THE PERFORMANCE OF CONCURRENCY CONTROL ALGORITHMS FOR DATABASE MANAGEMENT SYSTEMS

by

Michael J. Carey

Computer Sciences Technical Report #530

January 1984

The Performance of Concurrency Control Algorithms for Database Management Systems

Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was supported by AFOSR Grant AFOSR-78-3596, NESC Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship while the author was at the University of California, Berkeley.

The Performance of Concurrency Control Algorithms for Database Management Systems

Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper describes a study of the performance of centralized concurrency control algorithms. An algorithm-independent simulation framework was developed in order to support comparative studies of various concurrency control algorithms. We describe this framework in detail and present performance results which were obtained for what we believe to be a representative cross-section of the many proposed algorithms. The basic algorithms studied include four locking algorithms, two timestamp algorithms, and one optimistic algorithm. Also, we briefly summarize studies of several multiple version algorithms and several hierarchical algorithms. We show that, in general, locking algorithms provide the best performance.

1. INTRODUCTION

1.1. Background

Much research on algorithm construction has been done in the area of concurrency control for both centralized (single-site) and distributed database systems. This research has led to the development of many concurrency control algorithms, most of which are based on one of three mechanisms: *locking* [Mena78, Rose78, Gray79, Lind79, Ston79], *timestamps* [Reed78, Thom79, Bern80, Bern81], and commit-time *validation* (also called optimistic concurrency control or certification) [Bada79, Casa79, Baye80, Kung81, Ceri82]. Bernstein and Goodman [Bern81] survey many of these algorithms and describe how new algorithms can be created by combining these mechanisms.

Given the ever-growing number of available concurrency control algorithms, the database system designer is faced with a difficult decision: Which concurrency control algorithm should be chosen? Several recent studies have evaluated concurrency control algorithm performance using qualitative, analytical, and simulation techniques. Bernstein and Goodman performed a comprehensive qualitative study which discussed performance issues for several

This research was supported by AFOSR Grant AFOSR-78-3596, NESC Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship at the University of California, Berkeley.

distributed locking and timestamp algorithms [Bern80]. Results of analytical studies of locking performance have been reported by Irani and Lin [Iran79], Potier and Leblanc [Poti80], and Goodman et al [Good83]. Simulation studies of locking done by Ries and Stonebraker provide insight into granularity versus concurrency tradeoffs [Ries77, Ries79a, Ries79b]. Analytical and simulation studies by Garcia-Molina [Garc79] provide some insight into the relative performance of several distributed variants of locking as well as a voting algorithm and a "ring" algorithm. Simulation studies by Lin and Nolte [Lin82, Lin83] provide some comparative performance results for several distributed locking and timestamp algorithms. A thesis by Galler [Gall82] provides a new analysis technique for locking, a qualitative method for comparing distributed algorithms, and some simulation results for distributed locking versus timestamps (which contradict those of Lin and Nolte). A study by Robinson [Robi82a, Robi82c] includes some experimental results on locking versus serial validation. Finally, Agrawal and Dewitt recently completed a performance study of several integrated concurrency control and recovery algorithms based on a combination of analytical and simulation techniques [Agra83a, Agra83b].

These performance studies are informative, but they fail to offer definitive results regarding the choice of a concurrency control algorithm for several reasons. First, the analytical and simulation studies have mostly examined only one or a few alternative algorithms. Second, the underlying system models and assumptions vary from study to study. Examples include whether transaction sizes are fixed or random, whether there is one or several classes of transactions, which system resources are modeled and which are omitted, and what level of detail is used in representing resources which are included in the models. It is thus difficult to arrive at general conclusions about the many alternative algorithms. Third, the models used are in some cases insufficiently detailed to reveal certain important effects. For example, some models group the I/O, CPU, and message delay times for transactions into a single random delay [Lin82, Lin83], in which case the performance benefit of achieving CPU-I/O overlap cannot be revealed. Finally, the few comprehensive studies of alternative algorithms which have been performed were of a non-quantitative nature.

1.2. Our Work

The remainder of this paper describes a comprehensive quantitative study of the performance of centralized concurrency control algorithms. An algorithm-independent simulation framework was developed in order to support fair comparative studies of various concurrency control algorithms. We describe this framework in detail and present performance results which were obtained for what we claim to be a representative cross-section of the many proposed algorithms. The basic concurrency control algorithms studied include four locking algorithms, two timestamp algorithms, and one optimistic algorithm. Also, we briefly summarize the results of studies of several multiple version algorithms and several hierarchical algorithms. We show that, in general, locking algorithms provide the best performance.

2. THE SIMULATION MODEL

This section outlines the structure and details of the simulation model which was used to evaluate the performance of the algorithms. The model was designed to support performance studies for a variety of centralized concurrency control algorithms [Care83c], so the design was made to be as algorithm-independent as possible.

2.1. The Workload Model

An important component of the simulation model is a transaction workload model. When a transaction is initiated from a terminal in the simulator, it is assigned a readset and a writeset. These determine the objects that the transaction will read and write during its execution. Two transaction classes, *large* and *small*, are recognized in order to aid in the modeling of realistic workloads. The class of a transaction is determined at transaction initiation time and is used to determine the manner in which the readset and writeset are to be assigned. Transaction classes, readsets, and writesets are generated using the workload parameters shown in Table 1.

The parameter *num_terms* determines the number of terminals, or level of multiprogramming, for the workload. The pararneter *restart_delay* determines the mean of an

Workload Parameters	
num_terms	level of multiprogramming
restart_delay	mean xact restart delay
db_size	size of database
gran_size	size of granules in database
small_prob	Pr(xact is small)
small_mean	mean size for small xacts
small_xact_type	type for small xacts
small_size_dist	size distribution for small xacts
small_write_prob	Pr(write X | read X) for small xacts
large_mean	mean size for large xacts
large_xact_type	type for large xacts
large_size_dist	size distribution for large xacts
large_write_prob	Pr(write X | read X) for large xacts

Table 1: Workload parameters for simulation.

exponential delay time required for a terminal to resubmit a transaction after finding that its current transaction has been restarted. The parameter *db_size* determines the number of objects in the database, and *gran_size* determines the number of objects in each granule of the database. When a transaction reads or writes an object, any associated concurrency control request is made for the granule which contains the object. To model read and write requests, objects and granules are given integer names ranging from 1 to *db_size* and 1 to $\lceil db\_size / gran\_size \rceil$, respectively. Object i is contained in granule $\lceil (i-1)/gran\_size \rceil + 1$.

The readset and writeset for a transaction are lists of the numbers of the objects to be read and written, respectively, by the transaction. These lists are assigned at transaction startup time. When a terminal initiates a transaction, *small_prob* is used to randomly determine the class of the transaction. If the class of the transaction is small, the parameters *small_mean*, *small_xact_type*, *small_size_dist*, and *small_write_prob* are used to choose the readset and writeset for the transaction as described below. Readsets and writesets for the class of large transactions are determined in the same manner using the *large_mean*, *large_xact_type*, *large_size_dist*, and *large_write_prob* parameters.

The readset size distribution for small transactions is given by *small_dist*. It may be constant, uniform, or exponential. If it is constant, the readset size is simply *small_mean*. If it is uniform, the readset size is chosen from a uniform distribution on the range from 1 to

2**small_mean*. The exponential case is not used in the experiments of this paper. The particular objects accessed are determined by the parameter *small_xact_type*, which determines the type (either random or sequential) for small transactions. If they are random, the readset is assigned by randomly selecting objects without replacement from the set of all objects in the database. In the sequential case, all objects in the readset are adjacent, so the readset is selected randomly from among all possible collections of adjacent objects of the appropriate size. Finally, given the readset, the writeset is determined as follows using the *small_write_prob* parameter: It is assumed that transactions read all objects which they write ("no blind writes"). When an object is placed in the readset, it is also placed in the writeset with probability *small_write_prob*.

2.2. The Queuing Model

Central to the detailed simulation approach used here is the closed queuing model of a single-site database system shown in Figure 1. This model is an extended version of the model of Ries [Ries77, Ries79a, Ries79b]. There are a fixed number of terminals from which transactions originate. When a new transaction begins running, it enters the *startup queue*, where processing tasks such as query analysis, authentication, and other preliminary processing steps are performed. Once this phase of transaction processing is complete, the transaction enters the *concurrency control queue* (or *cc queue*) and makes the first of its concurrency control requests. If this request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object is to be accessed prior to the next concurrency control request, the transaction will cycle through this queue several times. (An example of this is when there are several objects per granule.) When the next concurrency control request is required, the transaction re-enters the concurrency control queue and makes the request. It is assumed for convenience that transactions which read and write objects perform all of their reads before performing any writes.

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a deci-

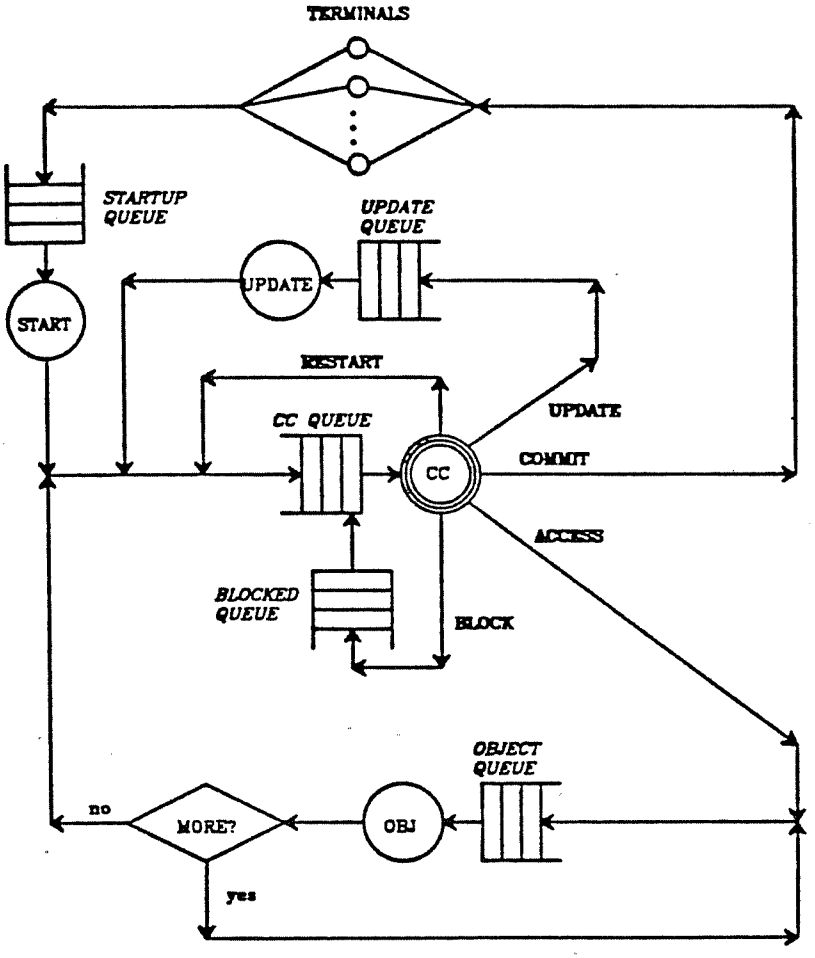

Figure 1: Logical database queuing model.

sion to restart the transaction, it goes to the back of the concurrency control queue after a randomly determined restart delay period of mean *restart_delay*; it then begins making all of its concurrency control requests and object accesses over again. Eventually, the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, it must first enter the *update queue* and write its deferred updates into the database.

Underlying the logical model of Figure 1 are two physical resources, the CPU and I/O (disk) resources. Associated with each logical service depicted in the figure (startup, concurrency control, object accesses, etc.) is some use of each of these two resources — each involves I/O processing followed by CPU processing. The amounts of CPU and I/O per logical service are specified as simulation parameters. All services compete for portions of the global I/O and CPU resources for their I/O and CPU cycles. The underlying physical system model is depicted in Figure 2. As shown, the physical model is simply a collection of terminals, a CPU server, and an I/O server. Each of the two servers has one queue for concurrency control service and another queue for all other service.

The scheduling policy used to allocate resources to transactions in the concurrency control I/O and CPU queues of the underlying physical model is FCFS (first-come, first-served). Concurrency control requests are thus processed one at a time, as they would be in an actual implementation. The resource allocation policies used for the normal I/O and CPU service queues of the physical model are FCFS and round-robin scheduling, respectively. These policies are again chosen to approximately model the characteristics which a real database system implementation would have. When requests for both concurrency control service and normal service are present at either resource, such as when one or more lock requests are pending while other transactions are processing objects, concurrency control service is given priority.

The parameters determining the service times (I/O and CPU) for the various logical resources in the model are given in Table 2. The parameters *startup_io* and *startup_cpu* are the amounts of I/O and CPU associated with transaction startup. Similarly, *obj_io* and *obj_cpu* are the amounts of I/O and CPU associated with reading or writing an object. Reading an object takes resources equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and *obj_io* at deferred update time, as it is assumed that transactions maintain deferred update lists in buffers in main memory. The parameters *cc_io* and *cc_cpu* are the amounts of I/O and CPU associated with a concurrency control request. All these parameters represent constant service time

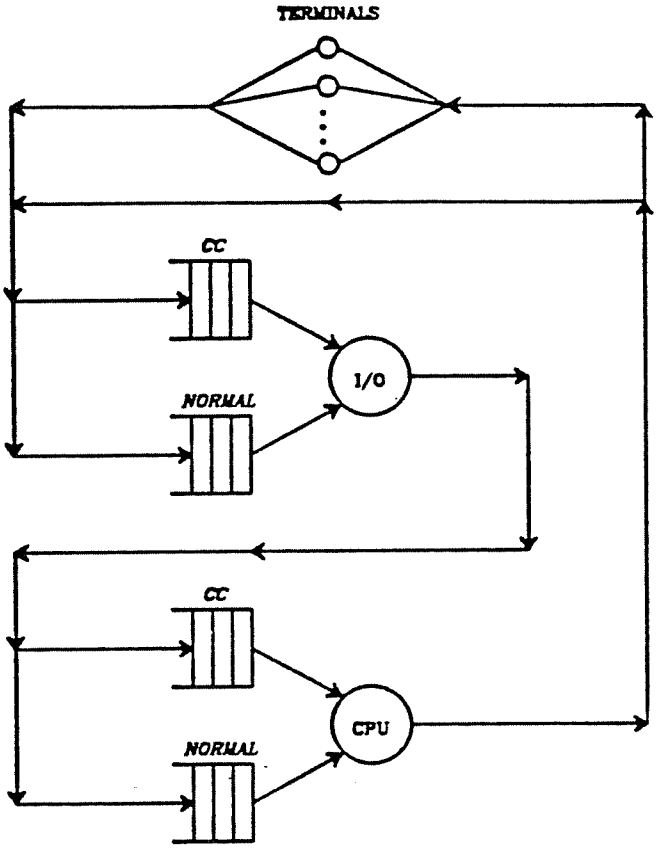

Figure 2: Physical database queuing model.

System Parameters	
startup_io	I/O time for transaction startup
startup_cpu	CPU time for transaction startup
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
cc_io	basic unit of concurrency control I/O time
cc_cpu	basic unit of concurrency control CPU time
stagger_mean	mean of exponential randomizing delay

Table 2: System parameters for simulation.

requirements rather than stochastic ones for simplicity. Finally, *stagger_mean* is the mean of an exponential time distribution which is used to randomly stagger transaction initiation times from terminals (not to model user thinking) each time a new transaction is started up. All parameters are specified in internal simulation time units, the unit of CPU time allocated to a transaction in one sweep of the round-robin allocation code for the simulator.

2.3. Algorithm Descriptions

Concurrency control algorithms are described for simulation purposes as a collection of four routines, *Init_CC_Algorithm*, *Request_Semantics*, *Commit_Semantics*, and *Update_Semantics*. Each routine is written in SIMPAS, a simulation language based on extending PASCAL with simulation-oriented constructs [Brya80], the language in which the simulator is implemented. *Init_CC_Algorithm* is called when the simulation starts up, and it is responsible for initializing all algorithm-dependent data structures and variables. The other three routines are responsible for implementing the semantics of the concurrency control algorithm being modeled. *Request_Semantics* handles concurrency control requests made by transactions before they reach their commit point. *Commit_Semantics* is invoked when a transaction reaches its commit point. *Update_Semantics* is called after a transaction has finished writing its deferred updates. Each of the latter three routines returns information to the simulator about how much simulation time to charge for CPU and I/O associated with concurrency control processing.

2.4. Some Other Details

In this section, we briefly discuss two issues which are relevant to the results presented in the remainder of the paper. We outline the way that concurrency control costs are modeled and the statistical methods used to analyze the experimental results.

2.4.1. Concurrency Control Costs

In order to simulate concurrency control algorithms, it is necessary to make some assumptions about their costs. To evaluate them fairly and determine how their blocking and restart decisions affect performance, the assumptions made for each of the algorithm simulations are consistent. It is assumed that the unit costs for concurrency control operations in locking, timestamps, and validation are all the same, *cc_cpu* and *cc_io*, as a first-order approximation. This is reasonable since the basic steps in each algorithm, such as setting a lock or checking a timestamp, involve doing one or two table lookups per request. Thus, the

costs of processing requests in the various algorithms are not likely to differ by more than small constant factors. (This is borne out by the cost results reported in [Care83b].)

To illustrate how concurrency control costs are modeled, let us take as an example the 2PL algorithm to be described subsequently. In this algorithm, transactions set read locks on items which they read, and they later upgrade read locks to write locks for items which they also write. Consider a transaction which makes N_r granule read requests and N_w granule write requests. A CPU cost of *cc_cpu* and an I/O cost of *cc_io* are assessed each time the transaction makes a read or write lock request for a granule. For 2PL, then, the total concurrency control CPU and I/O costs for the transaction in the absence of restarts are $(N_r+N_w)cc\_cpu$ and $(N_r+N_w)cc\_io$, respectively. The concurrency control costs for other locking algorithms, timestamp algorithms, and validation algorithms are determined similarly (see [Care83c] for details).

Of course, if the transaction is restarted, it will incur the additional costs involved in executing again from the beginning. These include all the costs associated with reading and writing the objects that the transaction accesses, plus the costs for making all of its concurrency control requests over again.

2.4.2. Statistical Analysis

In the simulation experiments reported here, the performance metric used is the transaction throughput rate. Mean throughput results and 90% confidence intervals for these results were obtained from the simulations using a variant of the *batch means* [Sarg76] approach to simulation output analysis. The approach used is due to Wolf [Wolf83]; it differs from the usual batch means approach in that an attempt is made to account for the correlation between adjacent batches. Briefly, we assume that adjacent batches are positively correlated, that non-adjacent batches are not correlated, and that the correlation between a pair of adjacent batches is independent of the pair under consideration. We then estimate this correlation and use it in computing a confidence interval for the mean throughput. In the remainder of this paper, we present only the mean throughput results. However, we point out

only performance differences which are significant in the sense that their confidence intervals do not overlap. More information and data on the statistical approach used in our experiments may be found in Appendix 3 of [Care83c].

3. BASIC CONCURRENCY CONTROL ALGORITHMS

In this section, we describe our performance results for *basic* concurrency control algorithms. The class of basic concurrency control algorithms consists of those algorithms which operate using the most recent version of each data granule and a single level of granularity. A later section will summarize results for *multiple version* algorithms, which use older versions of data items to provide greater potential concurrency (usually for large, read-only transactions), and *hierarchical* algorithms, which use a hierarchy of two or more granule sizes (e.g., files and pages) in an attempt to trade lower concurrency for reduced concurrency control costs.

3.1. Algorithms Studied

Given the large number of proposed concurrency control algorithms, it is not possible to study all (or even a significant fraction) of them. Thus, we chose to study a collection of algorithms which we believe to be a representative cross-section of the proposals. We believe our collection of algorithms to be representative because it contains algorithms which use a range of blocking and restarts in resolving conflicts: Some just use blocking, some just use restarts, and some use a mix of these two tactics. Also, the algorithms vary as to when conflicts are detected and dealt with.

Our study of basic concurrency control algorithms includes seven algorithms: four locking algorithms, two timestamp algorithms, and one optimistic algorithm. We describe each of the algorithms briefly below.

Dynamic Two-Phase Locking (2PL). Transactions set read locks on granules which they read, and these locks are later upgraded to write locks for granules which they also write. If a lock request is denied, the requesting transaction is blocked. A waits-for graph of transac-

tions is maintained [Gray79], and deadlock detection is performed each time a transaction blocks. If a deadlock is discovered, the transaction which just blocked and caused the deadlock is chosen as the victim and restarted. (This may not be the best victim choice, but it was selected initially for ease of implementation.)

Wait–Die Two–Phase Locking (WD). This algorithm is like 2PL, except that wait-die deadlock prevention [Rose78] is used instead of deadlock detection. When a lock request from transaction T_i conflicts with a lock held by T_j, T_i is permitted to wait only if it started running before T_j. Otherwise, T_i is restarted.

Dynamic Two–Phase Locking, No Upgrades (2PLW). The only difference between this algorithm and 2PL is that, if a transaction both reads and writes a granule x, it does not request a read lock and then later upgrade it to a write lock. Rather, for granules which are eventually written, write locks are requested the first time the granule is accessed, eliminating upgrades. Full deadlock detection is employed as before.

Exclusive Preclaimed Two–Phase Locking (PRE). In this algorithm, all granules read or written by a transaction are locked in exclusive (write) mode at transaction startup time. If a transaction cannot obtain all required locks, it blocks without obtaining any of its locks and waits until all required locks are available. The use of only exclusive locks will result in more blocking than would occur using both read and write locks; this will aid in establishing later performance results about the effects of blocking.

Basic Timestamp Ordering (BTO). This algorithm is described by Bernstein and Goodman [Bern81]: Each transaction T has a timestamp, $TS(T)$, which is issued when T begins executing. Each data granule x in the database has a read timestamp, $R\text{-}TS(x)$, and a write timestamp, $W\text{-}TS(x)$, which record the timestamps of the latest reader and writer (respectively) for x. A read request from T for x is granted only if $TS(T) \geq W\text{-}TS(x)$, and a write request from T for x is granted only if $TS(T) \geq R\text{-}TS(x)$ and $TS(T) \geq W\text{-}TS(x)$. Transactions whose requests are not granted are restarted. In the version of the algorithm tested here, read requests are checked dynamically; writes, however, go to a deferred update list, and

write timestamps are all checked together at commit time.

Basic Timestamp Ordering, Thomas Write Rule (TWW). This is BTO with a slight modification [Bern81]: When a transaction T makes a write request for an object x and $TS(T) \geq R\text{-}TS(x)$ but $TS(T) < W\text{-}TS(x)$, BTO will restart T. In this algorithm, T's request is instead granted, but the actual (outdated) write is ignored. In all of our experiments, TWW and BTO always performed identically. This is due to the fact that, under the "no blind writes" assumption which underlies the mechanism by which writesets are assigned in our simulations, TWW is identical to BTO [Care83c]. Thus, results for TWW are not presented separately.

Serial Validation (SV). This is the optimistic concurrency control algorithm of Kung and Robinson [Kung81]. Transactions record their read and write sets, and a transaction is restarted when it reaches its commit point if any granule in its readset has been written by a transaction which committed during its lifetime. Our version differs from the original algorithm in that we use startup and commit timestamps to enforce the semantics of the algorithm (instead of keeping old write sets around and explicitly checking for readset/writeset intersections) [Care83d]. However, transactions committed (restarted) using our version would be committed (restarted) by the original algorithm as well, so the change in implementation strategy does not affect the outcome of our performance experiments. (The only difference is a slight reduction in concurrency control costs using our scheme.)

3.2. Performance Experiments and Results

We performed six different performance experiments on these basic concurrency control algorithms. In this section we present some of the more interesting experimental data and summarize the remainder of our results. All of the experiments reported in this section were run with *batch_time* = 50,000 and *num_batches* = 20 (for a total of 1,000,000 simulation time units) in order to obtain tight confidence intervals. One simulation time unit represents one millisecond of simulated time for these experiments.

3.2.1. Experiment 1: Transaction Size

The first experiment examined the performance characteristics of the basic algorithms under homogeneous workloads of fixed-size transactions. Parameters varied in this experiment were the granularity of the database and the size of transactions. The purpose of this experiment was to observe the behavior of the algorithms of interest under various conflict probabilities and transaction sizes.

The system parameter settings for this experiment are given in simulated time in Table 3. The relevant workload parameter settings are given in Table 4. (These database and granularity parameters might correspond, for example, to a 40 megabyte database where objects are 4K byte pages and granules are groups of one or more pages.) Transactions read a fixed number of objects selected at random from among all objects in the database; this number was set at 1, 2, 5, 10, 15, and 30 for different simulation runs in the experiment. Transactions

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	35
startup_cpu	10
obj_io	35
obj_cpu	10
cc_io	0
cc_cpu	1
stagger_mean	20

Table 3: System parameters for experiment 1.

Workload Parameters	
db_size	10000 objects
gran_size	varied from 1 to 10000 objects/granule
num_terms	10
restart_delay	1 second
small_prob	1.0
small_mean	varied from 1 to 30 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5

Table 4: Workload parameters for experiment 1.

update each object that they read with 50% probability.

Tables 5 and 6 show the throughput results (in transactions/second) and the number of transactions restarted for transactions of size 2. Tables 7 and 8 give these figures for transactions of size 10. A number of observations can be made from these results. Examining Table 5, there is no significant performance difference between the algorithms when database is organized as many granules. This also holds for Table 7. The implication of this is that all algorithms perform equally well when the probability of conflicts is low. As the probability of conflicts increases, however, differences do begin to appear. This is visible in the results for coarser granularities in the two tables. We also observed differences at finer granularities as the transaction size parameter was increased to 15 and then 30.

Examining the coarser granularity results in both Tables 5 and 7, we see that the PRE and 2PLW algorithms yield the best performance. Tables 6 and 8 show that these are the two algorithms which caused the fewest restarts[†]. No restarts occurred with PRE, as it is deadlock free, and few occurred with 2PLW. In Table 5, 2PL is the next best algorithm, followed by WD, and then finally by SV and BTO. An examination of Table 6 shows that the ordering of the algorithms by throughput corresponds closely to the opposite of their ordering by restart counts. This implies that having more restarts leads to poorer performance, and therefore that restarts should be avoided if possible. Further evidence supporting this conclusion was observed in the other cases tested, and we will see more evidence in the remainder of these experiments as well. The main trend observed as transaction size was increased to 15 and 30, since restarts are more costly for large transactions, was that performance differences among the algorithms are more pronounced for larger transactions.

In Table 7, 2PL outperforms WD and SV under low conflicts, but it does worse than WD and SV at higher conflict probabilities (when *Grans* = 10, for example). The poor performance of 2PL occurs because its simplistic deadlock victim selection policy can lead to thrashing-like restart behavior when conflicts are likely [Care83c]. This same effect occurred for 2PLW when

[†]The one exception to this statement, which occurs in Table 8, will be discussed shortly.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	3.400	3.638	6.479	6.241	2.595	3.634
10	5.974	5.790	7.096	7.161	5.119	5.231
100	7.039	6.966	7.161	7.163	6.906	6.714
1000	7.152	7.149	7.161	7.161	7.138	7.113
10000	7.159	7.159	7.160	7.161	7.158	7.158

Table 5: Throughput, experiment 1, *small_mean* = 2.

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	6311	7638	0	0	7594	6529
10	2366	4633	242	0	4071	3844
100	247	706	3	0	508	879
1000	20	73	0	0	51	102
10000	2	11	0	0	4	9

Table 6: Restarts, experiment 1, *small_mean* = 2.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.281	0.240	1.518	1.425	0.001	0.336
10	0.074	0.234	0.432	1.415	0.004	0.355
100	0.827	0.701	1.414	1.759	0.235	0.784
1000	1.676	1.599	1.784	1.790	1.473	1.480
10000	1.776	1.770	1.788	1.788	1.763	1.749

Table 7: Throughput, experiment 1, *small_mean* = 10.

Restart Counts versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	2351	4142	0	0	2855	2323
10	2816	3837	4223	0	2848	2293
100	1543	2360	769	0	2482	1612
1000	180	388	7	0	501	485
10000	16	31	0	0	38	56

Table 8: Restarts, experiment 1, *small_mean* = 10.

the transaction size parameter was set at 15 and 30. BTO performs the worst here, especially at higher conflict probabilities. Its poor performance is due to a problem with the BTO algorithm: It is possible to have "cyclic restarts", with two or more conflicting transactions becoming involved in cycles where they continuously restart each other instead of making progress [Date82, Care83c, Ullm83]. At the coarsest granularities, SV actually outperforms WD. This is because, using WD, a transaction which is restarted due to a conflict can actually

be restarted several times before the conflict disappears [Rose78]; this does not happen in SV, so SV is in some sense more "stable" than WD.

Comparing Tables 7 and 8, we again see that higher numbers of restarts result in lower throughputs in most cases. There are a few exceptions, however. One example is WD, which seems to have performed better than its number of restarts would imply. This occurs because WD restarts the younger transaction when a conflict occurs, and thus tends to restart transactions which have completed less work (wasting fewer resources in the process). This implies that it is better to restart a transaction which has done little work if a restart is unavoidable. Another example is the performance of 2PL versus that of 2PLW at *Grans* = 10. 2PLW outperforms 2PL even though it causes more restarts because, in 2PLW, all restarts occur as locks are set in response to read requests. In 2PL, restarts can still occur as write requests are processed. Thus, the average 2PL restart wastes more resources than a 2PLW restart.

3.2.2. Experiment 2: Access Patterns

This experiment investigated the performance characteristics of the seven concurrency control algorithms under two workloads consisting solely of large transactions. One workload consisted of random transactions of mean size 30, and the other workload consisted of sequential transactions of mean size 30. In both cases, transaction sizes were chosen from a uniform distribution. The granularity of the database was varied in order to vary the probability of conflicts. The objective of this experiment was to observe the effects of random versus sequential object access patterns on algorithm performance.

The results of this experiment were similar to those of Experiment 1. Again, restart behavior determined performance. The main performance difference observed between the random and sequential transaction cases was that the 2PLW and 2PL locking algorithms had even more of a performance advantage with sequential transactions. This is because, with sequential transactions, 2PLW becomes deadlock-free (like PRE), and the only source of deadlocks for 2PL in this case is when read locks are upgraded to write locks. Thus, fewer

restarts occurred for these algorithms in the sequential case, resulting in better performance.

3.2.2.1. Experiment 3: Mixed Workload

This experiment investigated the performance characteristics of the seven concurrency control algorithms under a workload consisting of a mix of large and small transactions. The fraction of small transactions in the mix was varied in steps of 20% to investigate algorithm performance under different combinations of small and large transactions. The granularity of the database was also varied in order to vary the probability of conflicts. Table 9 gives the relevant workload parameters for this experiment.

Tables 10 and 11 give the throughput results obtained for the cases of 80% small transactions and 20% small transactions, respectively. The locking algorithms are again seen to outperform the alternatives. In particular, SV rarely outperforms a locking algorithm in these cases, only beating 2PL at very coarse granularities where 2PL's bad victim selection criteria causes it to perform poorly. As before, PRE and 2PLW are the dominant algorithms.

Workload Parameters	
db_size	10000 objects
gran_size	varied from 1 to 10000 objects/granule
num_terms	10
restart_delay	1 second
small_prob	varied from 0.2 to 0.8
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	30 objects
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.1

Table 9: Workload parameters for experiment 3.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.101	0.450	2.521	2.371	0.022	0.333
10	0.853	1.551	2.830	2.860	0.338	0.963
100	2.352	2.580	2.865	2.861	1.246	2.185
1000	2.675	2.752	2.859	2.864	2.415	2.504
10000	2.803	2.777	2.860	2.864	2.634	2.554

Table 10: Throughput, experiment 3, *small_prob* = 0.8.

Throughput versus Granularity						
Grans	2PL	WD	2PLW	PRE	BTO	SV
1	0.066	0.091	0.919	0.771	0.000	0.111
10	0.464	0.517	0.967	0.963	0.124	0.450
100	0.883	0.894	0.966	0.964	0.691	0.858
1000	0.930	0.945	0.966	0.969	0.775	0.905
10000	0.942	0.944	0.966	0.967	0.874	0.913

Table 11: Throughput, experiment 3, *small_prob* = 0.2.

3.2.3. Experiment 4: Multiprogramming Level

This experiment investigated the effects of the multiprogramming level on the results of the previous experiments. A portion of Experiment 3 was repeated with the multiprogramming level set to 5 and then 20 transactions. It was found that the multiprogramming level affects performance in an *absolute* sense, as changing it changes the probability of conflicts, but the *relative* performance of the basic concurrency control algorithms was not affected by changes in the multiprogramming level. Also, the effect of the level of multiprogramming on throughput without a concurrency control algorithm was investigated for the same portion of Experiment 3. It was found that having 4 or more non-conflicting transactions ready to run was sufficient to allow the system to reach its maximum throughput capacity.

3.2.4. Experiment 5: System Balance

This experiment investigated the effects of system balance on the results of the previous experiments. A portion of Experiment 3 was repeated with the system parameters set to yield I/O-boundedness, CPU-boundedness, and then good balance. It was found that system balance is another factor which is not significant with respect to the relative performance of

the algorithms.

3.2.5. Experiment 6: Concurrency Control Cost

In this experiment, a portion of Experiment 3 was repeated with the concurrency control cost parameters modified to investigate their importance. Experiments were run with concurrency control being free (*cc_cpu* = 0, *cc_io* = 0), expensive in terms of CPU cost (*cc_cpu* = 5 milliseconds, *cc_io* = 0), and expensive in terms of I/O cost (*cc_cpu* = 1 millisecond, *cc_io* = 35 milliseconds). We found that the effects of concurrency control cost are negligible as long as they are small compared to the costs associated with accessing objects (i.e., when *cc_io* = 0). In the last case, where concurrency control was very expensive, we observed results similar to those of Ries [Ries77]: A medium granularity was optimal for all algorithms rather than the finest granularity, as coarse granularities led to high conflict probabilities and fine granularities led to high concurrency control overhead. The relative performance of the algorithms, however, was not affected by changes in the concurrency control cost parameters.

4. MULTIPLE VERSION AND HIERARCHICAL ALGORITHMS

In addition to the basic concurrency control algorithm studies which have been described, studies were performed on several multiple version algorithms and several hierarchical algorithms. Space precludes a detailed description of these studies, but the main results are summarized in this section. The interested reader can find a detailed description of these experiments and results in [Care83c].

4.1. Multiple Version Algorithms

The multiple version algorithms which were studied include the CCA version pool algorithm [Chan82], which is based on locking, the multiple version timestamp ordering algorithm of Reed [Reed78], and a multiple version optimistic algorithm [Care83d]. In modeling the costs associated with these algorithms, we made the simplistic assumption that the cost of accessing any version of a granule is the same as the cost of accessing its most recent

version. We intend to investigate algorithm performance using a more realistic cost model in the future.

We performed several experiments, comparing the performance of the multiple version algorithms with each other and with their single version counterparts. For workloads consisting of large read-only transactions and small update transactions, all of the multiple version algorithms performed alike, enabling transactions to execute with little interference. Multiple versions did little to improve the performance of locking for the cases studied, but they did help improve performance both for timestamp ordering and optimistic concurrency control. The performance of mixes with mostly small update transactions and a small fraction of large read-only transactions was improved the most using multiple versions.

4.2. Hierarchical Algorithms

The hierarchical algorithms studied include hierarchical variants of preclaimed exclusive locking, basic timestamp ordering, serial validation, and multiversion timestamp ordering. Descriptions of the latter three hierarchical algorithms may be found in [Care83a]. The performance of the algorithms in a two-level granularity hierarchy was investigated under the assumption that the cost of processing a concurrency control request for fine granules was twice that for coarse granularity requests.

We performed several experiments, comparing the performance of the hierarchical algorithms with each other and with their single granularity counterparts for a mix of large and small transactions. No performance improvements were obtained using the hierarchical algorithms under normal concurrency control costs (i.e., those used in evaluating the basic concurrency control algorithms). However, under very high concurrency control cost settings, all four of the hierarchical algorithms improved system performance by reducing concurrency control costs. Hierarchical preclaimed locking performed the best of the algorithms examined.

5. CONCLUSIONS

We have described a study of the performance of centralized concurrency control algorithms. An algorithm-independent simulation framework was presented and used to study the performance of a number of basic algorithms: four locking algorithms, two timestamp algorithms, and one optimistic algorithm. All were found to perform equally well when conflicts are rare. When conflicts are not rare, it was shown that blocking is the preferred tactic for handling conflicts, as restarts waste resources and consequently lead to poorer performance. Hence, locking algorithms appear to be the concurrency control method of choice for centralized database systems. Preclaimed locking (PRE) consistently outperformed the other alternatives, and dynamic two-phase locking without lock upgrades (2PLW) also performed very well.

In some systems, especially those in which concurrency control is handled entirely at the physical data level, it may be impossible to accurately preclaim only the required locks or to predict writes at the time when reads are performed. In such systems, PRE and 2PLW will not be viable. Based on our results, dynamic two-phase locking (2PL) is recommended in this situation. However, the simple "pick the current blocker" criteria for deadlock victim selection is not recommended, as it led to poor performance under high conflict probabilities[†]. Instead, our results indicate that better policies will choose transactions which have completed little work. In situations where deadlock-free locking protocols are a feasible alternative [Silb80, Kort82, Moha82], such protocols are of course recommended.

The results of studies of the performance of several multiple version and hierarchical algorithms were summarized. For workloads consisting of large read-only transactions and small update transactions, the multiple version algorithms studied performed alike. Multiple versions did little to improve the performance of locking for the cases studied, but they were beneficial for timestamp ordering and optimistic concurrency control. Of the hierarchical algorithms examined, hierarchical locking performed the best. However, none of the

[†]This also applies to victim selection for 2PLW.

hierarchical algorithms studied were beneficial under normal concurrency control costs.

A number of questions remain for future research. First of all, our results only apply directly to centralized database systems. We hypothesize that locking algorithms will also provide the best performance in distributed database systems, and we plan to investigate this hypothesis through further simulation modeling and experimentation. Second, it appears worthwhile to make a thorough study of alternative deadlock-handling strategies for database systems. We intend to use our current simulation environment to perform such a study. Finally, we would like to verify our simulation results by adding an algorithm-independent concurrency control module to WiSS (the Wisconsin Storage System) [Chou83] and running performance experiments on this actual system.

ACKNOWLEDGEMENTS

Michael Stonebraker provided comments, suggestions, and support throughout this research. Discussions with Rakesh Agrawal influenced the cost modeling aspects of this work. David DeWitt and Toby Lehman made suggestions that helped to improve the presentation.

REFERENCES

[Agra83a] Agrawal, R., and DeWitt, D., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", Technical Report No. 497, Computer Sciences Department, University of Wisconsin-Madison, February 1983.

[Agra83b] Agrawal, R., "Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation", Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1983.

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings of the COMPSAC '79 Conference, Chicago, Illinois, November 1979.

[Baye80] Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems 5(2), June 1980.

[Bern80] Bernstein, P., and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Technical Report, Computer Corporation of America, 1980.

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

[Brya80] Bryant, R., "SIMPAS -- A Simulation Language Based on PASCAL", Technical Report No. 390, Computer Sciences Department, University of Wisconsin, Madison, June 1980.

[Care83a] Carey, M., "Granularity Hierarchies in Concurrency Control", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March 1983.

[Care83b] Carey, M., "An Abstract Model of Database Concurrency Control Algorithms", Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1983.

[Care83c] Carey, M., "Modeling and Evaluation of Database Concurrency Control Algorithms", Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, September 1983.

[Care83d] Carey, M., "Multiple Versions and the Performance of Optimistic Concurrency Control", Technical Report No. 517, Computer Sciences Department, University of Wisconsin, Madison, October 1983.

[Casa79] Casanova, M., "The Concurrency Control Problem for Database Systems", Ph.D. Thesis, Computer Science Department, Harvard University, 1979.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February 1982.

[Chan82] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., "The Implementation of An Integrated Concurrency Control and Recovery Scheme", Proceedings of the ACM SIGMOD International Conference on Management of Data, March 1982.

[Chou83d] Chou, H., DeWitt, D., Katz, R., and Klug, A., "Design and Implementation of the Wisconsin Storage System", Technical Report No. 524, Computer Sciences Department, University of Wisconsin, Madison, November 1983.

[Date82] Date, C., An Introduction to Database Systems (Volume II), Addison-Wesley Publishing Company, 1982.

[Elli77] Ellis, C., "A Robust Algorithm for Updating Duplicate Databases", Proceedings of the 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.

[Gall82] Galler, B., "Concurrency Control Performance Issues" Ph.D. Thesis, Computer Science Department, University of Toronto, September 1982.

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

[Good83] Goodman, N., Suri, R., and Tay, Y., "A Simple Analytic Model for Performance of Exclusive Locking in Database Systems", Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, March 1983.

[Gray79] Gray, J., "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[Iran79] Irani, K., and Lin, H., "Queuing Network Models for Concurrent Transaction Processing in a Database System", Proceedings of the ACM SIGMOD International Symposium on Management of Data, 1979.

[Kort82] Korth, H., "Deadlock Freedom Using Edge Locks", ACM Transactions on Database Systems 7(4), December 1982.

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

[Lin82] Lin, W., and Nolte, J., "Distributed Database Control and Allocation: Semi-Annual Report", Technical Report, Computer Corporation of America, Cambridge, Massachusetts, January 1982.

[Lin83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking", Proceedings of the Ninth International Conference on Very Large Data Bases, Florence, Italy, November 1983.

[Lind79] Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Price, T., Putzolu, F., Traiger, I., and Wade, B., "Notes on Distributed Databases", Report No. RJ2571, IBM San Jose Research Laboratory, 1979.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

[Moha82] Mohan, C., Fussel, D., and Silberschatz, A., "Compatibility and Commutativity in Non-Two-Phase Locking Protocols", Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, California, March 1982.

[Poti80] Potier, D., and LeBlanc, P., "Analysis of Locking Policies in Database Management Systems", Communications of the ACM 23(10), October 1980.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Ries77] Ries, D., and Stonebraker, M., "Effects of Locking Granularity on Database Management System Performance", ACM Transactions on Database Systems 2(3), September 1977.

[Ries79a] Ries, D., "The Effects of Concurrency Control on Database Management System Performance", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1979.

[Ries79b] Ries, D., and Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems 4(2), June 1979.

[Robi82a] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1982.

[Robi82b] Robinson, J., "Separating Policy from Correctness in Concurrency Control Design", Report No. RC9308, IBM Thomas J. Watson Research Center, March 1982.

[Robi82c] Robinson, J., "Experiments with Transaction Processing on a Multi-Microprocessor", Report No. RC9725, IBM Thomas J. Watson Research Center, December 1982.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems 3(2), June 1978.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data", Proceedings of the Fourth Annual Symposium on the Simulation of Computer Systems, August 1976.

[Silb80] Silberschatz, A., and Kedem, Z., "Consistency in Hierarchical Database Systems", Journal of the ACM 27(1), January 1980.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering 5(3), May 1979.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 4(2), June 1979.

[Ullm83] Ullman, J., Principles of Database Systems, Second Edition, Computer Science Press, Rockville, Maryland, 1983.

[Wolf83] Wolff, R., personal communication.