Graham, M.H., N.D. Griffeth and B. Smith-Thomas. 1984. Reliable scheduling of transactions on unreliable systems. *Proceedings of the 1984 Conference on Principles of Database System*, Waterloo, Canada.

Gray, J.N., et al. 1981. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13:2, pp. 223-242.

Griffeth, N.D., and J.A. Miller. 1985. Performance modeling of database recovery protocols. *IEEE Transactions on Software Engineering*, SE-11:6, pp. 564-572.

Hadzilacos, V. 1983. An operational model for database system reliability. *Proceedings of the 1983 Conference on Principles of Distributed Computing*, pp. 244-257.

Jefferson, D.R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:3, pp. 404-425.

Jefferson, D.R., and A. Motro. 1986. The time warp mechanism for database concurrency control. *Proceedings of the Second International Conference on Data Engineering*, Los Angeles, CA, pp. 474-481.

Jefferson, D.R., and P. Reiher. 1991. Supercritical speedup. *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, pp. 159-168.

Kim, W. 1990. *Introduction to Object-Oriented Databases*, The MIT Press, Cambridge, MA.

Kochut, K.J., J.A. Miller and W.D. Potter. 1991. 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 Conference*, New Orleans, LA, pp. 139-145.

Liu, X., J.A. Miller and N.R. Parate. 1992. Transaction management for object-oriented databases: Performance advantages of using multiple versions. *Proceedings of the 25th Annual Simulation Symposium*, Orlando, FL, pp. 222-231.

Miller, J.A. 1986. *Markovian Analysis and Optimization of Database Recovery Protocols*, Ph.D. Thesis, Information and Computer Science, Georgia Tech.

Miller, J.A. 1992. Evaluation of hybrids and variants of time warp protocols for database transaction management. *IEEE Transactions of Knowledge and Data Engineering*. (in review)

Miller, J.A., and N.D. Griffeth. 1991. Performance modeling of database and simulation protocols: design choices for query driven simulation. *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, pp. 205-216.

Miller, J.A., and N.D. Griffeth. 1992. Performance of time warp protocols for transaction management in object-oriented systems. *International Journal in Computer Simulation*. (to appear)

Miller, J.A., W.D. Potter, K.J. Kochut and O.R. Weyrich, Jr. 1990. Model instantiation for query driven simulation in Active KDL. *Proceedings of the 23rd Annual Simulation Symposium*, Nashville, TN, pp. 15-32.

Miller, J.A., K.J. Kochut, W.D. Potter, E. Ucar and A.A. Keskin. 1991a. Query driven simulation using Active KDL: A functional object-oriented database system. *International Journal in Computer Simulation*, 1:1, pp. 1-30.

Miller, J.A., W.D. Potter, K.J. Kochut, A.A. Keskin and E. Ucar. 1991b. The Active KDL object-oriented database system and its application to simulation support. *Journal of Object-Oriented Programming*, Special Issue on Databases, 4:4, pp. 30-45.

Miller, J.A., J. Arnold, K.J. Kochut, A.J. Cuticchia and W.D. Potter. 1992a. Query driven simulation as a tool for genetic engineers. *Proceedings of the International Conference on Simulation in Engineering Education*, Newport Beach, CA, pp. 67-72.

Miller, J.A., O.R. Weyrich, Jr., W.D. Potter and V.C. Kessler. 1992b. The SIMODULA/OBJECTR query driven simulation support environment. In *Progress in Simulation*, 3, Leonard and Zobrist (Eds.). (to appear)

Miller, J.A., and O.R. Weyrich, Jr. 1989. Query driven simulation using SIMODULA. *Proceedings of the 22nd Annual Simulation Symposium*, Tampa, FL, pp. 167-181.

Miller, J.A., O.R. Weyrich, Jr. and D. Suen. 1988. A software engineering oriented comparison of simulation languages. *Proceedings of the 1988 Eastern Simulation Conference: Tools for the Simulationists*, Orlando, FL, pp. 97-104.

Misra, J. 1986. Distributed discrete-event simulation. *ACM Computing Surveys*, 18:1, pp. 39-65.

Papadimitriou, C.H., P.A. Bernstein and J.B. Rothnie. 1977. Computational problems related to database concurrency control. *Proceedings of the Conference on Theoretical Computer Science*, Waterloo, Canada.

Potter, W.D., J.A. Miller, K.J. Kochut and S.W. Wood. 1990. Supporting an intelligent simulation/modeling environment using the Active KDL object-oriented database programming language. *Proceedings of the Twenty-First Annual Pittsburgh Conference on Modeling and Simulation*, Pittsburgh, PA, pp. 1895-1900.

Potter, W.D., T.A. Byrd, J.A. Miller and K.J. Kochut. 1992. Extending decision support systems: The integration of data, knowledge, and model management. *Annals of Operations Research*. (to appear)

Reed, D.P. 1978. *Naming and Synchronization in a Decentralized Computer System*, Ph.D. Thesis, MIT.

Ullman, J.D. 1982. *Principles of Database Systems*, Computer Science Press, Rockville, MD.

AUTHOR BIOGRAPHY

JOHN A. MILLER is an assistant professor of Computer Science at the University of Georgia in Athens, Georgia. His research interests include simulation, object-oriented database systems, knowledge base systems, object-oriented programming and performance analysis. Dr. Miller received the BS degree in Applied Mathematics from Northwestern University in 1980, and the MS and PhD in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory as part of a co-operative education program. He is a member of the Association for Computing Machinery (ACM) and the Society for Computer Simulation (SCS), and is currently the President of the 26th Annual Simulation Symposium and a Guest Editor for the International Journal in Computer Simulation.