MODELS FOR STUDYING CONCURRENCY CONTROL PERFORMANCE: ALTERNATIVES AND IMPLICATIONS

by

Rakesh Agrawal, Michael J. Carey and Miron Livny

Computer Sciences Technical Report #567

December 1984

Models for Studying Concurrency Control Performance: Alternatives and Implications

Rakesh Agrawal

AT&T Bell Laboratories
Summit, NJ 07901

Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award.

Models for Studying Concurrency Control Performance: Alternatives and Implications

Rakesh Agrawal

AT&T Bell Laboratories
Summit, NJ 07901

Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

A number of recent studies have examined the performance of concurrency control algorithms for database management systems. The results reported to date, rather than being definitive, have tended to be contradictory. In this paper, rather than presenting "yet another algorithm performance study", we critically investigate the assumptions made in the models used in past studies and their implications. We employ a "complete" model of a database environment to study the relative performance of three different approaches to the concurrency control problem under a variety of modeling assumptions. We show how differences in the underlying assumptions explain the seemingly contradictory performance results. We also examine how realistic the various assumptions would be for "real" database systems.

1. INTRODUCTION

Research in the area of concurrency control for database systems has led to the development of many concurrency control algorithms. Most of these algorithms are based on one of three basic mechanisms: *locking* [Mena78, Rose78, Gray79, Lind79, Ston79], *timestamps* [Reed78, Thom79, Bern80b], and *optimistic* concurrency control (also called commit-time validation or certification) [Bada79, Casa79, Kung81, Ceri82]. Bernstein and Goodman [Bern81, Bern82] survey many of the algorithms that have been developed and describe how new algorithms may be created by combining the three basic mechanisms.

Given the ever-growing number of available concurrency control algorithms, considerable research has recently been devoted to evaluating the performance of the various concurrency control algorithms. The behavior of locking has been investigated using both simulation [Spit76, Ries77, Ries79a, Ries79b, Lin82b] and analytical models [Iran79, Poti80, Gray81, Good83, Reut83, Thom83, Tay84a, Tay84b]. A qualitative

This research was partially supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award.

study that discussed performance issues for a number of distributed locking and timestamp algorithms was presented in [Bern80a], and an empirical comparison of several concurrency control schemes was given in [Pein83]. Recently, the performance of different concurrency control mechanisms have been compared in a number of simulation studies. The performance of locking was compared with the performance of basic timestamp ordering in [Gall82] and with basic and multiversion timestamp ordering in [Lin83]. Results of experiments comparing locking to the optimistic method appeared in [Robi82a, Robi82b], and the performance of several variants of locking, basic timestamp ordering, and the optimistic method were compared in [Care83, Care84]. Finally, the performance of several integrated concurrency control and recovery algorithms were evaluated in [Agra83a, Agra83b].

These performance studies are informative, but the results that have emerged, instead of being definitive, have been very contradictory. For example, studies by Carey and Stonebraker [Care84] and Agrawal and DeWitt [Agra83a] suggest that an algorithm that uses blocking instead of restarts is preferable from a performance viewpoint, but Tay [Tay84a, Tay84b] suggests that restarts lead to better performance than blocking. Results reported in [Gall82] regarding locking versus basic timestamp ordering contradict those of [Lin83]. Optimistic methods outperformed locking in [Fran83], whereas the opposite results were reported in [Agra83b, Care83].

Each of the previous studies employed a different performance model, and several of the studies were based on assumptions that have no clear physical meaning. In this paper, instead of presenting "yet another performance study", we address the assumptions made in the models used in past studies and their implications. We begin by establishing a framework based on a *complete* and (we believe) "realistic" model of a database management system. Our model captures all the main elements of a database environment, including both *users* (i.e., terminals, the source of transactions) and *physical resources* for storing and processing the data (i.e., disks and CPUs) in addition to the usual model components (workload and database characteristics). Based on this framework, we show how differences in assumptions explain the apparently contradictory performance results from the previous studies. We also examine what assumptions are reasonable for

real systems, and how more realistic assumptions would have altered the conclusions of several of the earlier studies.

In particular, we critically examine the common assumption of *infinite resources*. A number of analytical studies (for example, [Fran83, Tay84a, Ta784b]) and simulation studies (for example, [Lin82b, Lin83]) compare concurrency control algorithms under the assumption that transactions progress at a rate independent of the number of concurrent transactions. In other words, they proceed in *parallel* rather than in an interleaved manner. This is only really possible in a system with enough resources so that transactions *never* have to wait for CPU or I/O service — hence our choice of the phrase "infinite resources". We will investigate this assumption by performing studies with truly infinite resources, with multiple CPU-I/O devices, and with transactions that think while holding locks. The infinite resource case represents an "ideal" system, the multiple CPU-I/O device case models a class of multiprocessor database machines, and having transactions think while executing models an interactive workload.

We examine three concurrency control algorithms in this study, two locking algorithms and an optimistic algorithm, which differ as to when and how they detect and resolve conflicts. Section 2 describes our choice of concurrency control algorithms. We use a simulator based on a closed queuing model of a single-site database system for our performance studies. The structure and characteristics of this simulator are described in Section 3. Section 4 presents the performance experiments and our results. In Section 5 we summarize the main conclusions of our study.

2. CONCURRENCY CONTROL STRATEGIES

A transaction T is a sequence of actions $\{a_1, a_2, \ldots, a_n\}$, where a_i is either read or write. Given a concurrent execution of transactions, action a_i of transaction T_i and action a_j of T_j *conflict* if (i) a_i is read and a_j is write; or (ii) a_i is write and a_j is read or write. The various concurrency control algorithms basically differ in the time when they *detect conflicts* and the way that they *resolve conflicts* [Bern81]. For this study we have chosen to examine three concurrency control algorithms that represent extremes in conflict detection and resolution:

Blocking. Transactions set read locks on objects that they read, and these locks are later upgraded to write locks for objects which they also write. If a lock request is denied, the requesting transaction is blocked. A waits-for graph of transactions is maintained [Gray79], and deadlock detection is performed each time a transaction blocks. If a deadlock is discovered, the youngest transaction in the deadlock cycle is chosen as the victim and restarted. Dynamic two-phase locking [Gray79] is an example of this strategy.

Immediate-Restart. As in the case of blocking, transactions read-lock the objects that they read, and they later upgrade these locks to write locks for objects which they also write. However, if a lock request is denied, the requesting transaction is aborted and restarted after a restart delay. The delay period, which should be on the order of the expected response time of a transaction, prevents the same conflict from re-occurring repeatedly. A concurrency control strategy similar to this was considered in [Tay84a, Tay84b].

Optimistic. Transactions are allowed to execute unhindered and are validated only after they have reached their commit points. A transaction is restarted at its commit point if it finds that any object that it read has been written by another transaction which committed during its lifetime. The optimistic method proposed by Kung and Robinson [Kung81] is based on this strategy.

These algorithms represent two extremes with respect to when conflicts are detected. The blocking and immediate-restart algorithms are based on dynamic locking, so conflicts are detected as they occur. The optimistic algorithm, on the other hand, does not detect conflicts until transaction commit time. The three algorithms also represent two different extremes with respect to conflict resolution. The blocking algorithm blocks transactions to resolve conflicts, restarting them only when necessary because of a deadlock. The immediate-restart and optimistic algorithms always use restarts to resolve conflicts.

One final note in regard to the three algorithms: In the immediate-restart algorithm, a restarted transaction must be delayed for some time to allow the conflicting transaction to complete; otherwise, the same lock conflict will occur repeatedly. For the optimistic algorithm, it is unnecessary to delay the restarted transaction, as any detected conflict is with an already committed transaction. A restart delay is also unnecessary for the blocking algorithm, as the same deadlock cannot arise repeatedly.

3. SIMULATION MODEL

Central to our simulator for studying concurrency control algorithm performance is the closed queuing model of a single-site database system shown in Figure 1. This model is an extended version of the model used in [Care83, Care84], which in turn had its origins in the model of [Ries77, Ries79a, Ries79b]. There are a fixed number of terminals from which transactions originate. There is a limit to the number of transactions allowed to be active at any time in the system, the multiprogramming level *mpl*. A transaction is considered active if it is either receiving service or waiting for service inside the database system. When a new transaction originates, if the system already has a full set of active transactions, it enters the *ready queue* where it waits for a currently active transaction to complete or abort (transactions in the ready queue are not considered active). The transaction then enters the *cc queue* (concurrency control queue) and makes the first of its concurrency control requests. If the concurrency control 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. When the next concurrency control request is required, the transaction re-enters the concurrency control queue and makes the request. It is assumed for modeling convenience that a transaction performs all of its reads before performing any writes. In one of the performance studies later in the paper, we examine the performance of concurrency control algorithms under interactive workloads. The think path in the model provides an optional random delay that follows object accesses for this purpose. More will be said about modeling interactive transactions shortly.

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 decision to restart the transaction, it goes to the back of the ready queue, possibly after a randomly determined restart delay period of mean *restart_delay* (as in the immediate-restart algorithm). It then begins making all of the *same* concurrency control requests and object accesses over again.[1] 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

[1] The simulator maintains backup copies of transaction read and write sets

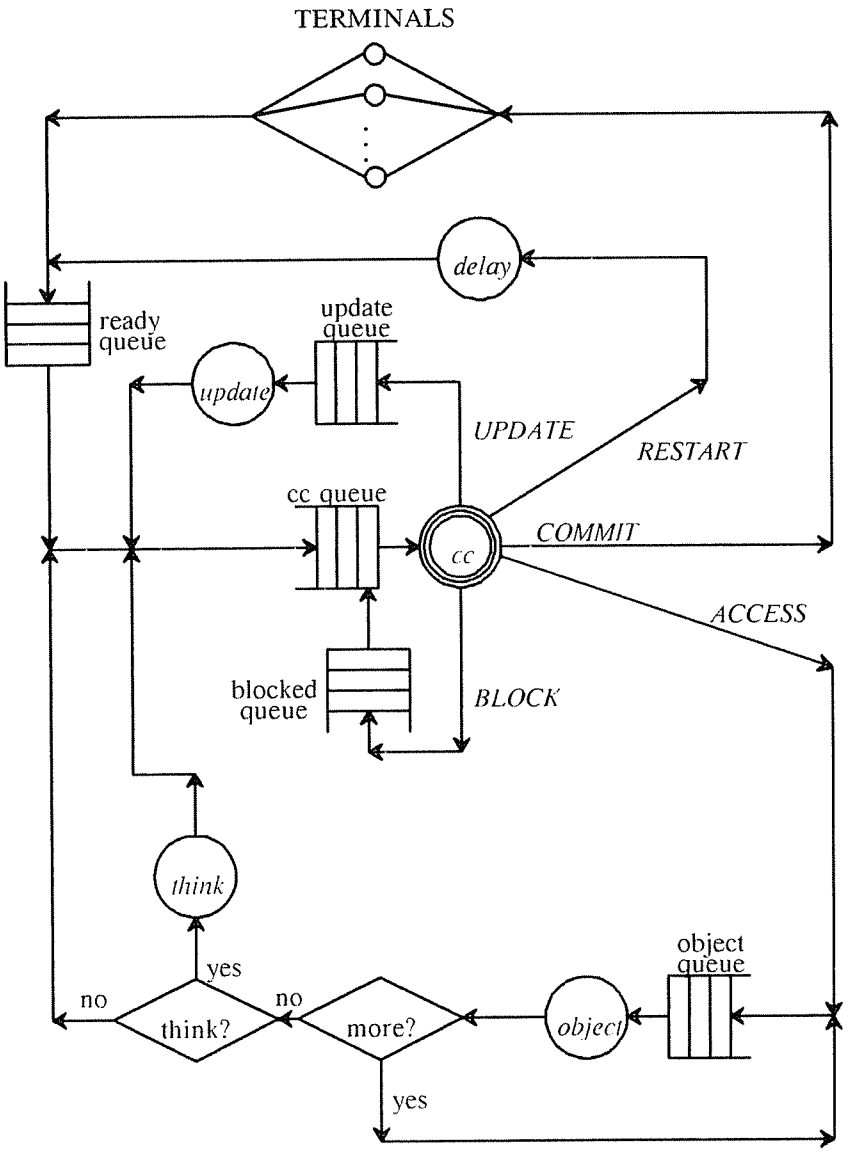

Figure 1: Logical Queuing Model.

has written one or more objects during its execution, however, it first enters the *update queue* and writes its deferred updates into the database. Deferred updates are assumed here because our simulation framework is intended to support *any* concurrency control algorithm — all algorithms operate correctly with deferred updates, but not all algorithms work with recovery schemes that do in-place updates.

To further illustrate how transactions flow through the model, we briefly describe how the locking algorithms and the optimistic algorithm are modeled. For locking, each concurrency control request corresponds

to a lock request for an object, and these requests alternate with object accesses. Locks are released together at end-of-transaction (after the deferred updates have been performed). For optimistic concurrency control, the first concurrency control request is granted immediately (i.e., it is a "no-op"); all object accesses are then performed with no intervening concurrency control requests. Only after the last object access is finished does a transaction return to the concurrency control queue in the optimistic case, at which time its validation test is performed (followed, if successful, by its deferred updates).

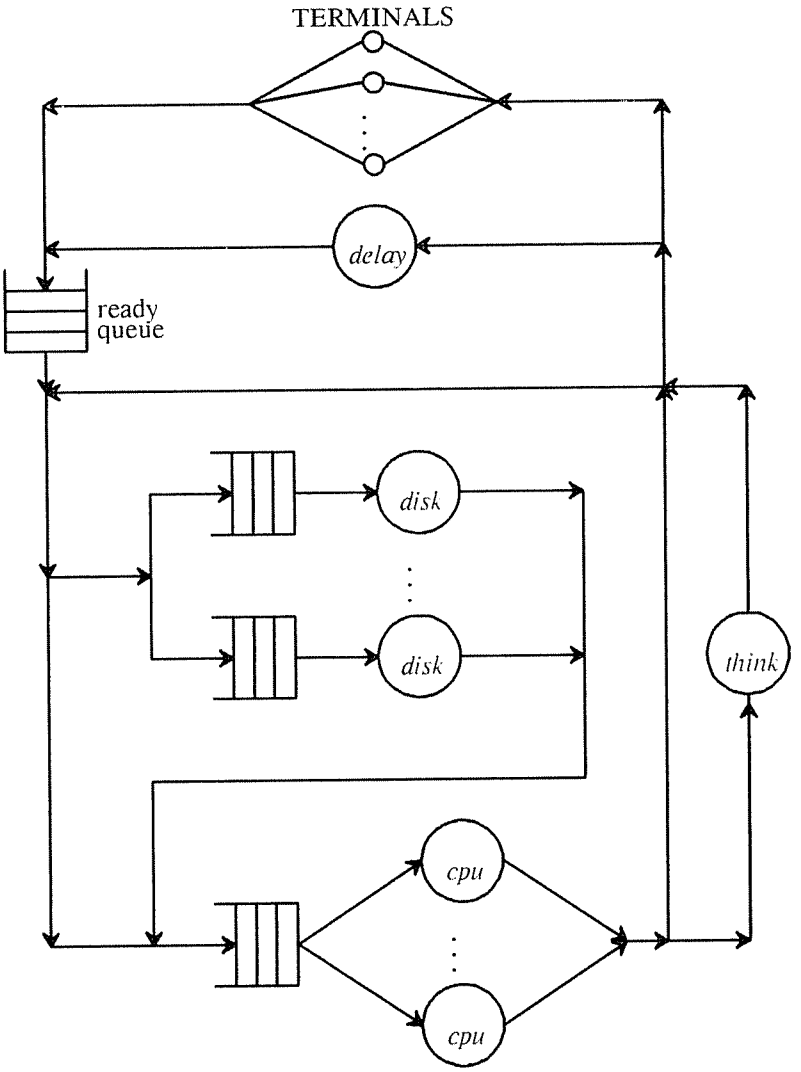

Figure 2: Physical Queuing Model.

Underlying the logical model of Figure 1 are two physical resources, the CPU and the I/O (i.e., disk) resources. Associated with the concurrency control, object access, and deferred update services in Figure 1 are some use of one or both of these two resources. The amounts of CPU and I/O time per logical service are specified as simulation parameters. The physical queuing model is depicted in Figure 2, and Table 1 summarizes its associated simulation parameters. As shown, the physical model is a collection of terminals, multiple CPU servers, and multiple I/O servers. The delay paths for the think and restart delays are also reflected in the physical queuing model. Simulation parameters specify the number of CPU servers, the number of I/O servers, and the number of terminals for the model. When a transaction needs CPU service, it is assigned a free CPU server; otherwise the transaction waits until one becomes free. Thus, the CPU servers may be thought of as being a pool of servers, all identical and serving one global CPU queue. Requests in the CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. Our I/O model is that of a partitioned database, where the data in the database is spread out across all of the disks. There is a queue associated with each of the I/O servers. When a transaction needs service, it chooses a disk (at random, with all disks being equally likely) and waits in an I/O queue associated with the selected disk. The service discipline for the I/O queues is also FCFS.

The parameters *obj_io* and *obj_cpu* are the amounts of I/O and CPU time 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. These parameters represent constant service time requirements rather than stochastic ones for simplicity. The *ext_think_time* parameter is the mean of an exponential time distribution which determines the time delay between the completion of a transaction and the initiation of a new transaction from a terminal. Finally, the *int_think_time* parameter is the mean of an exponential time distribution which determines the intra-transaction think time for the model (if any). To model interactive workloads, transactions can be made to undergo a thinking period between finishing their reads and starting their writes.

Parameter	Meaning
db_size	number of objects in database
tran_size	mean size of transaction
max_size	size of largest transaction
min_size	size of smallest transaction
write_prob	Pr(write X | read X)
restart_delay	mean transaction restart delay (optional)
num_terms	number of terminals
mpl	multiprogramming level
ext_think_time	mean time between transactions (per terminal)
int_think_time	mean intra-transaction think time (optional)
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	number of cpus
num_disks	number of disks

Table 1: Simulation Parameters.

A transaction is modeled according to the number of objects that it reads and writes. The parameter *tran_size* is the average number of objects read by a transaction, the mean of a uniform distribution between *min_size* and *max_size* (inclusive). These objects are randomly chosen (without replacement) from among all of the objects in the database. The probability that an object read by a transaction will also be written is determined by the parameter *write_prob*. The size of the database is assumed to be *db_size*.

4. PERFORMANCE EXPERIMENTS

We performed a number of simulation experiments to study the implications of different assumptions on the performance of the three concurrency control algorithms described in Section 3. We first examined the performance of the three strategies in the case of very low conflicts. We then investigated the performance under the infinite resources assumption, with limited resources, and with multiple CPUs and disks. Last, we examined the case of an interactive workload.

Table 2 gives the simulation parameter values used for the experiments reported here (except where otherwise noted). The parameters that vary from experiment to experiment are not listed in the table, but will instead be given in the description of the experiments. The database and transaction sizes were selected so as to jointly yield a region of operation which allows the interesting performance effects to be observed without

Parameter	Value
db_size	1000 pages
tran_size	8 page readset
max_size	12 page readset (maximum)
min_size	4 page readset (minimum)
write_prob	0.25
num_terms	200 terminals
mpl	5, 10, 25, 50, 75, 100, and 200 transactions
ext_think_time	1 second
obj_io	35 milliseconds
obj_cpu	15 milliseconds

Table 2: Simulation Parameter Settings.

necessitating impossibly long simulation times. These sizes are expressed in pages, as we equate objects and pages in this study. The multiprogramming level is varied between a limit of 5 transactions and a limit of the total number of terminals, set to 200 in this study, to allow a range of conflict probabilities to be investigated. The object processing costs were chosen based on our notion of roughly what realistic values might be. We employed a modified form of the batch means method [Sarg76] for our statistical data analyses, and each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90% confidence intervals.[2] The actual batch time varied from experiment to experiment, but the throughput confidence intervals were typically in the range of plus or minus a few percent of the mean value, more than sufficient for our purposes. We discuss only the statistically significant performance differences when summarizing our results.

Throughout the paper we use a fixed set of symbols for representing the data points obtained from the three different concurrency control algorithms. These symbols are summarized at the top of each of the pages with graphs.

4.1. Experiment 1: Low Conflict Situation

For the first experiment, we used a larger database size of 10,000 objects. Due to the large database size and the relatively small transaction size, there were few conflicts in this experiment. larger The

[2] More information on the details of the modified batch means method may be found in [Care83].

throughput results for a system with infinite resources and a system with finite resources (1 CPU and 2 disks) are shown in Figures 3 and 4 respectively. The performance of the three concurrency control strategies was close in both cases, confirming the results in [Care83, Care84, Agra83a, Agra83b] — if conflicts are rare, it makes little difference which concurrency control algorithm is used. In both cases, blocking outperformed the other two algorithms by a small amount.

Since we were interested in investigating *differences* in concurrency control strategies, we decreased the database size to 1000 objects (as shown in Table 2) to create a situation where conflicts are more frequent. The rest of the experiments were performed using this database size.

4.2. Experiment 2: Infinite Resources

The next experiment examined the performance characteristics of the three strategies assuming infinite resources for a variety of multiprogramming levels. With infinite resources, as the multiprogramming level is increased, the throughput should also increase in the absence of data contention. However, for a given size database, the probability of conflicts increases as the multiprogramming level increases. For blocking, the increased conflict probability will manifest itself in the form of more blocking due to denial of lock requests and an increased number of restarts due to deadlocks. For the restart-oriented strategies, the higher probability of conflicts will result in a larger number of restarts.

Figure 5 shows the throughput results for Experiment 2. Blocking starts thrashing as the multiprogramming level is increased beyond a certain level, whereas the throughput keeps increasing for the optimistic algorithm. These results agree with predictions in [Fran83] that were based on similar assumptions. Figure 6 shows the average number of times that a transaction was blocked and restarted per commit, called the block ratio (dotted line) and the restart ratio (solid lines), for the three concurrency control algorithms. Note that the thrashing in blocking is due to the large increase in the number of times that a transaction is blocked, which reduces the effective multiprogramming level, rather than to an increase in the number of restarts. This result is in agreement with the assertion in [Tay84a, Tay84b] that under low resource contention and a high level of multiprogramming, blocking may start thrashing before restarts do. Although the restart ratio for the optimistic algorithm increases quickly with an increase in the multiprogramming level, new

□ — blocking ○ — immediate-restart Δ — optimistic

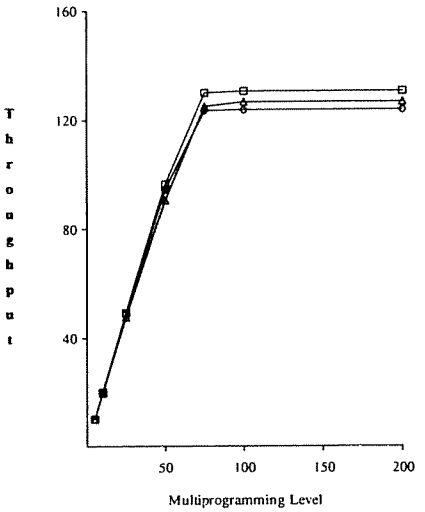

Figure 3: Throughput (Infinite Resources).

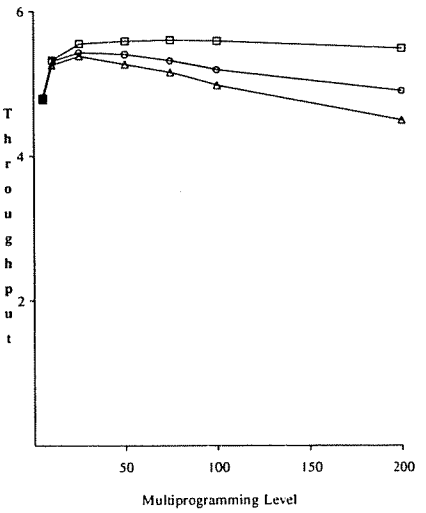

Figure 4: Throughput (1 CPU, 2 Disks).

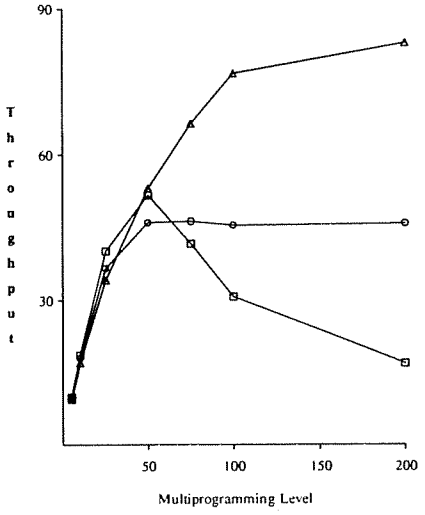

Figure 5: Throughput (Infinite Resources).

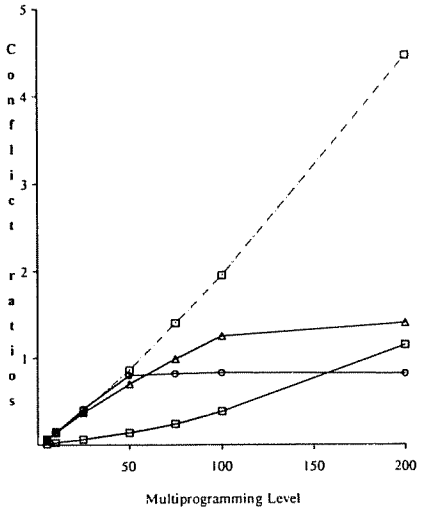

Figure 6: Conflict Ratios (Infinite Resources).

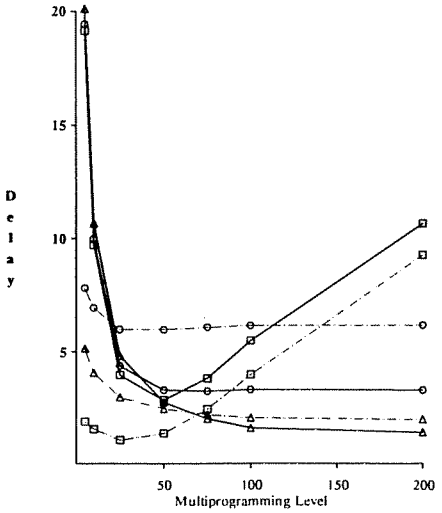

Figure 7: Response Time (Infinite Resources).

transactions start executing in place of the restarted ones, keeping the effective multiprogramming level high and thus entailing an increase in throughput.

Unlike the other two algorithms, the throughput of the immediate-restart algorithm reaches a plateau. This happens for the following reason: When a transaction is restarted in the immediate-restart strategy, a restart delay is invoked to allow the conflicting transaction to complete before the restarted transaction is placed back in the ready queue. The duration of the restart delay is exponential with a mean equal to the running average of the transaction response time — that is, the duration of the delay is *adaptive*, depending on the observed average response time. We chose to employ an adaptive delay after performing a sensitivity analysis that showed us that the performance of immediate-restarts is sensitive to the restart delay time, particularly in the infinite resource case. Our experiments indicated that a delay of about one transaction time is best, and that throughput begins to drop off rapidly when the delay exceeds more than a few transaction times. Because of this adaptive delay, then, the immediate-restart algorithm reaches a state where increasing the multiprogramming level does not result in an actual increase in the number of active transactions — there are no transactions waiting in the ready queue, so increasing the allowed population has no effect. All of the non-active transactions are either in a terminal thinking state or a restart delay state.

Figure 7 shows the mean response time (solid lines) and the standard deviation of response time (dotted lines) for each of the three algorithms. The response times are basically what one would expect, given the throughput results plus the fact that we have employed a closed queuing model. This figure does illustrate one interesting phenomenon that occurred in nearly all of the experiments reported in this paper: The standard deviation of the response time is smaller for blocking than for the immediate-restart algorithm over most of the multiprogramming levels explored — the response time variance for the immediate-restart algorithm is quite significant. A high variance in response time is undesirable from a user's standpoint.

4.3. Experiment 3: Resource-Limited Situation

In Experiment 3 we analyzed the impact of finite resources on the performance characteristics of the three concurrency control algorithms. A database system with 1 CPU and 2 disks was assumed for this experiment. The throughput results are presented in Figure 8.

Observe that for all three algorithms, as the multiprogramming level is increased, the throughput first increases, then reaches a peak, and then finally either decreases or remains roughly constant. In a system with finite CPU and I/O resources, the achievable throughput may be constrained by one or more of the following factors: It may be that not enough transactions are available to keep the system resources busy. Alternatively, it may be that enough transactions are available, but because of data contention, the "useful" number of transactions is less than what is required to keep the resources "usefully" busy. That is, transactions that are blocked due to lock conflicts are not useful; similarly, the use of resources to process transactions that are later restarted is not useful. Finally, it may be that enough useful, non-conflicting transactions are available, but that the available resources are already saturated.

As the multiprogramming level was increased, the throughput first increased for all three concurrency control algorithms, as there were not enough transactions to keep the resources utilized at low levels of multiprogramming. Figure 9 shows the total (solid lines) and useful (dotted lines) disk utilizations for this experiment. The useful utilizations indicate the fraction of the resources used to do work that actually completed (i.e., they exclude the fraction used for work that was later undone by restarts). The utilization of the disks is selected here because the disks are the bottleneck resource with our parameter settings. For blocking, the throughput peaks at $mpl = 25$, where the disks are being 97.2% utilized, with a useful utilization of 92.1%. Increasing the multiprogramming level further only increases data contention, and the throughput decreases as the amount of blocking and the number of restarts increase at a much faster rate. For the optimistic algorithm, the useful utilization of the disks peaks at $mpl = 10$, and the throughput decreases with an increase in the multiprogramming level because of the increase in the restart ratio. This restart ratio increase means that a larger fraction of the disk time is spent on processing objects that will be redone later. For the immediate-restart algorithm, the throughput also peaks at $mpl = 10$ and then decreases, remaining roughly constant beyond 50. The thoughput remains constant for this algorithm for the same reason as described in the last experiment — increasing the allowable number of transactions has no effect beyond 50, as all of the non-active transactions are either thinking or in a restart delay state.

With regards to the throughput for the three strategies, several observations are in order. First, the maximum throughput (i.e., the best global throughput) was obtained with the blocking algorithm. Second, immediate-restarts performed as well as or better than the optimistic algorithm. There were more restarts with the optimistic algorithm, and each restart was more expensive; this is reflected in the relative useful disk utilizations for the two strategies. Finally, the throughput achieved with the immediate-restart strategy for *mpl* = 200 was somewhat better than the throughput achieved with either blocking or with the optimistic algorithm.

Figure 10 gives the average and the standard deviation of response time for the three algorithms in the finite resource case. The differences are even more noticeable than in the infinite case. Blocking has the lowest response time over most of the multiprogramming levels, and it has the lowest response time globally. The immediate-restart algorithm is next, and the optimistic algorithm has the worst response time. As for the standard deviations, blocking is the best, immediate-restarts is the worst, and the optimistic algorithm is in between the two. As in Experiment 1, the immediate-restart algorithm exhibits a very high response time variance.

One of the points raised earlier merits further discussion. Should the performance of the immediate-restart algorithm at *mpl* = 200 lead us to conclude that immediate-restart is a better strategy at high levels of multiprogramming? We believe that the answer is no, for several reasons. First, the multiprogramming level is internal to the database system, controlling the number of transactions that may concurrently compete for data and resources, and has nothing to do with the number of users that the database system may support; the latter is determined by the number of terminals. Thus, one should configure the system to control the multiprogramming at a level which gives the best performance. In our experiment, the highest throughput and smallest standard deviation of response time were achieved using the blocking algorithm at *mpl* = 50. Second, the restart delay in the immediate-restart strategy is there so that the conflicting transaction can complete before the restarted transaction is placed back into the ready queue. However, an unintended side effect of this restart delay in a system with a finite number of users is that it limits the actual multiprogramming level, and hence also limits the number of conflicts and resulting restarts due to reduced data contention.

□ — blocking ○ — immediate-restart Δ — optimistic

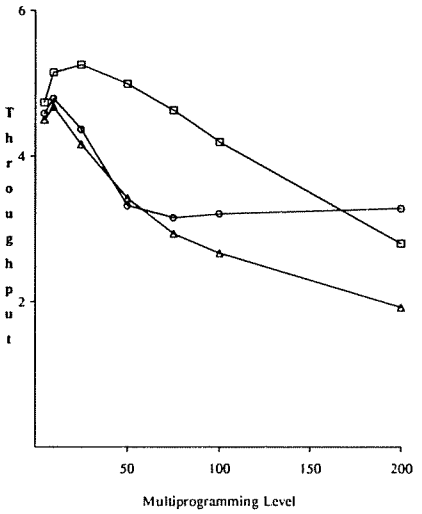

Figure 8: Throughput (1 CPU, 2 Disks).

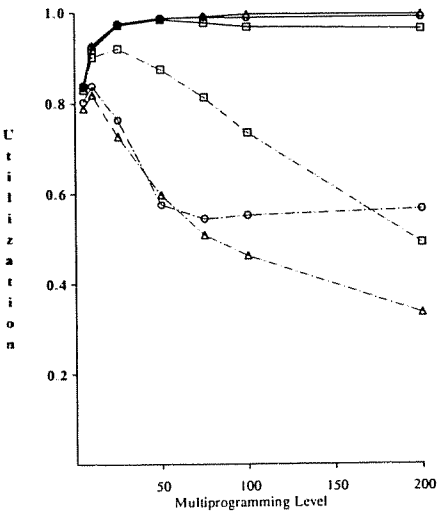

Figure 9: Disk Utilization (1 CPU, 2 Disks).

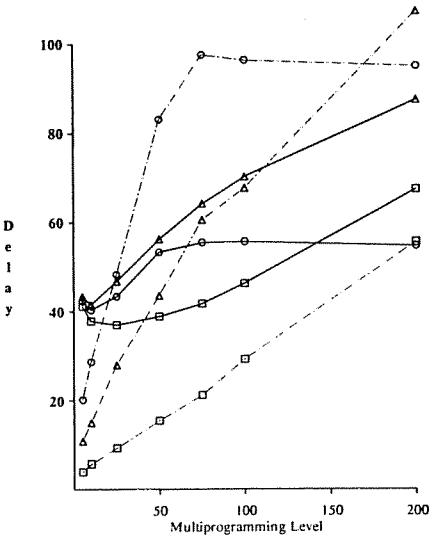

Figure 10: Response Time (1 CPU, 2 Disks).

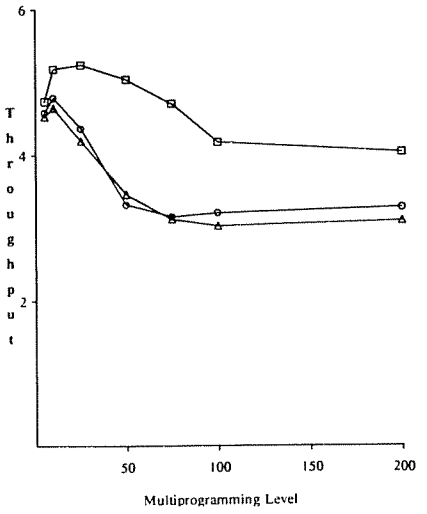

Figure 11: Throughput (Adaptive Delays).

Although the multiprogramming level was increased to the total number of users (200), the actual average multiprogramming level never exceeded about 60. Thus, the restart delay provides a crude mechanism for limiting the multiprogramming level when restarts become overly frequent, and adding a restart delay to the other two algorithms should improve their performance at high levels of multiprogramming as well.

To verify this latter argument, we performed another experiment where the adaptive restart delay was used for restarted transactions in both the blocking and optimistic algorithms as well. The throughput results that we obtained are shown in Figure 11. It can be seen that introducing an adaptive restart delay helped to limit the multiprogramming level for the blocking and optimistic algorithms under high conflicts, as it does for immediate-restarts, reducing data contention at the upper end of the curves. Blocking emerges as the clear winner, and the performance of the optimistic algorithm becomes comparable to the immediate-restart strategy. The one negative effect that we observed from adding this delay was an increase in the standard deviation of the response times for the blocking and optimistic algorithms. Since a restart delay only helps performance for high multiprogramming levels, it seems that a better strategy is to enforce a lower multiprogramming level limit to avoid thrashing due to high contention and to maintain a small standard deviation of response time.

4.4. Experiment 4: Multiple Resources

In this experiment we moved the system from finite resources towards infinite resources. We increased the number of resources available to 5 CPUs and 10 disks and then to 25 CPUs and 50 disks to determine where finite resources start behaving like infinite resources in a multiprocessor database machine environment.

For 5 CPUs and 10 disks, the behavior of the three concurrency control strategies was fairly similar to the behavior in the case of 1 CPU and 2 disks. The throughput results for this case are shown in Figure 12, and the disk utilization figures for this case are given in Figure 13. Blocking again provided the highest overall throughput. For large multiprogramming levels, however, the immediate-restart strategy provided better throughput than blocking, but not enough to beat the highest throughput provided by the blocking algorithm. In this resource configuration, the maximum useful utilizations of the disks with the blocking,

immediate-restarts, and the optimistic algorithm were 55.5%, 44.6%, and 46.6% respectively, whereas the maximum total disk utilizations were 61.8%, 72.6%, and 94.1%. Note that, due to restarts, the total utilizations for the restart-oriented algorithms are higher than those for blocking; the difference is partially due to wasted resources. By "wasted resources" here, we mean resources used to process objects that were later undone due to restarts — these resources are wasted in the sense that they were consumed, making them unavailable for other purposes such as background tasks.

For 25 CPUs and 50 disks, the maximum throughput obtained with the optimistic algorithm beats the maximum throughput obtained with blocking (although not by very much). The throughput results for this case are shown in Figure 14, and the utilizations are given in Figure 15. The total and the useful disk utilizations for the maximum throughput point for blocking were 33.5% and 30.1% (respectively) whereas the corresponding numbers for the optimistic algorithm were 62.6% and 32.6%. Thus, the optimistic algorithm has become attractive because a large amount of otherwise unused resources are available, and thus the resources wasted due to restarts does not adversely affect performance. In other words, with useful utilizations in the 30% range, the system begins to behave somewhat like it has infinite resources.

Another interesting observation from these results is that, with blocking, resource utilization decreases as the level of multiprogramming increases and hence throughput decreases. This is a further indication that blocking may thrash due to waiting for locks before it thrashes due to the number of restarts [Tay84a, Tay84b], as we saw in the infinite resource case. On the other hand, with the optimistic algorithm, as the multiprogramming level increases, the total utilization of resources and resource waste increases, and the throughput decreases somewhat. Thus, this strategy eventually thrashes due to the number of restarts (i.e., because of resources). With immediate-restarts, as explained earlier, a plateau is reached for throughput and resource utilization because the actual multiprogramming level is limited by the restart delay under high data contention.

4.5. Experiment 5: Interactive Workloads

In our last experiment, we modeled interactive transactions that perform a number of reads, think for some period of time, and then perform their writes. This model of interactive transactions was motivated by a

□ — blocking ○ — immediate-restart Δ — optimistic

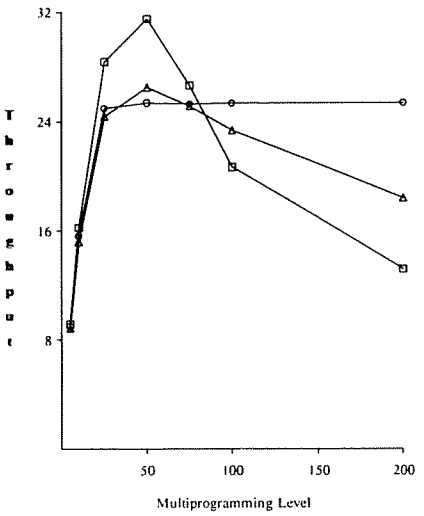

Figure 12: Throughput (5 CPUs, 10 Disks).

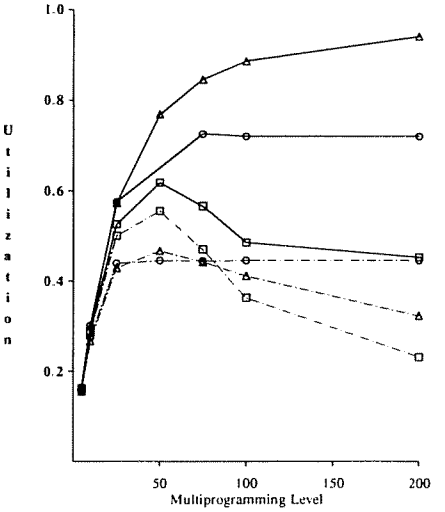

Figure 13: Disk Utilization (5 CPUs, 10 Disks).

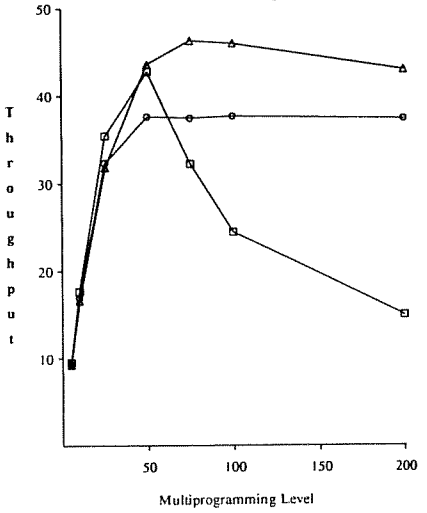

Figure 14: Throughput (25 CPUs, 50 Disks).

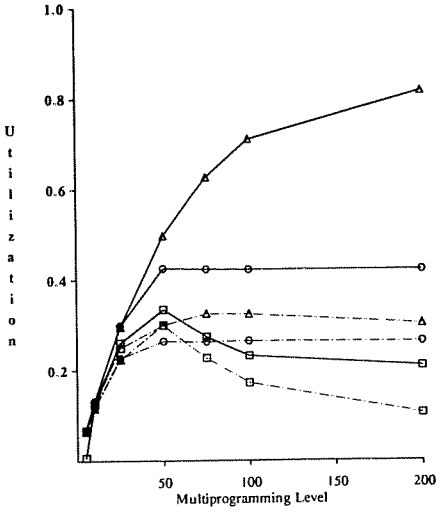

Figure 15: Disk Utilization (25 CPUs, 50 Disks).

large body of form-screen applications where data is put up on the screen, the user may change some of the fields after staring at the screen for a while, and then the user types "enter" causing the updates to be performed. The intent of this experiment was to find out whether large intra-transaction — internal — think times can cause a system with finite resources to behave like it has infinite resources. The interactive workload experiment was performed for internal think times of 1, 5, and 10 seconds. At the same time, the external think times were increased to 3, 11, and 21 seconds (respectively) in order to keep roughly the same ratio of thinking transactions to active transactions. We have assumed a finite resource environment with 1 CPU and 2 disks for the system in this experiment.

Figure pairs (16, 17), (18, 19), and (20, 21) show the throughput and disk utilizations obtained for the 1, 5, and 10 second inter-transaction think time experiments (respectively). On the average, a transaction requires 150 milliseconds of CPU time and 350 milliseconds of disk time, so internal think times of 1, 5, and 10 seconds thus considerably increase the duration for which a locks are held. This causes the resource utilization for blocking to decrease as the internal think time increases. With the optimistic algorithm, the demand for resources is also reduced due to large think times, and the resources start behaving as though they were infinite resources. Consequently, for large think times, the optimistic algorithm performs better than the blocking strategy (see Figures 18 and 20). For an internal think time of 21 seconds, the useful utilization of resources is much higher with the optimistic algorithm than the blocking strategy, and the highest throughput value is also considerably higher. For 5 seconds of internal think time, the throughput and the useful utilization with the optimistic algorithm is also better than for blocking. For a 1 second internal think time, however, blocking performs better (see Figure 16). The resource utilizations here are such that wasted resources due to restarts makes the optimistic algorithm the loser.

The highest throughput obtained with the optimistic algorithm was consistently better than that for immediate-restarts, although for higher levels of multiprogramming the throughput obtained with immediate-restarts was better than the throughput obtained with the optimistic algorithm due to the multiprogramming-limiting effect of immediate-restart's restart delay. As noted before, this high multiprogramming level difference could be reversed by adding a restart delay to the optimistic algorithm.

□ — blocking ○ — immediate-restart Δ — optimistic

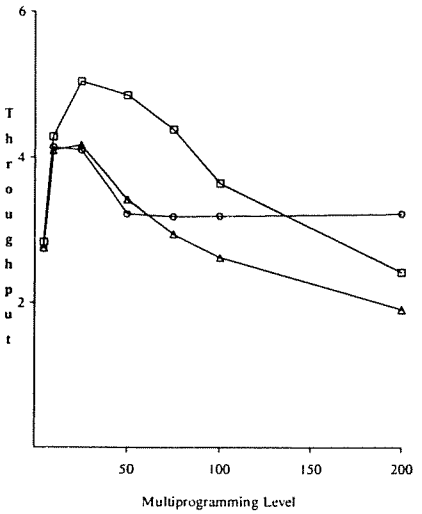

Figure 16: Throughput (1 Second Internal Thinking).

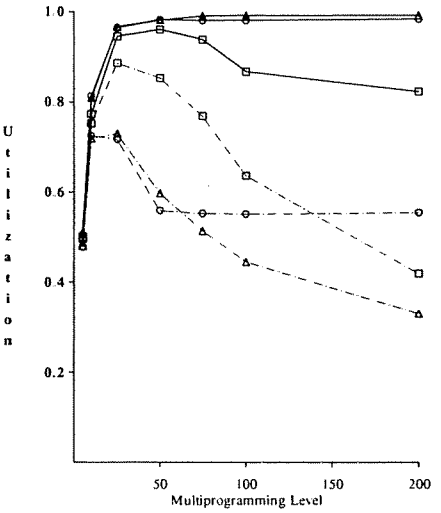

Figure 17: Disk Utilization (1 Second Internal Thinking).

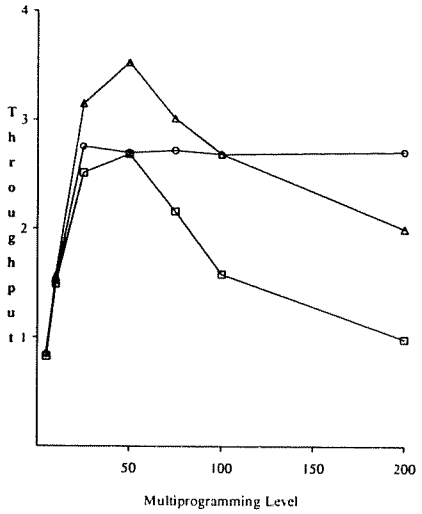

Figure 18: Throughput (5 Seconds Internal Thinking).

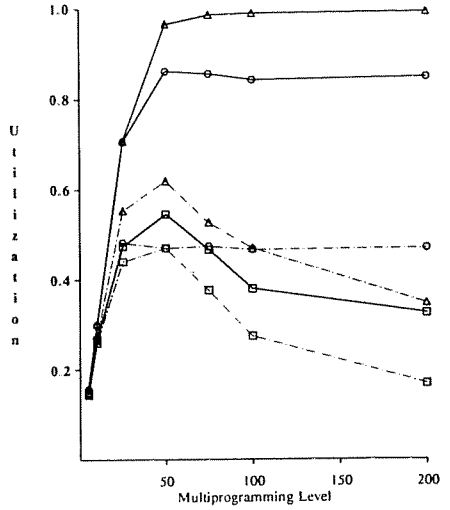

Figure 19: Disk Utilization (5 Seconds Internal Thinking).

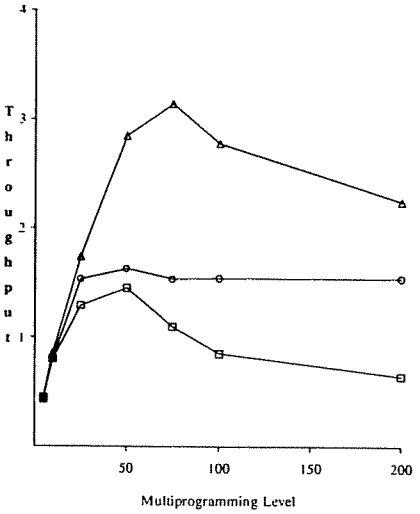

Figure 20: Throughput (10 Seconds Internal Thinking).

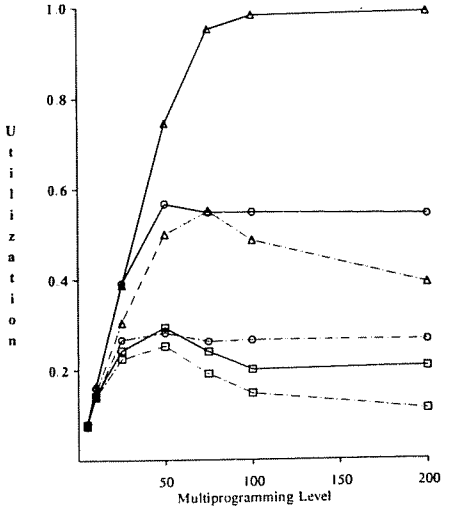

Figure 21: Disk Utilization (10 Seconds Internal Thinking).

5. CONCLUSIONS AND IMPLICATIONS

One major conclusion of this study is that for medium to high levels of resource utilization, a blocking algorithm like dynamic two-phase locking is a better choice than a restart-oriented concurrency control like the immediate-restart or optimistic algorithms. However, if resource utilizations are sufficiently low that a large amount of wasted resources can be tolerated, and in addition there are a large number of transactions available to execute, then a restart-oriented algorithm is a better choice. We found the optimistic algorithm to be the best choice under these conditions. Such low resource utilizations could arise in a database machine with a large number of CPUs and disks and a number of terminals similar to what one finds on typical timesharing systems today. They could also arise in primarily interactive applications where very large think times occur and the number of terminals is such that the utilization of the system is low as a result. It is an open question whether or not such utilizations could really occur in real systems (i.e., whether it will ever be cost effective to run at such low utilizations). If not, blocking algorithms will remain the preferred method for database concurrency control.

Another result of this study is that we have re-confirmed the results of a number of other studies, including those reported in [Agra83a, Agra83b, Care83, Care84, Fran83, Tay84a, Tay84b]. We have shown that seemingly contradictory results, some of which favor blocking algorithms and others of which favor restarts, are not contradictory at all. The studies are all correct within the limits of their assumptions, particularly their assumptions about system resources. While it is possible to study the effects of data contention and resource contention separately in some models [Tay84a, Tay84b], it appears not to be worth doing so if the goal is to select a concurrency control algorithm for a real system — the proper algorithm choice is resource-dependent.

An interesting side result of this study is that the level of multiprogramming in database systems should be carefully controlled. We refer here to the multiprogramming level internal to the database system, controlling the number of transactions that may concurrently compete for data, CPU, and I/O services. As in the case of paging operating systems, if the multiprogramming level is increased beyond a certain level, the blocking and optimistic concurrency control strategies start thrashing. We have confirmed the results of

[Fran83, Tay84a, Tay84b] for locking in the low resource contention case, but more importantly we have also seen that the effect can be significant for both locking and optimistic concurrency control under higher levels of resource contention. We found that when we delayed restarted transactions by an amount equal to the running average response time, it had the beneficial side effect of limiting the actual multiprogramming level, and the degradation in throughput was arrested (albeit a little bit late). Since the use of a restart delay to limit the multiprogramming level is at best a crude strategy, adaptive algorithms that dynamically adjust the multiprogramming level in order to maximize system throughput need to be designed. Some performance indicators that might be used in the design of such an algorithm are useful resource utilization, running averages of throughput or response time, etc. The design of such adaptive algorithms is an open problem.

In closing, we wish to leave the reader with the following thoughts about the future, due to Bill Wulf [Wulf81]:

> *"Although the hardware costs will continue to fall dramatically and machine speeds will increase equally dramatically, we must assume that our aspirations will rise even more. Because of this, we are not about to face either a cycle or memory surplus. For the near-term future, the dominant effect will not be machine cost or speed alone, but rather a continuing attempt to increase the return from a finite resource — that is, a particular computer at our disposal."*

ACKNOWLEDGEMENTS

The authors wish to acknowledge helpful discussions that one or more of us have had with Mary Vernon, Nat Goodman, and (especially) Y.C. Tay. Also, the NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for this study.

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.

[Bern80a] Bernstein, P., and Goodman, N., *Fundamental Algorithms for Concurrency Control in Distributed Database Systems*, Technical Report, Computer Corporation of America, 1980.

[Bern80b] Bernstein, P., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", *Proceedings of the Sixth International Conference on Very Large Data Bases*, October 1980.

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

[Bern82] Bernstein, P., and Goodman, N., "A Sophisticate's Introduction to Distributed Database Concurrency Control", *Proceedings of the Eighth International Conference on Very Large Data Bases*, September 1982.

[Care83] Carey, M., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, September 1983.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore, August 1984.

[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.

[Fran83] Franaszek, P., and Robinson, J., *Limitations of Concurrency in Transaction Processing*, Report No. RC10151, IBM Thomas J. Watson Research Center, August 1983.

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

[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.

[Gray81] Gray, J., Homan, P., Korth, H., and Obermarck, R., *A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System*, Technical Report No. RJ3066, IBM San Jose Research Laboratory, February 1981.

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

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

[Lin82a] Lin, W., and Nolte, J., *Distributed Database Control and Allocation: Semi-Annual Report*, Technical Report, Computer Corporation of America, Cambridge, Massachusetts, January 1982.

[Lin82b] Lin, W., and Nolte, J., "Performance of Two Phase Locking", *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February 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., et al, *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.

[Pein83] Peinl, P., and Reuter, A., "Empirical Comparison of Database Concurrency Control Schemes", *Proceedings of the Ninth International Conference on Very Large Databases*, 1983.

[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.

[Reut83] Reuter, A., *An Analytic Model of Transaction Interference in Database Systems*, IB 68/83, University of Kaiserslautern, West Germany, 1983.

[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., *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.

[Spit76] Spitzer, J., "Performance Prototyping of Data Management Applications", *Proceedings of the ACM '76 Annual Conference*, October 1976.

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

[Tay84a] Tay, Y., *A Mean Value Performance Model for Locking in Databases*, Ph.D. Thesis, Computer Science Department, Harvard University, February 1984.

[Tay84b] Tay, Y., and Suri, R., "Choice and Performance in Locking for Databases", *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore, August 1984.

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

[Thom83] Thomasian, A, and Ryu, I., "A Decomposition Solution to the Queuing Network Model of the Centralized DBMS with Static Locking", *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Minneapolis, MN, August 1983.

[Wulf81] Wulf, W., "Compilers and Computer Architecture", *IEEE Computer*, July 1981.