Concurrency Control Performance Modeling: Alternatives and Implications

RAKESH AGRAWAL
AT&T Bell Laboratories
MICHAEL J. CAREY and MIRON LIVNY
University of Wisconsin

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 fairly complete model of a database environment for studying the relative performance of three different approaches to the concurrency control problem under a variety of modeling assumptions. The three approaches studied represent different extremes in how transaction conflicts are dealt with, and the assumptions addressed pertain to the nature of the database system's resources, how transaction restarts are modeled, and the amount of information available to the concurrency control algorithm about transactions' reference strings. We show that differences in the underlying assumptions explain the seemingly contradictory performance results. We also address the question of how realistic the various assumptions are for actual database systems.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*transaction processing*; D.4.8 [**Operating Systems**]: Performance—*simulation, modeling and prediction*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Concurrency control

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* [23, 31, 32, 44, 48], *timestamps* [8, 36, 52], and *optimistic* concurrency control (also called commit-time validation or certification) [5, 16, 17, 27]. Bernstein and Goodman [9, 10] survey many of

A preliminary version of this paper appeared as "Models for Studying Concurrency Control Performance: Alternatives and Implications," in *Proceedings of the International Conference on Management of Data* (Austin, Tx., May 28–30, 1985).
M. J. Carey and M. Livny were partially supported by the Wisconsin Alumni Research Foundation under National Science Foundation grant DCR-8402818 and an IBM Faculty Development Award.
Authors' addresses: R. Agrawal, AT&T Bell Laboratories, Murray Hill, NJ 07974; M. J. Carey and M. Livny, Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

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 concurrency control algorithms. The behavior of locking has been investigated using both simulation [6, 28, 29, 39–41, 47] and analytical models [22, 24, 26, 35, 37, 50, 51, 53]. A qualitative study that discussed performance issues for a number of distributed locking and timestamp algorithms was presented in [7], and an empirical comparison of several concurrency control schemes was given in [34]. Recently, the performance of different concurrency control mechanisms has been compared in a number of studies. The performance of locking was compared with the performance of basic timestamp ordering in [21] and with basic and multiversion timestamp ordering in [30]. The performance of several alternatives for handling deadlock in locking algorithms was studied in [6]. Results of experiments comparing locking to the optimistic method appeared in [42 and 43], and the performance of several variants of locking, basic timestamp ordering, and the optimistic method was compared in [12 and 15]. Finally, the performance of several integrated concurrency control and recovery algorithms was evaluated in [1 and 2].

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 [15] and Agrawal and DeWitt [2] suggest that an algorithm that uses blocking instead of restarts is preferable from a performance viewpoint, but studies by Tay [50, 51] and Balter et al. [6] suggest that restarts lead to better performance than blocking. Optimistic methods outperformed locking in [20], whereas the opposite results were reported in [2 and 15]. In this paper, rather than presenting "yet another algorithm performance study," we examine the reasons for these apparent contradictions, addressing the models used in past studies and their implications.

The research that led to the development of the many currently available concurrency control algorithms was guided by the notion of *serializability* as the correctness criteria for general-purpose concurrency control algorithms [11, 19, 33]. Transactions are typically viewed as sequences of read and write requests, and the interleaved sequence of read and write requests for a concurrent execution of transactions is called the execution *log*. Proving algorithm correctness then amounts to proving that any log that can be generated using a particular concurrency control algorithm is equivalent to some serial log (i.e., one in which all requests from each individual transaction are adjacent in the log). Algorithm correctness work has therefore been guided by the existence of this widely accepted standard approach based on logs and serializability. Algorithm performance work has not been so fortunate—no analogous standard performance model has been available to guide the work in this area. As we will see shortly, the result is that nearly every study has been based on its own unique set of assumptions regarding database system resources, transaction behavior, and other such issues.

In this paper, we begin by establishing a performance evaluation framework based on a fairly complete model of a database management system. Our model

captures 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 characteristics of the workload and the database. On the basis of this framework, we then show that differences in assumptions explain the apparently contradictory performance results from previous studies. We examine the effects of alternative assumptions, and we briefly address the question of which alternatives seem most reasonable for use in studying the performance of database management systems.

In particular, we critically examine the common assumption of *infinite resources.* A number of studies (e.g., [20, 29, 30, 50, 51]) compare concurrency control algorithms under the assumption that transactions progress at a rate independent of the number of active 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 before receiving 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.

In addition to these resource-related assumptions, we examine two modeling assumptions related to transaction behavior that have varied from study to study. In each case, we investigate how alternative assumptions affect the performance results. One of the additional assumptions that we address is the *fake restart* assumption, in which it is assumed that a restarted transaction is replaced by a new, independent transaction, rather than running the same transaction over again. This assumption is nearly always used in analytical models in order to make the modeling of restarts tractable. Another assumption that we examine has to do with *write-lock acquisition.* A number of studies that distinguish between read and write locks assume that read locks are set on read-only items and that write locks are set on the items to be updated when they are first read. In reality, however, transactions often acquire a read lock on an item, then examine the item, and only then request that the read lock be upgraded to a write lock—because a transaction must usually examine an item before deciding whether or not to update it [B. Lindsay, personal communication, 1984].

We examine three concurrency control algorithms in this study, two locking algorithms and an optimistic algorithm, which represent extremes 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 our model are described in Section 3. Section 4 discusses the performance metrics and statistical methods used for the experiments, and it also discusses how a number of our parameter values were chosen. Section 5 presents the resource-related performance experiments and results. Section 6 presents the results of our examination of the other modeling assumptions

described above. Finally, in Section 7 we summarize the main conclusions of this 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 they access the same object and either (1) a_i is read and a_j is write, or (2) 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* [9]. For this study we have chosen to examine the following 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 that they also write. If a lock request is denied, the requesting transaction is blocked. A waits-for graph of transactions is maintained [23], and deadlock detection is performed each time a transaction blocks.[1] If a deadlock is discovered, the youngest transaction in the deadlock cycle is chosen as the victim and restarted. Dynamic two-phase locking [23] 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 that 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 occurring repeatedly. A concurrency control strategy similar to this one was considered in [50 and 51].

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 that committed during its lifetime. The optimistic method proposed by Kung and Robinson [27] 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

[1] Blocking's performance results would change very little if periodic deadlock detection were assumed instead [4].

transaction, since any detected conflict is with an already committed transaction. A restart delay is also unnecessary for the blocking algorithm, since the same deadlock cannot arise repeatedly.

3. PERFORMANCE MODEL

There are three main parts of a concurrency control performance model: a database system model, a user model, and a transaction model. The *database system model* captures the relevant characteristics of the system's hardware and software, including the physical resources (CPUs and disks) and their associated schedulers, the characteristics of the database (e.g., its size and granularity), the load control mechanism for controlling the number of active transactions, and the concurrency control algorithm itself. The *user model* captures the arrival process for users, assuming either an open system or else a closed system with terminals. Also included in the user model is the nature of users' transactions, since they may either be batch-style (noninteractive) or interactive in their behavior. Finally, the *transaction model* captures the reference behavior and processing requirements of the transactions in the workload. A transaction can be thought of as being described via two characteristic strings. There is a logical reference string, which contains concurrency control level read and write requests, and a physical reference string, which contains requests for accesses to physical items on disk and the associated CPU processing time for each item accessed. These strings are typically described in a probabilistic manner for a class of transactions. In addition, if there is more than one class of transactions in the workload, the transaction model must specify the mix of transaction classes. It is our view that a concurrency control performance model that fails to include any of these three major parts is in some sense incomplete. Although our model description is not broken down in exactly this way (it is more transaction-flow oriented), it should become clear to the reader that our model includes all three parts.

Central to our simulation model 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 [12 and 15], which in turn had its origins in the model of [39, 40, and 41]. 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 reenters the concurrency control queue and makes the request. It is assumed for modeling

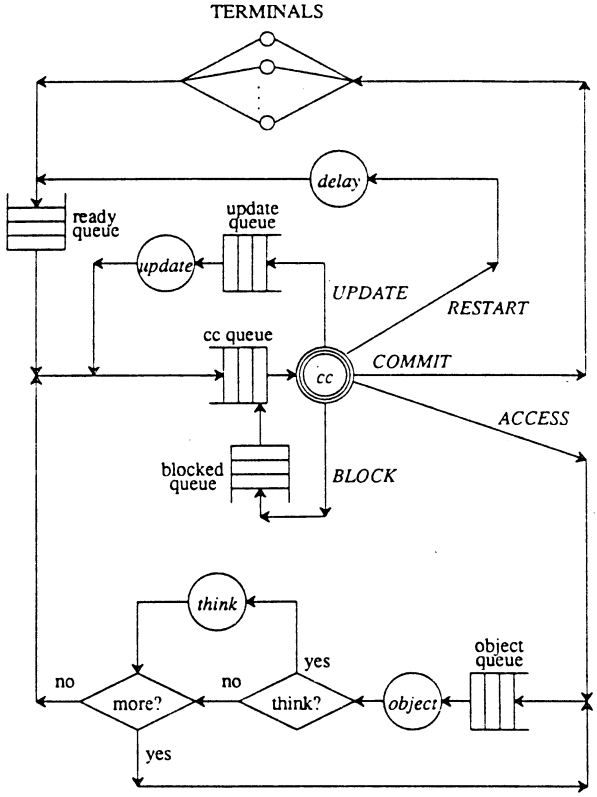

Fig. 1. Logical queuing model.

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.[2] 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 first enters the *update queue* and writes its deferred updates into the database. Deferred updates are assumed here because our modeling framework is intended to support *any* concurrency control algorithm—all algorithms operate correctly with

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

deferred updates, but not all algorithms work with recovery schemes that allow in-place updates prior to transaction-commit time. (Examples of recovery schemes that use some form of deferred updates to minimize the cost of backing out aborted transactions include the Commercial INGRES recovery mechanism [45], the database cache of Elhard and Bayer [18], and the POSTGRES recovery mechanism [49]. Results on the performance of such recovery schemes and their alternatives may be found in [2 and 38]; a study of deferred versus in-place updates and the associated cost of backing out restarted transactions in the presence of limited buffer space is included in [2].)

To further illustrate the flow of transactions in 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). Wait queues for locks and a waits-for graph are maintained by an algorithm-specific portion of the simulator. 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).

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 model parameters. The physical queuing model is depicted in Figure 2, and Table I summarizes the associated model 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. Model 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 a probabilistic model of a database that 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, since 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

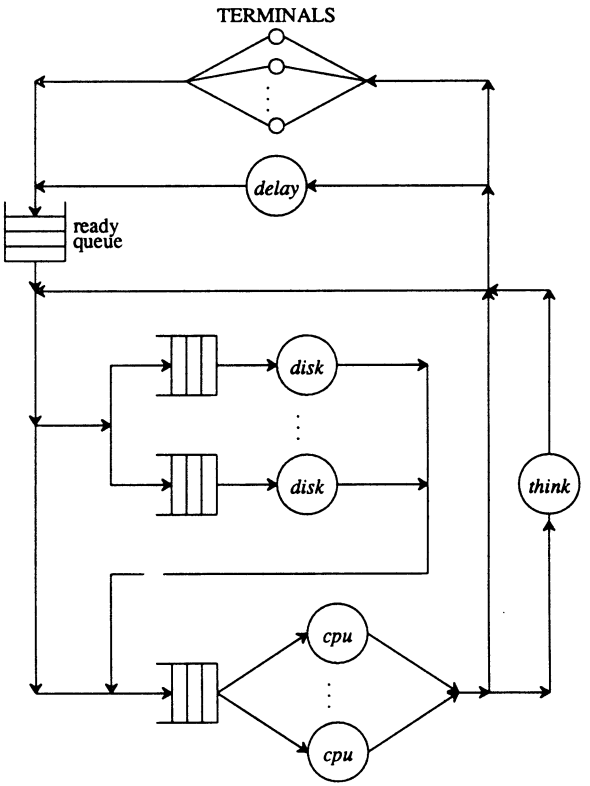

Fig. 2. Physical queuing model.

Table I. Model Parameters

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)
int_think_time	Mean intratransaction think time
restart_delay	Mean transaction restart delay
num_terms	Number of terminals
mpl	Multiprogramming level
ext_think_time	Mean time between transactions
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

time delay between the completion of a transaction and the initiation of a new transaction from a terminal, and the *int_think_time* parameter is the mean intratransaction think time (if any). We assume that both of these think times are exponentially distributed. To model interactive workloads, transactions can

be made to undergo a thinking period between finishing their reads and starting their writes.

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*.

The reader may have noted the absence of explicit concurrency control cost parameters. We assume for the purpose of this study that the cost of performing concurrency control operations is negligible compared to the cost of accessing objects. It has been shown elsewhere that the concurrency control request processing costs for algorithms based on locking and optimistic methods are roughly comparable [13], the main difference being the times at which these costs are incurred; it has also been argued that deadlock detection should not add significantly to the overhead of locking in a centralized database system [3, 13]. Thus, our negligible concurrency control cost assumption should not bias our results. Other sources of overhead that we do not consider in this study are the degradation of operating system and/or database system performance under high multiprogramming levels because of effects such as increased instruction path lengths due to large control structures (e.g., process tables), page thrashing due to memory limitations, increases in context switching, contention for high-traffic semaphores, and so forth. Database operating system issues such as context switching overhead and semaphore contention have been previously addressed by others [25].

4. GENERAL EXPERIMENT INFORMATION

The remainder of this paper presents results from a number of experiments designed to investigate the alternative modeling assumptions discussed in Section 1. Section 5 will describe studies with infinite resources, a small number of resources, various intermediate resource levels, and interactive transactions. Section 6 will describe studies of alternative modeling assumptions regarding restarted transactions and write-lock acquisition. First, however, this section of the paper will discuss the performance metrics and statistical methods used in the rest of the paper, and it will also explain how we chose many of the parameter settings used in the experiments reported here.

4.1 Performance Metrics

The primary performance metric used throughout the paper is the transaction throughput rate, which is the number of transactions completed per second. We employed a modified form of the batch means method [46] for the statistical data analyses of our throughput results, and each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90 percent confidence intervals.[3] The actual batch time varied from experiment to experiment, but our

[3] More information on the details of the modified batch means method may be found in [12].

throughput confidence intervals were typically in the range of plus or minus a few percentage points of the mean value, more than sufficient for our purposes. We omit confidence interval information from our graphs for clarity, but we discuss only the statistically significant performance differences when summarizing our results. Response times, expressed in seconds, are also given in some cases. These are measured as the difference between when a terminal first submits a new transaction and when the transaction returns to the terminal following its successful completion, including any time spent waiting in the ready queue, time spent before (and while) being restarted, and so forth. The response time variance is also examined in a few cases.

Several additional performance-related metrics are used in analyzing the results of our experiments. In analyzing concurrency control activity for the three algorithms, two conflict-related metrics are employed. The first metric is the *blocking ratio*, which gives the average number of times that a transaction has to block per commit (computed as the ratio of the number of transaction-blocking events to the number of transaction commits). The other conflict-related metric is the *restart ratio*, which gives the average number of times that a transaction has to restart per commit (computed similarly). The last set of metrics used in our analysis are the total and useful disk utilizations. The total utilization for a disk gives the fraction of time during which it is busy. The useful utilization indicates the fraction of disk time used to do work that actually completed, excluding the fraction of time used for work that was later undone because of restarts. Note that since disks are uniformly selected when a request arrives, all of the disks have the same utilization over the range of our long simulation times. Disk utilization is used instead of CPU utilization because the disks turn out to be the bottleneck resource with our parameter settings (discussed next).

4.2 Parameter Settings

Table II gives the values of the simulation parameters that all of our experiments have in common (except where otherwise noted). Parameters that vary from experiment to experiment are not listed in Table II and will instead be given with the description of the relevant experiments.

The number of terminals is set to 200 in this study. The multiprogramming level, which limits the number of active transactions, is varied from 5 transactions up to the total number of terminals. This range was chosen for several reasons. First, it includes values that we consider to be reasonable for actual database systems. Second, varying the multiprogramming level in this way provides a wide range of operating conditions with respect to both data contention (conflict probabilities) and resource contention (waiting for CPUs and disks). The object processing costs were chosen on the basis of our notion of roughly what realistic values might be. In varying the number of CPUs and disks, an issue not addressed in Table II, we decided to use 1 CPU and 2 disks as 1 *resource unit*, and then vary the number of resource units assumed. In cases in which we have 1 CPU, we have 2 disks, and in cases in which we have N CPUs, we have $2N$ disks. This balance of CPUs and disks makes the utilization of these resources about equal (i.e., balanced) with our parameter values, as opposed to being either strongly CPU bound or strongly I/O bound; in particular, the system is just slightly I/O

Table II. Simulation Parameter Settings

Parameter	Value
db_size	1,000 pages
tran_size	8-page readset
max_size	12-page readset (maximum)
min_size	4-page readset (minimum)
write_prob	0.25
restart_delay	zero or adaptive (see text)
num_terms	200 terminals
mpl	5, 10, 25, 50, 75, 100, and 200
ext_think_time	1 second
obj_io	35 milliseconds
obj_cpu	15 milliseconds

bound. As for the *restart_delay* parameter, we mentioned in Section 2 that only the immediate-restart algorithm demands a restart delay. Thus, the delay is set to zero for the other two algorithms. For the immediate-restart algorithm we use an exponential delay 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 the immediate-restart algorithm is sensitive to the restart delay time, particularly in the infinite resource case. Our preliminary 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.

In choosing parameter values for our experiments, we wanted to choose database and transaction sizes[4] that would jointly yield a region of operation that would allow the interesting performance effects to be observed without requiring impossibly long simulation times. A preliminary experiment was conducted with an average transaction size of 8 reads and 2 writes, as shown in Table II, but with a database size of 10,000 pages. Due to the large database size and the relatively small transaction size, there were few conflicts in this experiment. The throughput results for a system with infinite resources and a system with limited resources (1 resource unit, meaning 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 reported in [1, 2, 12, and 15] and elsewhere. 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. Note also that the throughput curves reach a plateau at a multiprogramming level of 75 in Figure 3 (the infinite resource case). This is due to the fact that with 200 terminals, a 1 second think time, and an expected execution time of 500 milliseconds, increasing the allowed number of active transactions beyond 75 has no effect—all available transactions are already active, and the rest are in the think state. A plateau is reached earlier in Figure 4 (the limited resource case) because the resources are already saturated

[4] These sizes are expressed in pages, as we equate objects and pages in this study.

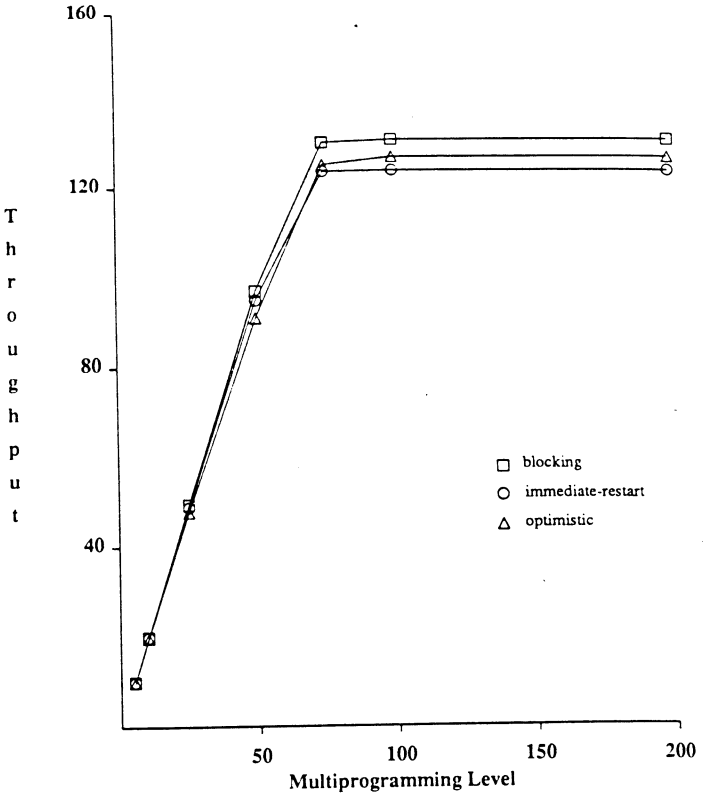

Fig. 3. Throughput (∞ resources).

with 25 concurrently active transactions. Since we are interested in investigating *differences* in concurrency control strategies, we decreased the database size to 1000 objects, as shown in Table II, to create a situation in which conflicts are more frequent. The remainder of our experiments were performed using this smaller database size.

Before closing this section, we should mention that there are a number of parameters that we could have varied but did not. For example, we could have varied the size of transactions, their distribution of sizes, or the granularity of the database; we could have varied the write probability for transactions; we could have investigated workloads containing several classes of transactions, and so forth. We also could have varied our notion of a resource unit, examining systems with less balanced resource utilization characteristics. For the purposes of this study, such variations were not of interest. Our goal was to see how certain basic assumptions affect the results of a concurrency control performance study, not to investigate exactly how performance varies with all possible sets of parameter values. Also, we have reported on the results of experiments with variations such as these elsewhere [12, 14, 15]. Our experience with variations of the first type is that for the most part they are just different ways of varying the probability of conflicts. Results regarding the relative performance of concurrency

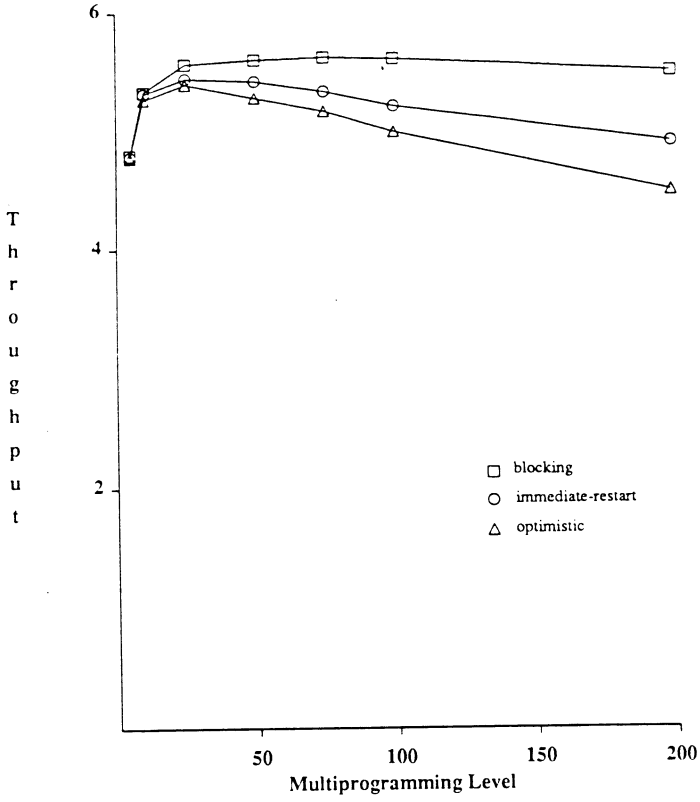

Fig. 4. Throughput (1 resource unit).

control algorithms appear to be insensitive to the particular manner in which this probability is varied, as long as it is indeed varied. (The only significant exception to this statement applies to workloads containing a wide range of transaction sizes or types, where algorithms sometimes display performance biases against a particular transaction class—for example, very large transactions can starve under the immediate-restart and optimistic algorithms, particularly with a workload containing a mix of short and long transactions—but we have studied this effect elsewhere as well [14].) Variations of the second type have been found to be of little or no interest in determining relative algorithm performance, since the primary factor determining the shape of the performance curves for an algorithm is the utilization of the bottleneck resource and not the fact that the bottleneck is the CPU or disk subsystem.

5. RESOURCE-RELATED ASSUMPTIONS

We performed several simulation experiments to study the implications of different resource-related assumptions on the performance of the three concurrency control algorithms described in Section 3. We investigated the performance of the three algorithms under the infinite resources assumption, with limited resources, and with varying numbers of CPUs and disks. To vary the number of

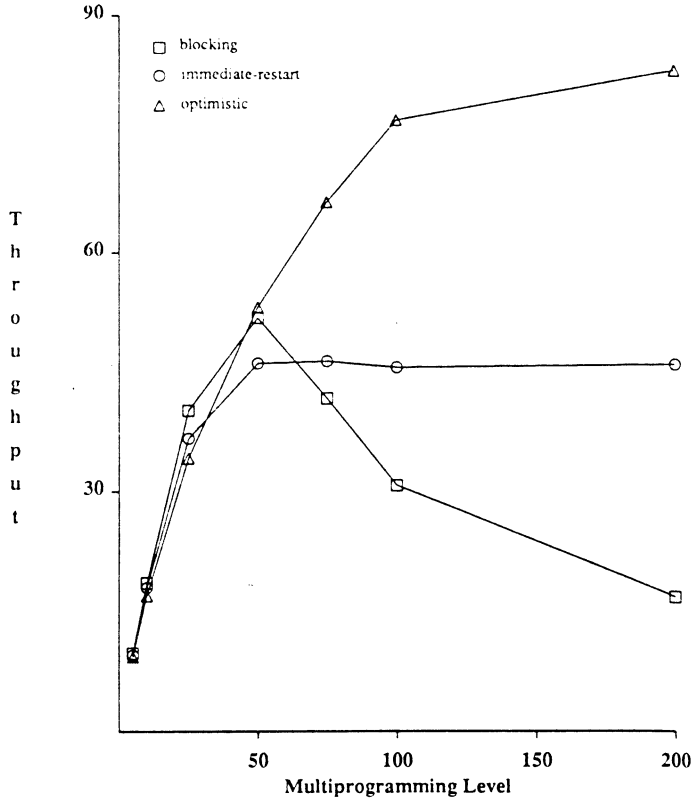

Fig. 5. Throughput (∞ resources).

CPUs and disks, we varied the number of resource units employed (each of which consists of 1 CPU and 2 disks). We also examined the case of an alternative workload type, an interactive workload.

5.1 Experiment 1: Infinite Resources

The first resource-related experiment examined the performance characteristics of the three strategies for a variety of multiprogramming levels, assuming infinite resources. With infinite resources, the throughput should be a nondecreasing function of multiprogramming level in the absence of data contention.[5] However, for a given size database, the probability of conflicts increases as the multiprogramming level increases. For blocking, the increased conflict probability manifests 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 results in a larger number of restarts.

Figure 5 shows the throughput results for Experiment 1. Blocking starts thrashing as the multiprogramming level is increased beyond a certain level,

[5] We do not attempt to model system overhead phenomena such as the effects of increased operating system and database system control structure sizes on system path lengths.

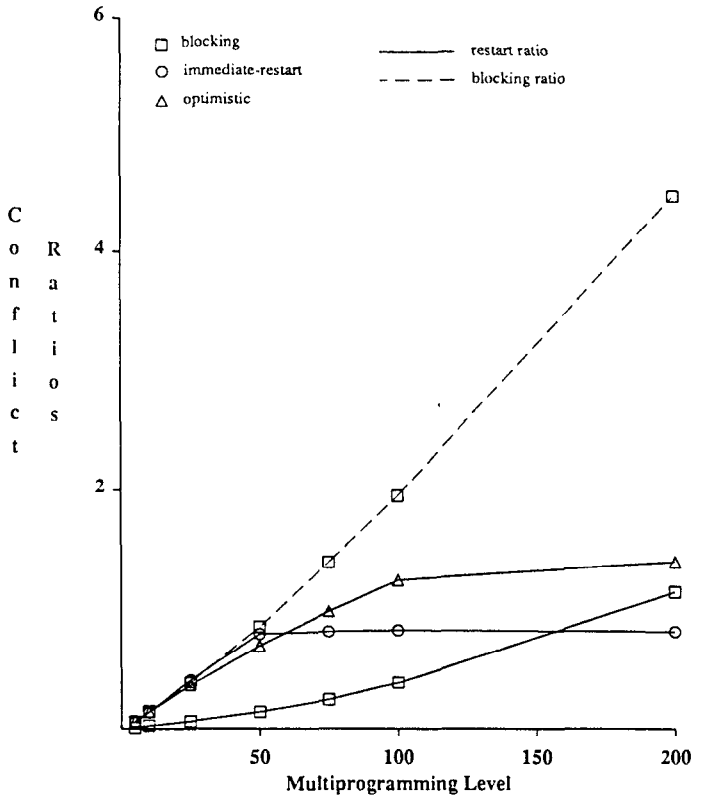

Fig. 6. Conflict ratios (∞ resources).

whereas the throughput keeps increasing for the optimistic algorithm. These results agree with predictions in [20] that were based on similar assumptions. Figure 6 shows the blocking and restart ratios 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 number of transactions available to run and make forward progress, rather than to an increase in the number of restarts. This result is in agreement with the assertion in [6, 50 and 51] 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 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. As described in Section 4, the duration of the delay is *adaptive*, equal to the running average of the response

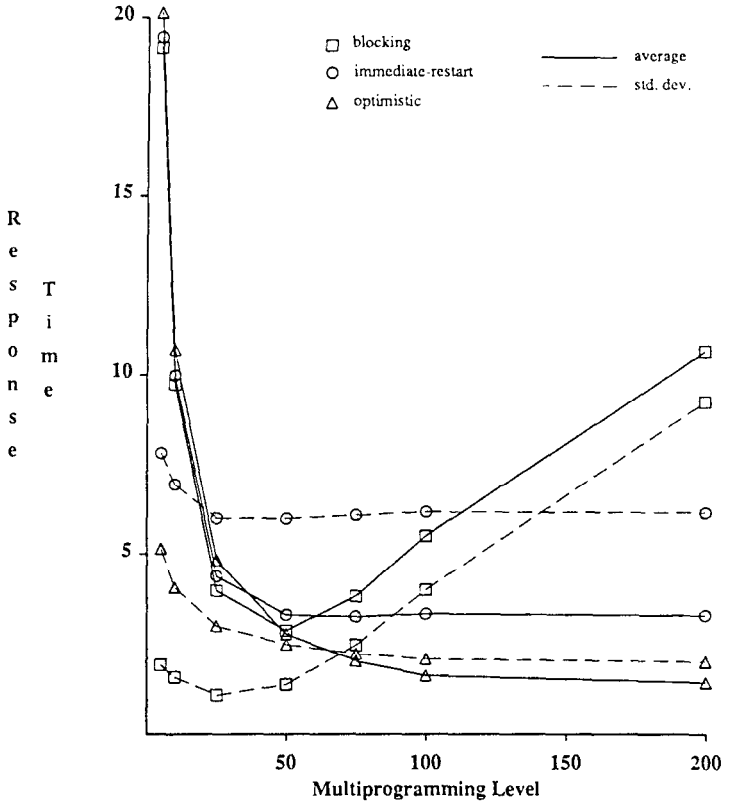

Fig. 7. Response time (∞ resources).

time. Because of this adaptive delay, the immediate-restart algorithm reaches a point beyond which all of the transactions that are not active are either in a restart delay state or else in a terminal thinking state (where a terminal is pausing between the completion of one transaction and submitting a new transaction). This point is reached when the number of active transactions in the system is such that a new transaction is basically sure to conflict with an active transaction and is therefore sure to be quickly restarted and then delayed. Such delays increase the average response time for transactions, which increases their average restart delay time; this has the effect of reducing the number of transactions competing for active status and in turn reduces the probability of conflicts. In other words, the adaptive restart delay creates a negative feedback loop (in the control system sense). Once the plateau is reached, there are simply no transactions waiting in the ready queue, and increasing the multiprogramming level is a "no-op" beyond this point. (Increasing the *allowed* number of active transactions cannot increase the *actual* number if none are waiting anyway.)

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 much smaller for blocking than for the immediate-restart algorithm over most of the multiprogramming levels explored, and it is also smaller than that of the optimistic algorithm for the lower multiprogramming levels (i.e., until blocking's performance begins to degrade significantly because of thrashing). The immediate-restart algorithm has a large response-time variance due to its restart delay. When a transaction has to be restarted because of a lock conflict during its execution, its response time is increased by a randomly chosen restart delay period with a mean of one entire response time, and in addition the transaction must be run all over again. Thus, a restart leads to a large response time increase for the restarted transaction. The optimistic algorithm restarts transactions at the end of their execution and requires restarted transactions to be run again from the beginning, but it does not add a restart delay to the time required to complete a transaction. The blocking algorithm restarts transactions much less often than the other algorithms for most multiprogramming levels, and it restarts them during their execution (rather than at the end) and without imposing a restart delay. Because of this, and because lock waiting times tend to be quite a bit smaller than the additional response time added by a restart, blocking has the lowest response time variance until it starts to thrash significantly. A high variance in response time is undesirable from a user's standpoint.

5.2 Experiment 2: Resource-Limited Situation

In Experiment 2 we analyzed the impact of limited resources on the performance characteristics of the three concurrency control algorithms. A database system with one resource unit (one CPU and two disks) was assumed for this experiment. The throughput results are presented in Figure 8.

Observe that for all three algorithms, the throughput curves indicate thrashing—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 limited 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, nonconflicting 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 since 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. As one would expect, there is a direct correlation between the useful utilization curves of Figure 9 and the throughput curves of Figure 8. For blocking, the throughput peaks at mpl = 25, where the disks are being

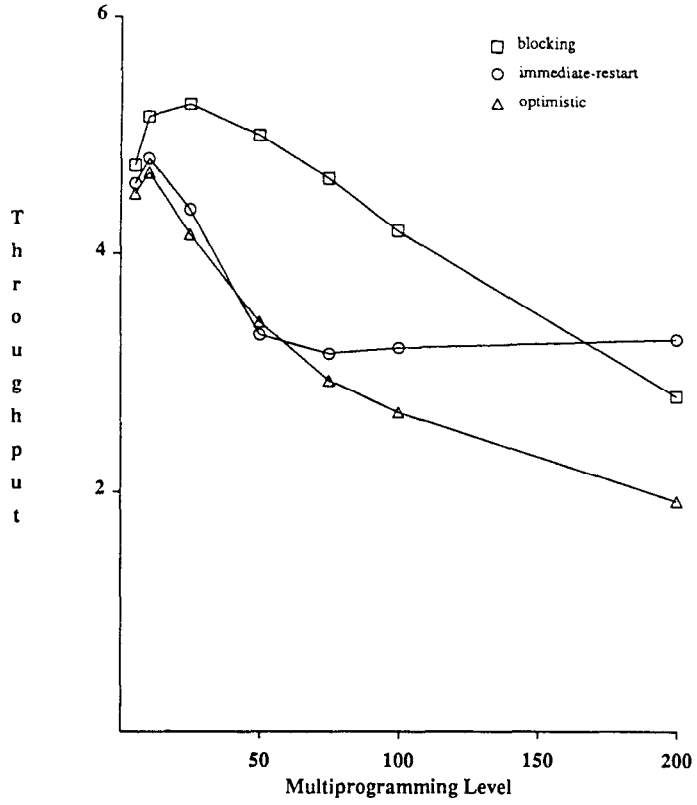

Fig. 8. Throughput (1 resource unit).

97 percent utilized, with a useful utilization of 92 percent.[6] Increasing the multiprogramming level further only increases data contention, and the throughput decreases as the amount of blocking and thus the number of deadlock-induced restarts increase rapidly. 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 increase in the restart ratio means that a larger fraction of the disk time is spent doing work 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 throughput 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, since all of the nonactive transactions are either in a restart delay state or thinking.

With regard 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-restart performed

[6] The actual throughput peak may of course be somewhere to the left or right of 25, in the 10–50 range, but that cannot be determined from our data.

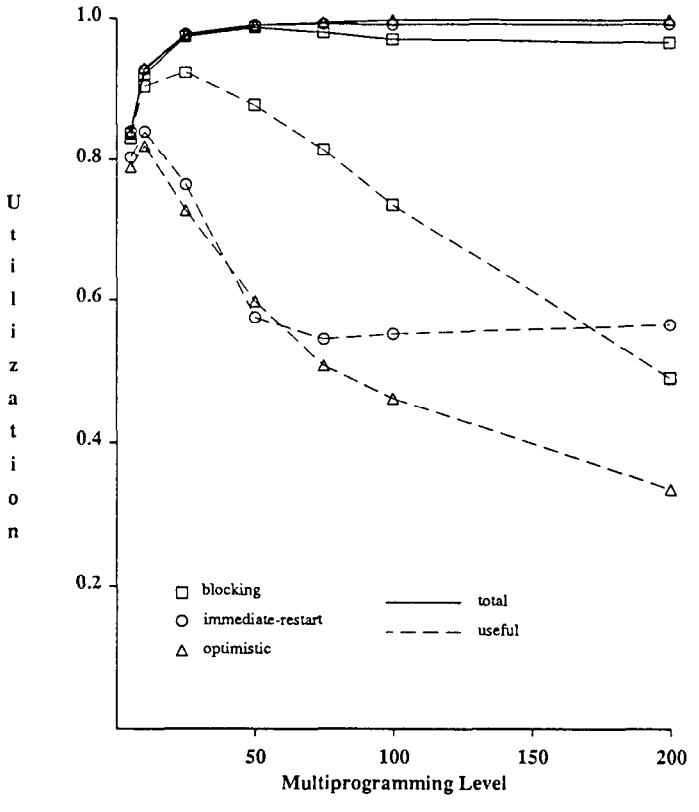

Fig. 9. Disk utilization (1 resource unit).

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 the optimistic algorithm at this same multiprogramming level.

Figure 10 gives the average and the standard deviation of response time for the three algorithms in the limited resource case. The differences are even more noticeable than in the infinite resource case. Blocking has the lowest delay (fastest response time) over most of the multiprogramming levels. 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-restart is the worst, and the optimistic algorithm is in between the two. As in Experiment 1, the immediate-restart algorithm exhibits a 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

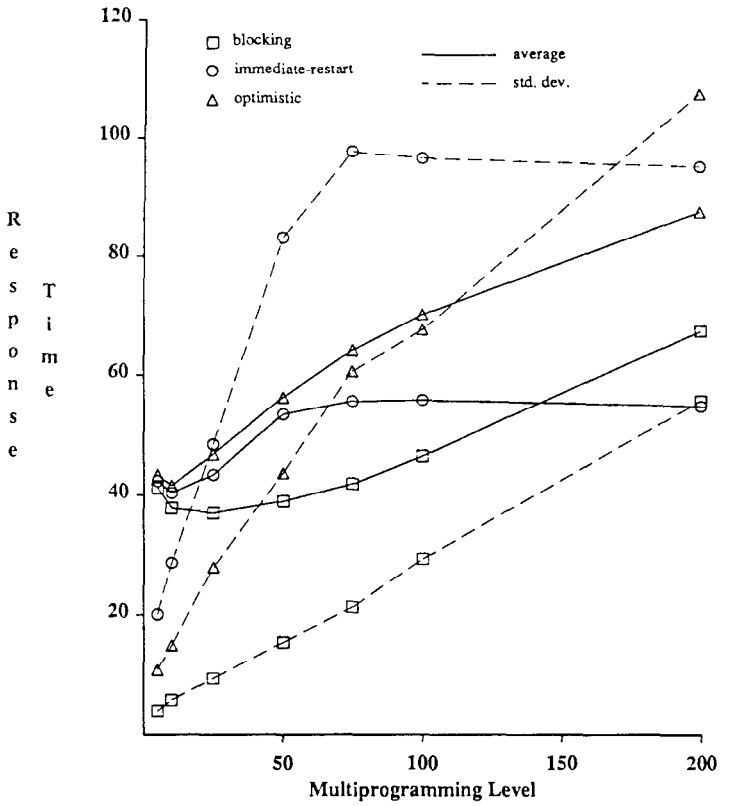

Fig. 10. Response time (1 resource unit).

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 keep multiprogramming at a level that gives the best performance. In this experiment, the highest throughput and smallest response time were achieved using the blocking algorithm at mpl = 25. 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 population 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. 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 in which the adaptive restart delay was used for restarted transactions in both the blocking

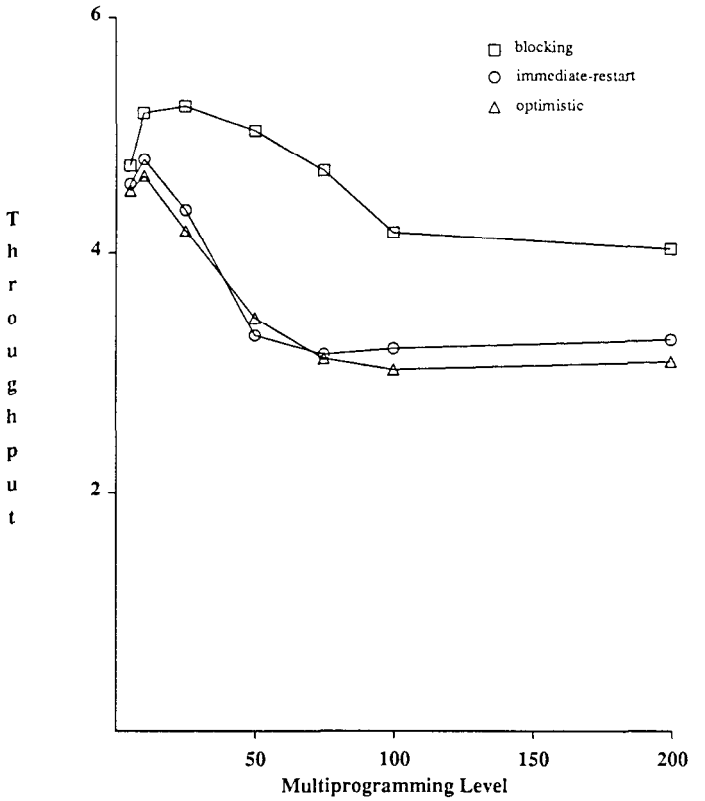

Fig. 11. Throughput (adaptive restart delays).

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-restart, reducing data contention at the upper range of multiprogramming levels. 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.

5.3 A Brief Aside

Before discussing the remainder of the experiments, a brief aside is in order. Our concurrency control performance model includes a time delay, *ext_think_time*, between the completion of one transaction and the initiation of the next transaction from a terminal. Although we feel that such a time delay is necessary in a

realistic performance model, a side effect of the delay is that it can lead the database system to become "starved" for transactions when the multiprogramming level is increased beyond a certain point. That is, increasing the multiprogramming level has no effect on system throughput beyond this point because the actual number of active transactions does not change. This form of starvation can lead an otherwise increasing throughput to reach a plateau when viewed as a function of the multiprogramming level. In order to verify that our conclusions were not distorted by the inclusion of a think time, we repeated Experiments 1 and 2 with no think time (i.e., with *ext_think_time* = 0).

The throughput results for these experiments are shown in Figures 12 and 13, and the figures to which these results should be compared are Figures 5 and 8. It is clear from these figures that, although the exact performance numbers are somewhat different (because it is now never the case that the system is starved for transactions while one or more terminals is in a thinking state), the relative performance of the algorithms is not significantly affected. The explanations given earlier for the observed performance trends are almost all applicable here as well. In the infinite resource case (Figure 12), blocking begins thrashing beyond a certain point, and the immediate-restart algorithm reaches a plateau because of the large number of restarted transactions that are delaying (due to the restart delay) before running again. The only significant difference in the infinite resource performance trends is that the throughput of the optimistic algorithm continues to improve as the multiprogramming level is increased, instead of reaching a plateau as it did when terminals spent some time in a thinking state (and thus sometimes caused the actual number of transactions in the system to be less than that allowed by the multiprogramming level). Franaszek and Robinson predicted this [20], predicting logarithmically increasing throughput for the optimistic algorithm as the number of active transactions increases under the infinite resource assumption. Still, this result does not alter the general conclusions that were drawn from Figure 5 regarding the relative performance of the algorithms. In the limited resource case (Figure 13), the throughput for each of the algorithms peaks when resources become saturated, decreasing beyond this point as more and more resources are wasted because of restarts, just as it did before (Figure 8). Again, fewer and/or earlier restarts lead to better performance in the case of limited resources. On the basis of the lack of significant differences between the results obtained with and without the external think time, then, we can safely conclude that incorporating this delay in our model has not distorted our results. The remainder of the experiments in this paper will thus be run using a nonzero external think time (just like Experiments 1 and 2).

5.4 Experiment 3: Multiple Resources

In this experiment we moved the system from limited resources toward infinite resources, increasing the level of resources available to 5, 10, 25, and finally 50 resource units. This experiment was motivated by a desire to investigate performance trends as one moves from the limited resource situation of Experiment 2 toward the infinite resource situation of Experiment 1. Since the infinite resource assumption has sometimes been justified as a way of investigating what performance trends to expect in systems with many processors [20], we were interested

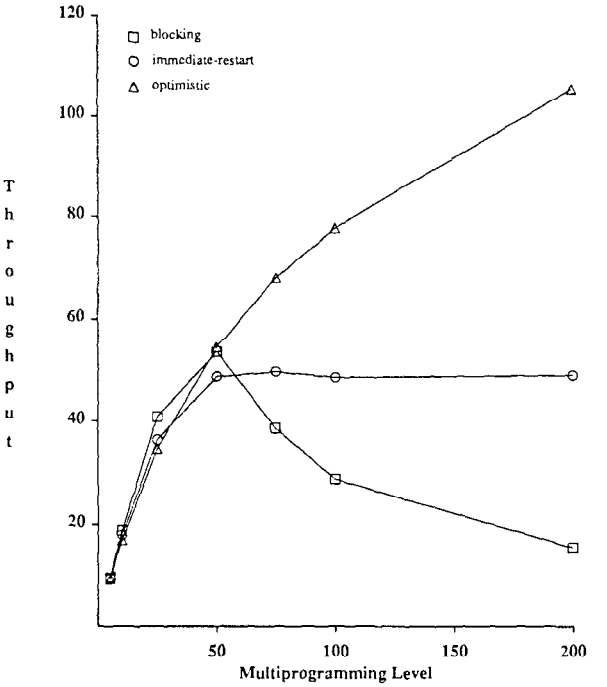

Fig. 12. Throughput (∞ resources, no external think time).

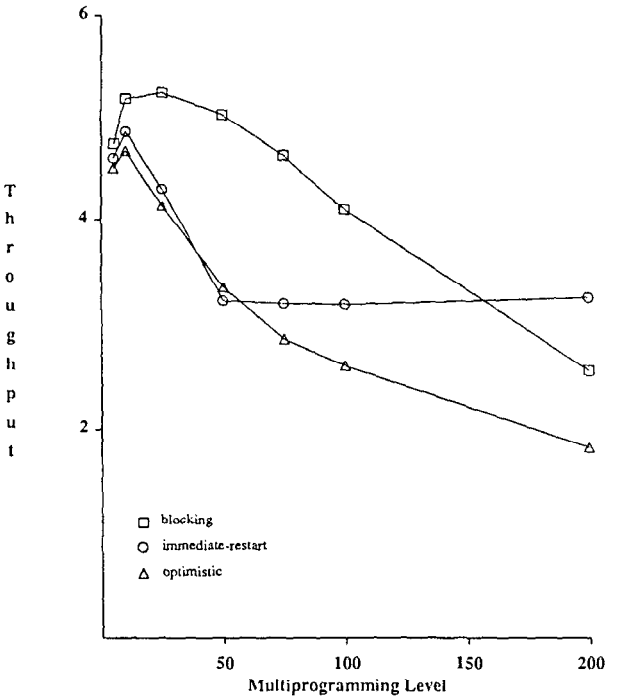

Fig. 13. Throughput (1 resource unit, no external think time).

in determining where (i.e., at what level of resources) the behavior of the system would begin to approach that of the infinite resource case in an environment such as a multiprocessor database machine.

For the cases with 5 and 10 resource units, the relative behavior of the three concurrency control strategies was fairly similar to the behavior in the case of just 1 resource unit. The throughput results for these two cases are shown in Figures 14 and 16, respectively, and the associated disk utilization figures are given in Figures 15 and 17. Blocking again provided the highest overall throughput. For large multiprogramming levels, however, the immediate-restart strategy provided better throughput than blocking (because of its restart delay), but not enough so as to beat the highest throughput provided by the blocking algorithm. With 5 resource units, where the maximum useful disk utilizations for blocking, immediate-restart, and the optimistic algorithm were 72, 60, and 58 percent, respectively, the results followed the same trends as those of Experiment 2. Quite similar trends were obtained with 10 resource units, where the maximum useful utilizations of the disks for blocking, immediate-restart, and optimistic were 56, 45, and 47 percent, respectively. Note that in all cases, the total disk utilizations for the restart-oriented algorithms are higher than those for the blocking algorithm because of restarts; this difference is partly due to wasted resources. By *wasted resources* here, we mean resources used to process objects that were later undone because of restarts—these resources are wasted in the sense that they were consumed, making them unavailable for other purposes such as background tasks.

With 25 resource units, 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 18, and the utilizations are given in Figure 19. The total and the useful disk utilizations for the maximum throughput point for blocking were 34 and 30 percent (respectively), whereas the corresponding numbers for the optimistic algorithm were 81 and 30 percent. Thus, the optimistic algorithm has become attractive because a large amount of otherwise unused resources are available, and thus the waste of resources due to restarts does not adversely affect performance. In other words, with useful utilizations in the 30 percent range, the system begins to behave somewhat like it has infinite resources. As the number of available resources is increased still further to 50 resource units, the results become very close indeed to those of the infinite resource case; this is illustrated by the throughput and utilizations shown in Figures 20 and 21. Here, with maximum useful utilizations down in the range of 15 to 25 percent, the shapes and relative positions of the throughput curves are very much like those of Figure 5 (although the actual throughput values here are still not quite as large).

Another interesting observation from these latter 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 [6, 50, 51], 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

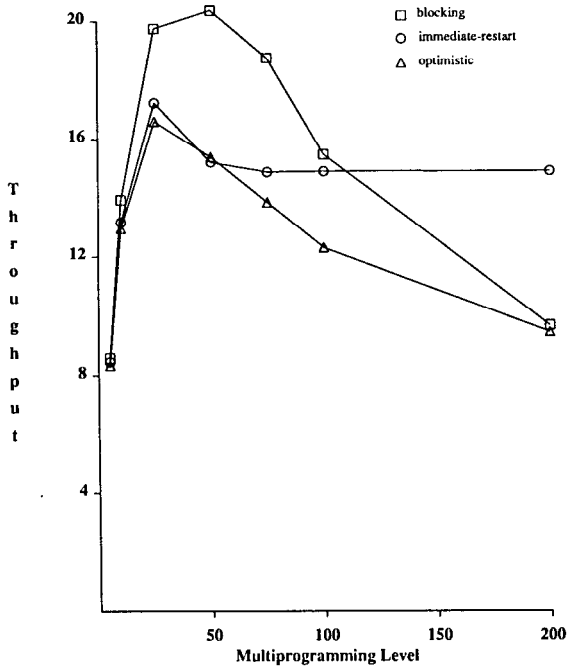

Fig. 14. Throughput (5 resource units).

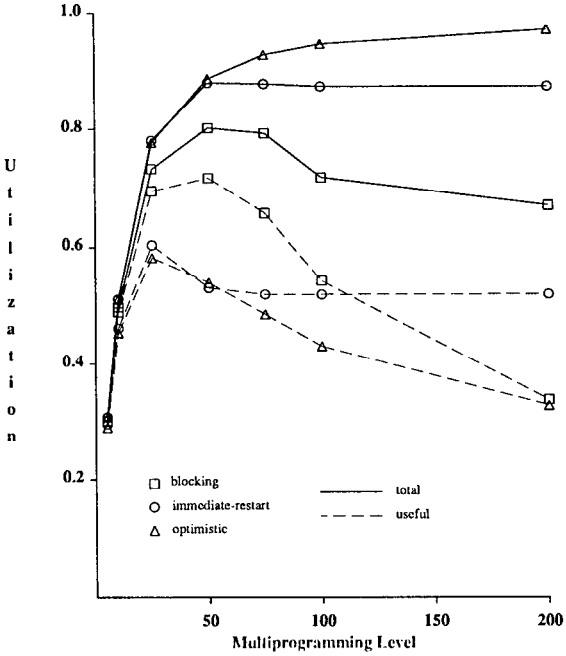

Fig. 15. Disk utilization (5 resource units).

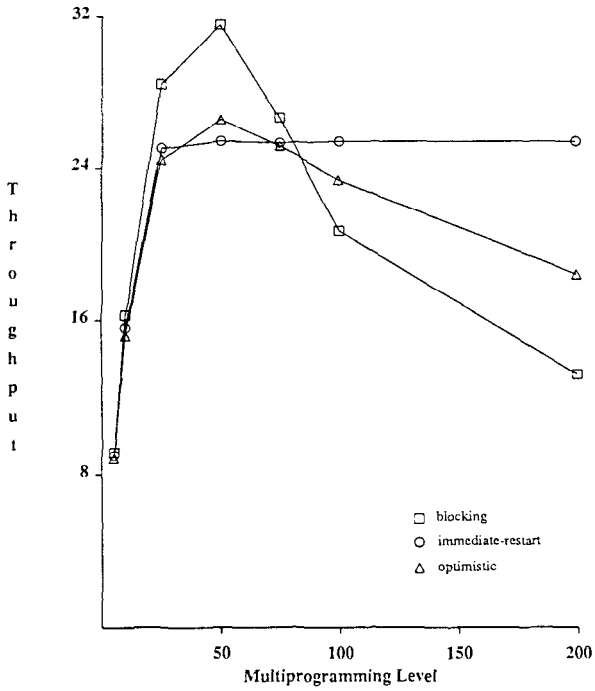

Fig. 16. Throughput (10 resource units).

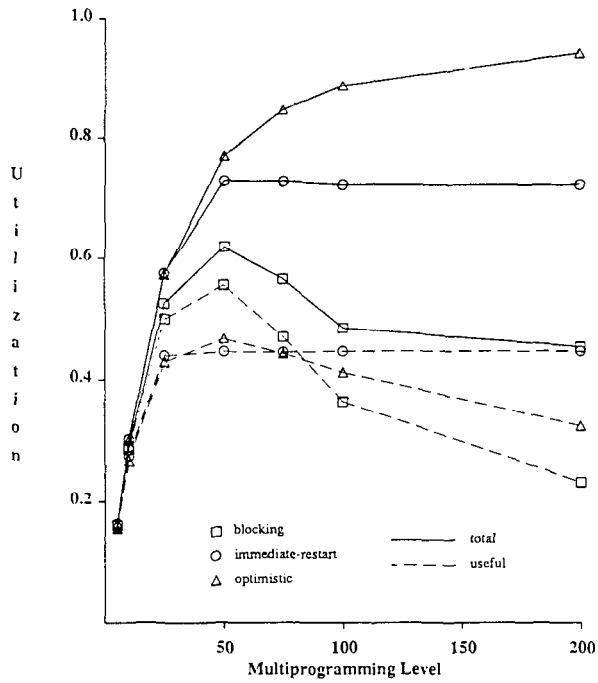

Fig. 17. Disk utilization (10 resource units).

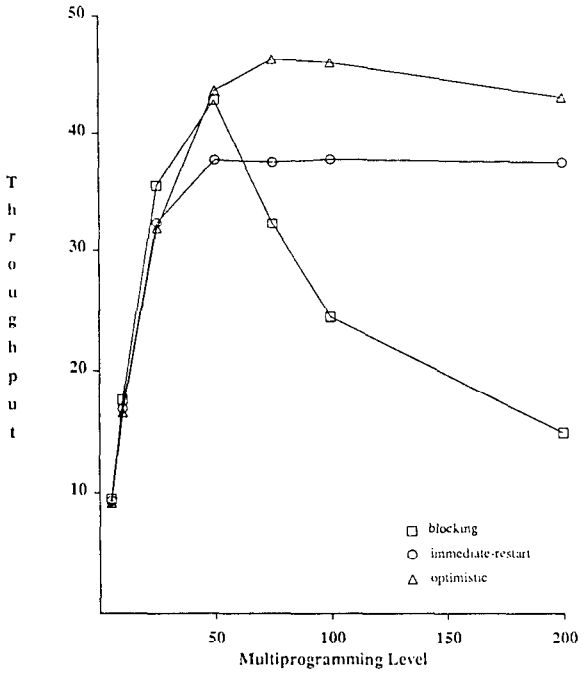

Fig. 18. Throughput (25 resource units).

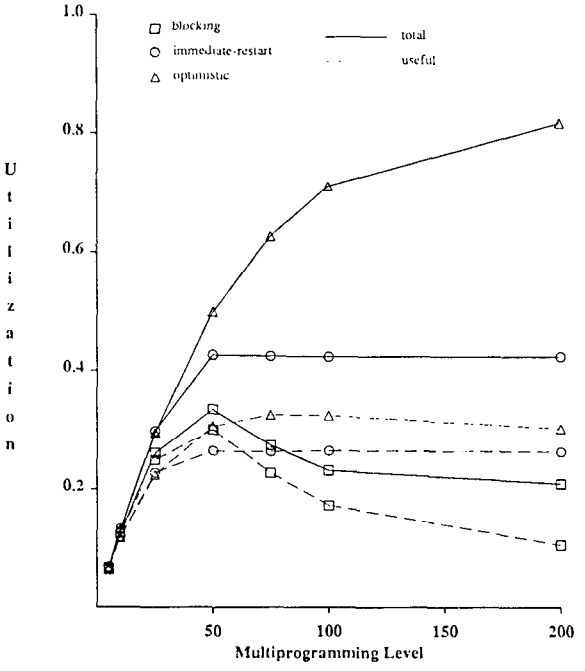

Fig. 19. Disk utilization (25 resource units).

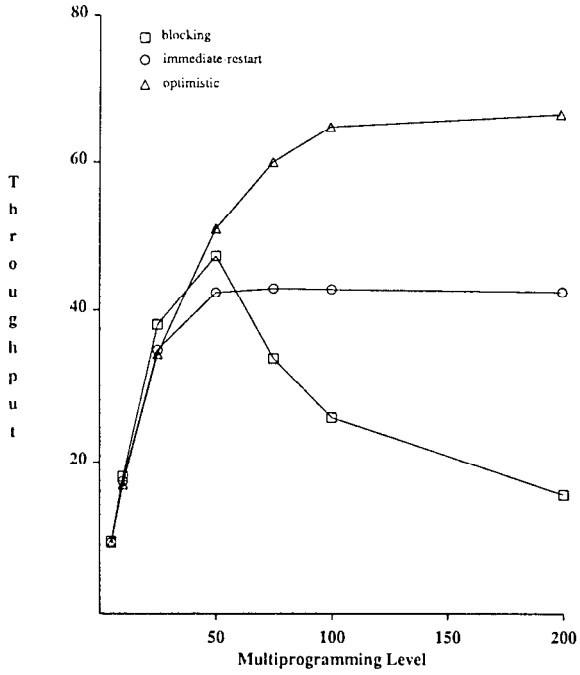

Fig. 20. Throughput (50 resource units).

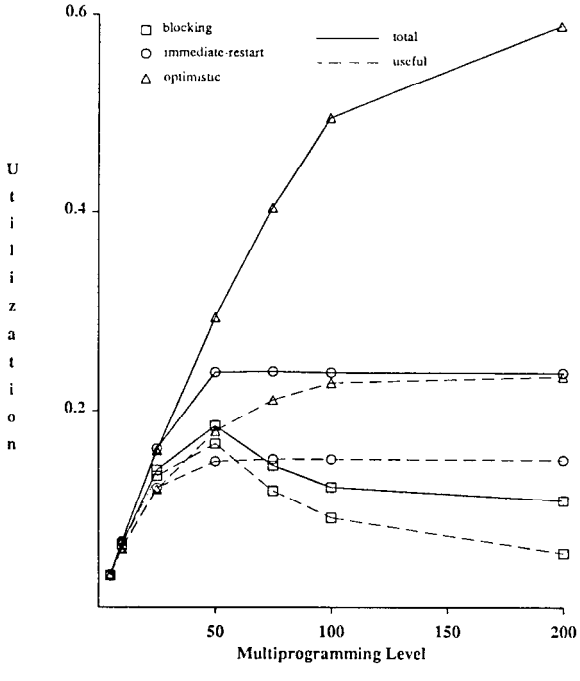

Fig. 21. Disk utilization (50 resource units).

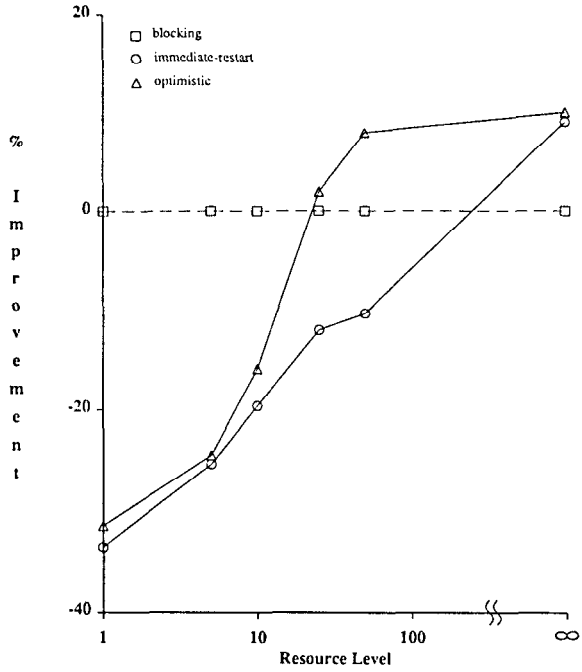

Fig. 22. Improvement over blocking (MPL = 50).

somewhat (except with 50 resource units). Thus, this strategy eventually thrashes because of the number of restarts (i.e., because of resources). With immediate-restart, 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.

As a final illustration of how the level of available resources affects the choice of a concurrency control algorithm, we plotted in Figures 22 through 24 the percent throughput improvement of the algorithms with respect to that of the blocking algorithm as a function of the resource level. The resource level axis gives the number of resource units used, which ranges from 1 to infinity (the infinite resource case). Figure 22 shows that, for a multiprogramming level of 50, blocking is preferable with up to almost 25 resource units; beyond this point the optimistic algorithm is preferable. For a multiprogramming level of 100, as shown in Figure 23, the crossover point comes earlier because the throughput for blocking is well below its peak at this multiprogramming level. Figure 24 compares the maximum attainable throughput (over all multiprogramming levels) for each algorithm as a function of the resource level, in which case locking again wins out to nearly 25 resource units. (Recall that useful utilizations were down in the mid-20 percent range by the time this resource level, with 25 CPUs and 50 disks, was reached in our experiments.)

5.5 Experiment 4: Interactive Workloads

In our last resource-related experiment, we modeled interactive transactions that perform a number of reads, think for some period of time, and then perform their

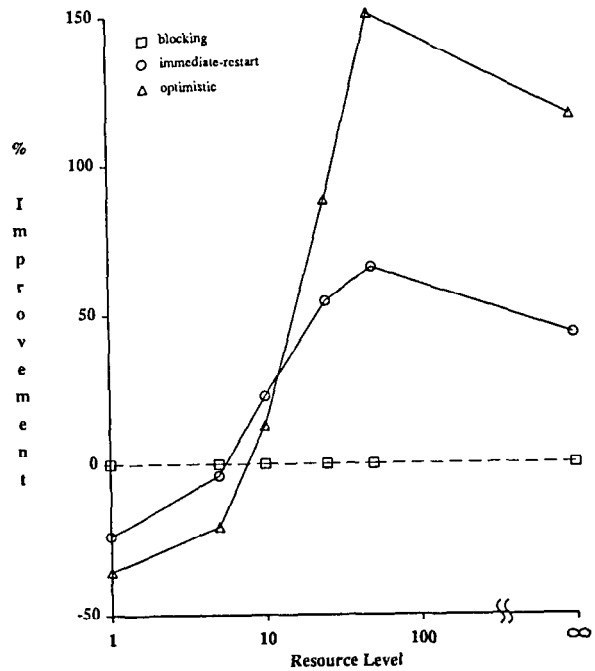

Fig. 23. Improvement over blocking (MPL = 100).

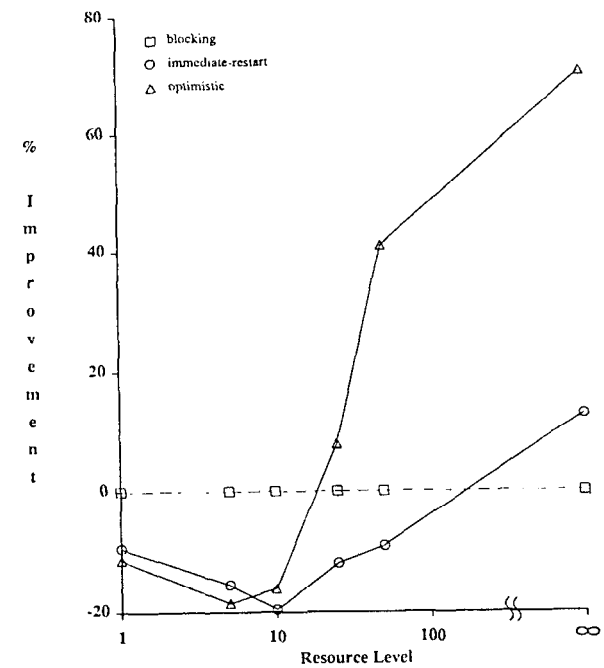

Fig. 24. Improvement over blocking (maximum).

writes. This model of interactive transactions was motivated by a 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 awhile, and then the user types "enter," causing the updates to be performed. The intent of this experiment was to find out whether large intratransaction (internal) think times would be another way to cause a system with limited resources to behave like it has infinite resources. Since Experiment 3 showed that low utilizations can lead to behavior similar to the infinite resource case, we suspected that we might indeed see such behavior here. 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 maintain roughly the same ratio of idle terminals (those in an external thinking state) to active transactions. We have assumed a limited resource environment with 1 resource unit for the system in this experiment.

Figure pairs (25, 26), (27, 28), and (29, 30) show the throughput and disk utilizations obtained for the 1, 5, and 10 second intratransaction think time experiments, respectively. On the average, a transaction requires 150 milliseconds of CPU time and 350 milliseconds of disk time, so an internal think time of 5 seconds or more is an order of magnitude larger than the time spent consuming CPU or I/O resources. Even with many transactions in the system, resource contention is significantly reduced because of such think times, and the result is that the CPU and I/O resources behave more or less like infinite resources. Consequently, for large think times, the optimistic algorithm performs better than the blocking strategy (see Figures 27 and 29). For an internal think time of 10 seconds, the useful utilization of resources is much higher with the optimistic algorithm than the blocking strategy, and its highest throughput value is also considerably higher than that of blocking. For a 5-second internal think time, the throughput and the useful utilization with the optimistic algorithm are again better than those for blocking. For a 1-second internal think time, however, blocking performs better (see Figure 25). In this last case, in which the internal think time for transactions is closer to their processing time requirements, the resource utilizations are such that resources wasted because of restarts make the optimistic algorithm the loser.

The highest throughput obtained with the optimistic algorithm was consistently better than that for immediate-restart, although for higher levels of multiprogramming the throughput obtained with immediate-restart was better than the throughput obtained with the optimistic algorithm due to the *mpl*-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.

5.6 Resource-Related Conclusions

Reflecting on the results of the experiments reported in this section, several conclusions are clear. First, a blocking algorithm like dynamic two-phase locking is a better choice than a restart-oriented concurrency control algorithm like the immediate-restart or optimistic algorithms for systems with medium to high

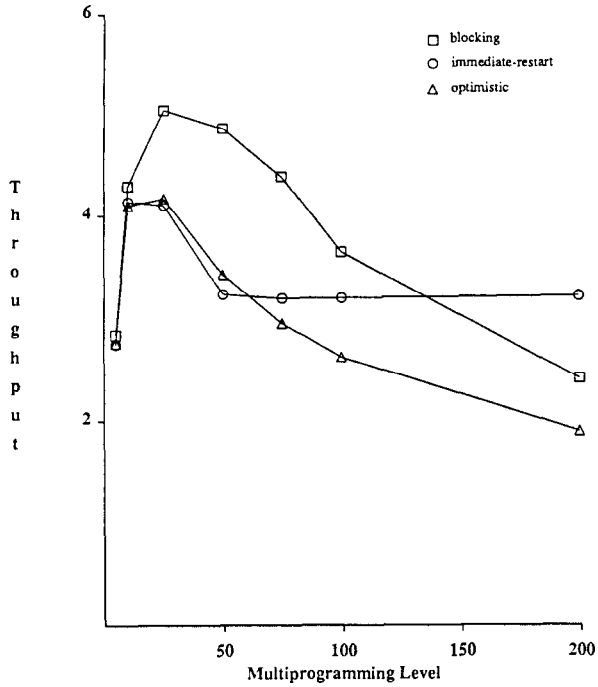

Fig. 25. Throughput (1 second thinking).

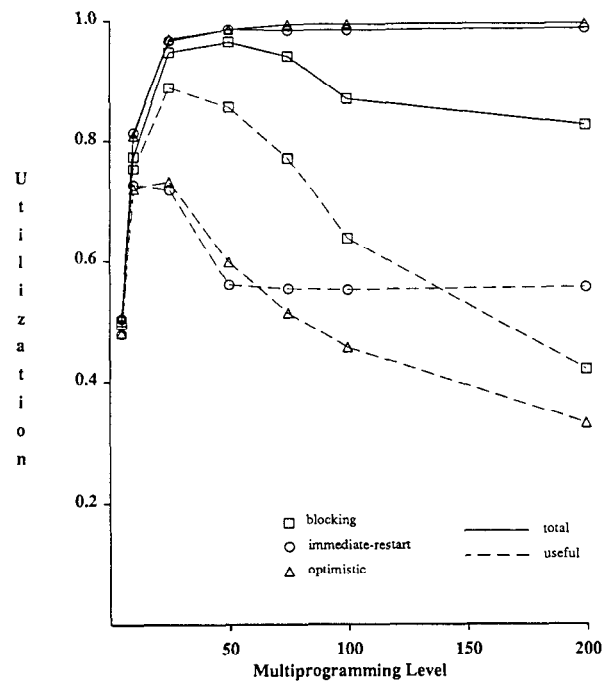

Fig. 26. Disk utilization (1 second thinking).

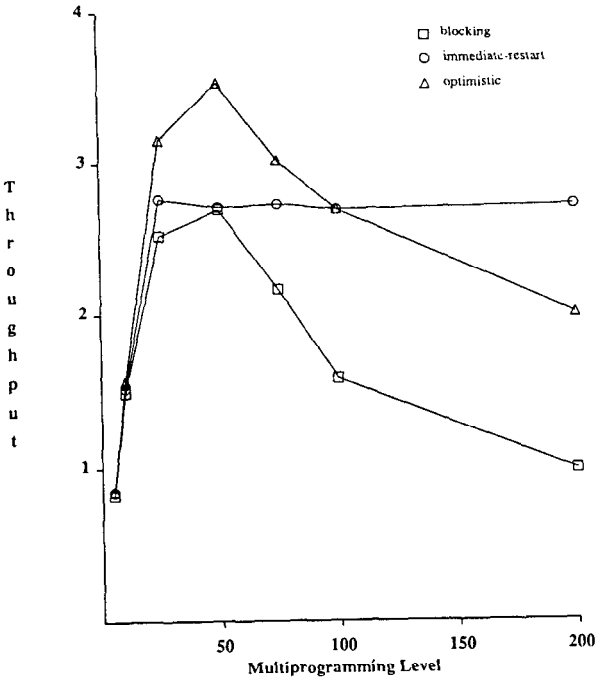

Fig. 27. Throughput (5 seconds thinking).

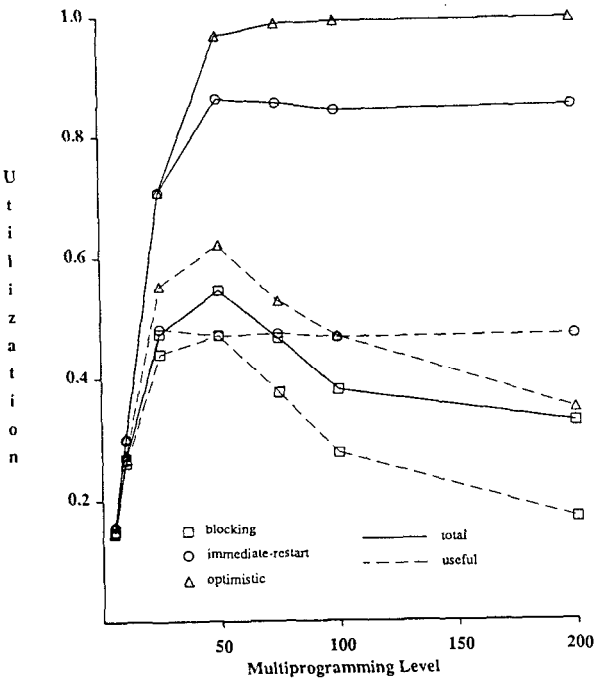

Fig. 28. Disk utilization (5 seconds thinking).

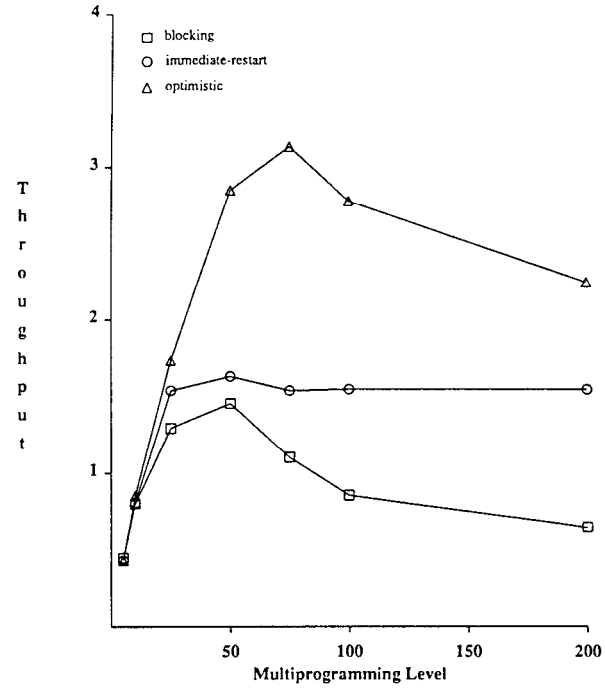

Fig. 29. Throughput (10 seconds thinking).

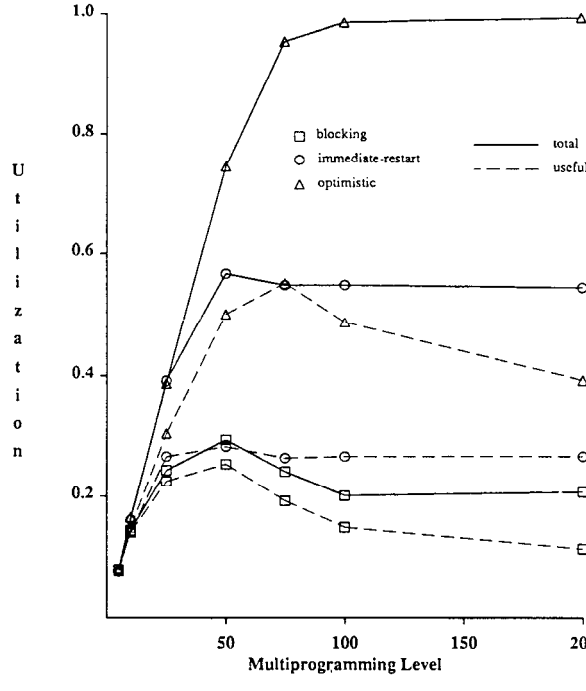

Fig. 30. Disk utilization (10 seconds thinking).

levels of resource utilization. On the other hand, if utilizations are sufficiently low, a restart-oriented algorithm becomes a better choice. Such low resource utilizations arose in our experiments with large numbers of resource units and in our interactive workload experiments with large intratransaction think times. The optimistic algorithm provided the best performance in these cases. Second, the past performance studies discussed in Section 1 were not really contradictory after all: they simply obtained different results because of very different resource modeling assumptions. We obtained results similar to each of the various studies [1, 2, 6, 12, 15, 20, 50, 51] by varying the level of resources that we employed in our database model. Clearly, then, a physically justifiable resource model is a critical component for a reasonable concurrency control performance model. Third, our results indicate that it is important to control the multiprogramming level in a database system for concurrency control reasons. We observed thrashing behavior for locking in the infinite resource case, as did [6, 20, 50, and 51], but in addition we observed that a significant thrashing effect occurs for both locking *and* optimistic concurrency control under higher levels of resource contention. (A similar thrashing effect would also have occurred for the immediate-restart algorithm under higher resource contention levels were it not for the *mpl*-limiting effects of its adaptive restart delay.)

6. TRANSACTION BEHAVIOR ASSUMPTIONS

This section describes experiments that were performed to investigate the performance implications of two modeling assumptions related to transaction behavior. In particular, we examined the impact of alternative assumptions about how restarts are modeled (real versus fake restarts) and how write locks are acquired (with or without upgrades from read locks). Based on the results of the previous section, we performed these experiments under just two resource settings: infinite resources and one resource unit. These two settings are sufficient to demonstrate the important effects of the alternative assumptions, since the results under other settings can be predicted from these two. Except where explicitly noted, the simulation parameters used in this section are the same as those given in Section 4.

6.1 Experiment 6: Modeling Restarts

In this experiment we investigated the impact of transaction-restart modeling on performance. Up to this point, restarts have been modeled by "reincarnating" transactions with their previous read and write sets and then placing them at the end of the ready queue, as described in Section 3. An alternative assumption that has been used for modeling convenience in a number of studies is the *fake restart* assumption, in which a restarted transaction is assumed to be replaced by a new transaction that is independent of the restarted one. In order to model this assumption, we had the simulator reinitialize the read and write sets for restarted transactions in this experiment. The throughput results for the infinite resource case are shown in Figure 31, and Figure 32 shows the associated conflict ratios. Solid lines show the new results obtained using the fake restart assumption, and the dotted lines show the results obtained previously under the real restart model. For the conflict ratio curves, hollow points show restart ratios and

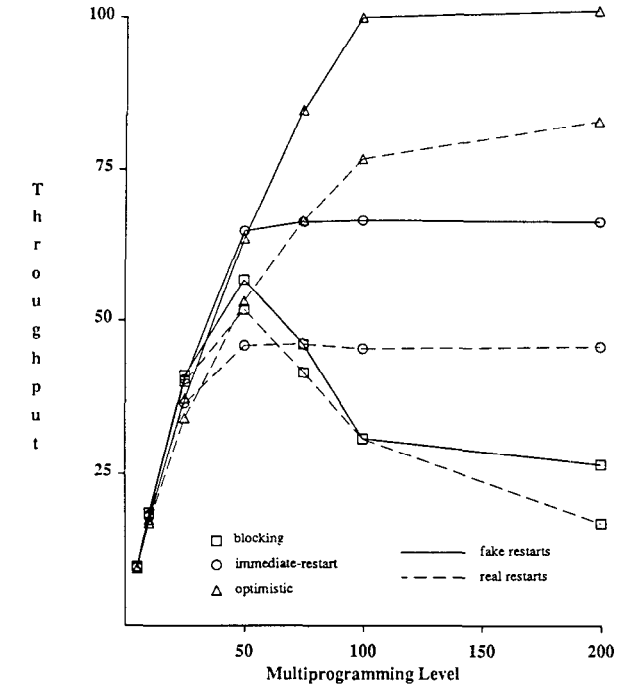

Fig. 31. Throughput (fake restarts, ∞ resources).

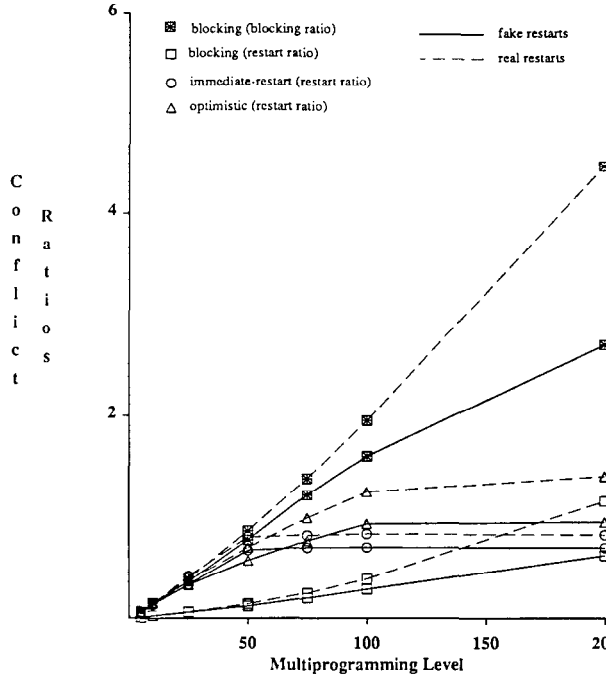

Fig. 32. Conflict ratios (fake restarts, ∞ resources).

solid points show blocking ratios. Figures 33 and 34 show the throughput and conflict ratio results for the limited resource (1 resource unit) case.

In comparing the fake and real restart results for the infinite resource case in Figure 31, several things are clear. The fake restart assumption produces significantly higher throughputs for the immediate-restart and optimistic algorithms. The throughput results for blocking are also higher than under the real restart assumption, but the difference is quite a bit smaller in the case of the blocking algorithm. The restart-oriented algorithms are more sensitive to the fake-restart assumption because they restart transactions much more often. Figure 32 shows how the conflict ratios changed in this experiment, helping to account for the throughput results in more detail. The restart ratios are lower for each of the algorithms under the fake-restart assumption, as is the blocking algorithm's blocking ratio. For each algorithm, if three or more transactions wish to concurrently update an item, repeated conflicts can occur. For blocking, the three transactions will all block and then deadlock when upgrading read locks to write locks, causing two to be restarted, and these two will again block and possibly deadlock. For optimistic, one of the three will commit, which causes the other two to detect readset/writeset intersections and restart, after which one of the remaining two transactions will again restart when the other one commits. A similar problem will occur for immediate-restart, as the three transactions will collide when upgrading their read locks to write locks—only the last of the three will be able to proceed, with the other two being restarted. Fake restarts eliminate this problem, since a restarted transaction comes back as an entirely new transaction. Note that the immediate-restart algorithm has the smallest reduction in its restart ratio. This is because it has a restart delay that helps to alleviate such problems even with real restarts.

Figure 33 shows that, for the limited resource case, the fake-restart assumption again leads to higher throughput predictions for all three concurrency control algorithms. This is due to the reduced restart ratios for all three algorithms (see Figure 34). Fewer restarts lead to better throughput with limited resources, as more resources are available for doing useful (as opposed to wasted) work. For the two restart-oriented algorithms, the difference between fake and real restart performance is fairly constant over most of the range of multiprogramming levels. For blocking, however, fake restarts lead to only a slight increase in throughput at the lower multiprogramming levels. This is expected since its restart ratio is small in this region. As higher multiprogramming levels cause the restart ratio to increase, the difference between fake and real restart performance becomes large. Thus, the results produced under the fake-restart assumption in the limited resource case are biased in favor of the restart-oriented algorithms for low multiprogramming levels. At higher multiprogramming levels, all of the algorithms benefit almost equally from the fake restart assumption (with a slight bias in favor of blocking at the highest multiprogramming level).

6.2 Experiment 7: Write-Lock Acquisition

In this experiment we investigated the impact of write-lock acquisition modeling on performance. Up to now we have assumed that write locks are obtained by upgrading read locks to write locks, as is the case in many real database systems.

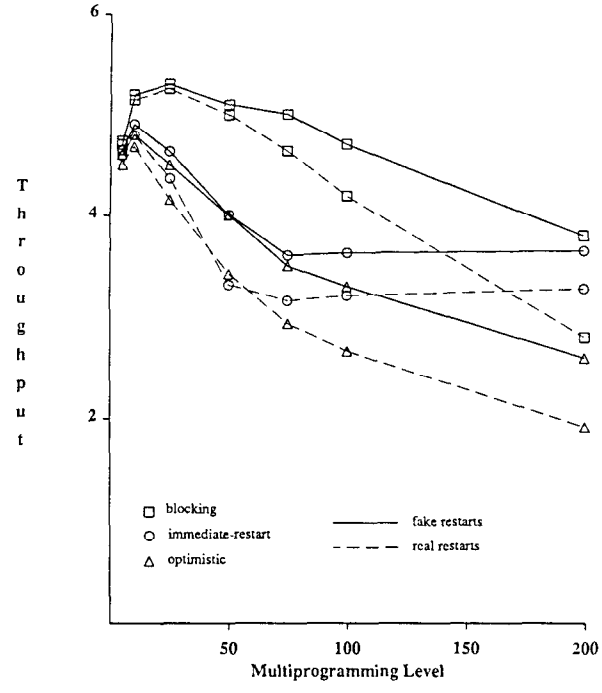

Fig. 33. Throughput (fake restarts, 1 resource unit).

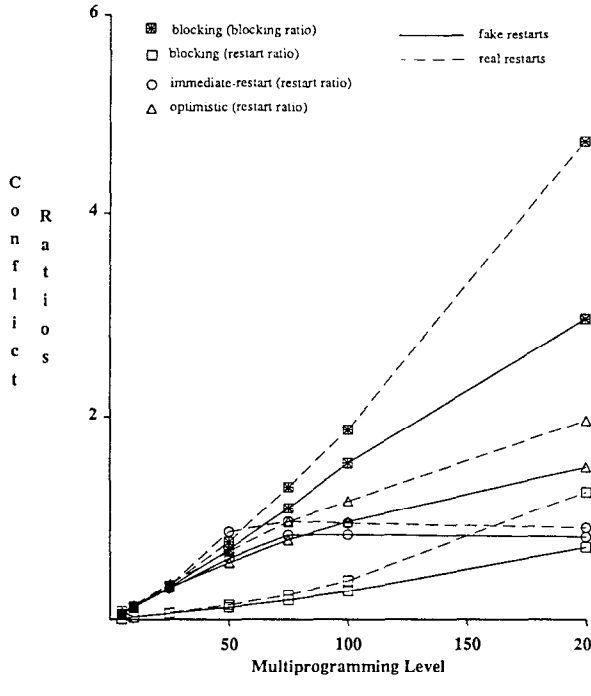

Fig. 34. Conflict ratios (fake restarts, 1 resource unit).

In this section we make an alternative assumption, the *no lock upgrades* assumption, in which a write lock is obtained instead of a read lock on each item that is to eventually be updated the first time the item is read. Figures 35 and 36 show the throughputs and conflict ratios obtained under this new assumption for the infinite resource case, and Figures 37 and 38 show the results for the limited resource case. The line and point-style conventions are the same as those in the previous experiment. Since the optimistic algorithm is (obviously) unaffected by the lock upgrade model, results are only given for the blocking and immediate-restart algorithms.

The results obtained in this experiment are quite easily explained. The upgrade assumption has little effect at the lowest multiprogramming levels, as conflicts are rare there anyway. At higher multiprogramming levels, however, the upgrade assumption does make a difference. The reasons can be understood by considering what happens when two transactions attempt to read and then write the same data item. We consider the blocking algorithm first. With lock upgrades, each transaction will first set a read lock on the item. Later, when one of the transactions is ready to write the item, it will block when it attempts to upgrade its read lock to a write lock; the other transaction will block as well when it requests its lock upgrade. This causes a deadlock, and the younger of the two transactions will be restarted. Without lock upgrades, the first transaction to lock the item will do so using a write lock, and then the other transaction will simply block without causing a deadlock when it makes its lock request. As indicated in Figures 36 and 38, this leads to lower blocking and restart ratios for the blocking algorithm under the no-lock upgrades assumption. For the immediate-restart algorithm, no restart will be eliminated in such a case, since one of the two conflicting transactions must be still restarted. The restart will occur much sooner under the no-lock upgrades assumption, however.

For the infinite resource case (Figures 35 and 36), the throughput predictions are significantly lower for blocking under the no-lock upgrades assumption. This is because write locks are obtained earlier and held significantly longer under this assumption, which leads to longer blocking times and therefore to lower throughput. The elimination of deadlock-induced restarts as described above does not help in this case, since wasted resources are not really an issue with infinite resources. For the immediate-restart algorithm, the no-lock upgrades assumption leads to only a slight throughput increase—although restarts occur earlier, as described above, again this makes little difference with infinite resources.

For the limited resource case (Figures 37 and 38), the throughput predictions for both algorithms are significantly higher under the no-lock upgrades assumption. This is easily explained as well. For blocking, eliminating lock upgrades eliminates upgrade-induced deadlocks, which leads to fewer transactions being restarted. For the immediate-restart algorithm, although no restarts are eliminated, they do occur much sooner in the lives of the restarted transactions under the no-lock upgrades assumption. The resource waste avoided by having fewer restarts with the blocking algorithm or by restarting transactions earlier with the immediate-restart algorithm leads to considerable performance increases for both algorithms when resources are limited.

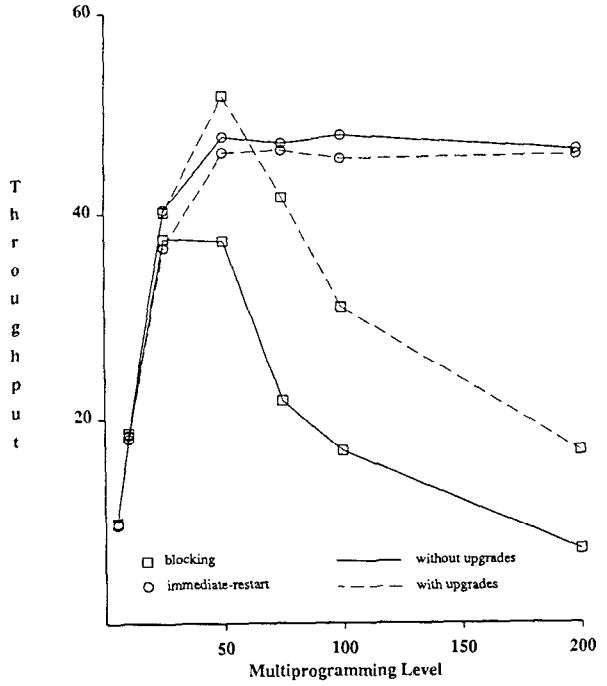

Fig. 35. Throughput (no lock upgrades, ∞ resources).

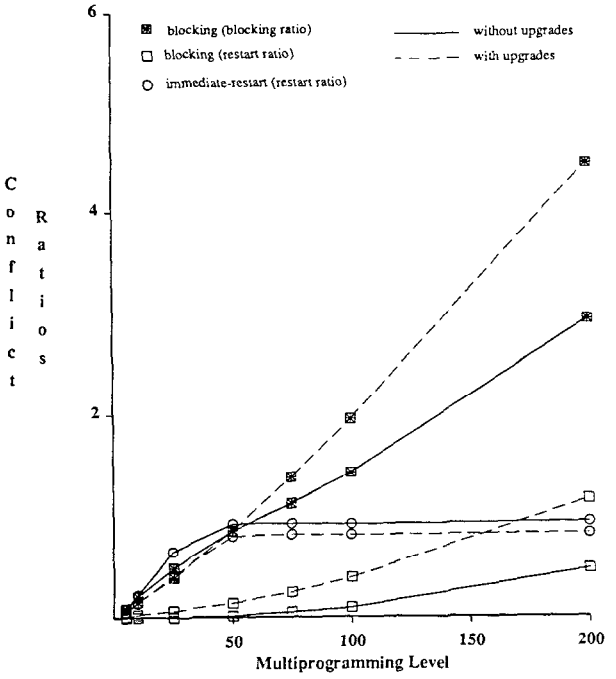

Fig. 36. Conflict ratios (no lock upgrades, ∞ resources).

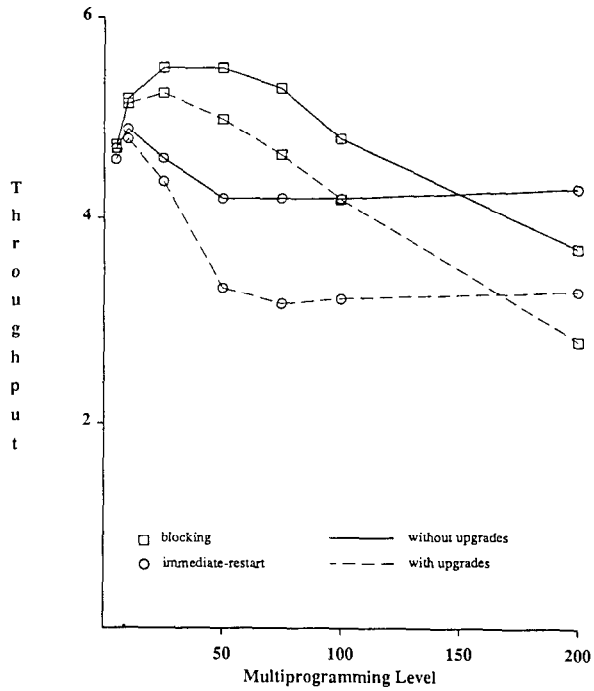

Fig. 37. Throughput (no lock upgrades, 1 resource unit).

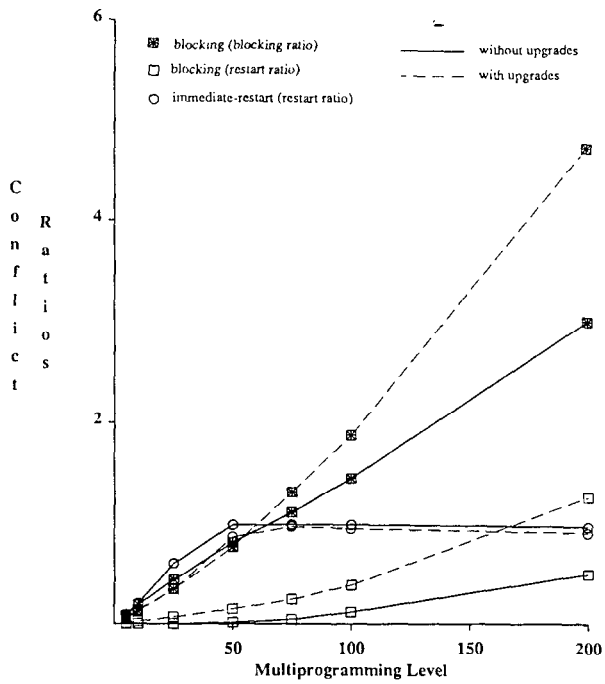

Fig. 38. Conflict ratios (no lock upgrades, 1 resource unit).

6.3 Transaction Behavior Conclusions

Reviewing the results of Experiments 6 and 7, several conclusions can be drawn. First, it is clear from Experiment 6 that the fake-restart assumption does have a significant effect on predicted throughput, particularly for high multiprogramming levels (i.e., when conflicts are frequent). In the infinite resource case, the fake-restart assumption raises the throughput of the restart-oriented algorithms more than it does for blocking, so fake restarts bias the results against blocking somewhat in this case. In the limited resource case, the results produced under the fake-restart assumption are biased in favor of the restart-oriented algorithms at low multiprogramming levels, and all algorithms benefit about equally from the assumption at higher levels of multiprogramming. In both cases, however, the relative performance results are not all that different with and without fake restarts, at least in the sense that assuming fake restarts does not change which algorithm performs the best of the three. Second, it is clear from Experiment 7 that the no-lock upgrades assumption biases the results in favor of the immediate-restart algorithm, particularly in the infinite resource case. That is, the performance of blocking is significantly underestimated using this assumption in the case of infinite resources, and the throughput of the immediate-restart algorithm benefits slightly more from this assumption than blocking does in the limited resource case.

7. CONCLUSIONS AND IMPLICATIONS

In this paper, we argued that a physically justifiable database system model is a requirement for concurrency control performance studies. We described what we feel are the key components of a reasonable model, including a model of the database system and its resources, a model of the user population, and a model of transaction behavior. We then presented our simulation model, which includes all of these components, and we used it to study alternative assumptions about database system resources and transaction behavior.

One specific conclusion of this study is that a concurrency control algorithm that tends to conserve physical resources by blocking transactions that might otherwise have to be restarted is a better choice than a restart-oriented algorithm in an environment where physical resources are limited. Dynamic two-phase locking was found to outperform the immediate-restart and optimistic algorithms for medium to high levels of resource utilization. However, if resource utilizations are low enough so 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 that allows a higher degree of concurrent execution is a better choice. We found the optimistic algorithm to perform the best of the three algorithms tested under these conditions. Low resource utilizations such as these could arise in a database machine with a large number of CPUs and disks and with a number of users similar to those of today's medium to large time-sharing systems. They could also arise in primarily interactive applications in which large think times are common and in which the number of users is such that the utilization of the system is low as a result. It is an open question whether or not such low utilizations will ever actually occur in real systems (i.e., whether

or not such operating regions are sufficiently cost-effective). If not, blocking algorithms will remain the preferred method for database concurrency control.

A more general result of this study is that we have reconfirmed results from a number of other studies, including studies reported in [1, 2, 6, 12, 15, 20, 50, and 51]. We have shown that seemingly contradictory performance results, some of which favored blocking algorithms and others of which favored restarts, are not contradictory at all. The studies are all correct within the limits of their assumptions, particularly their assumptions about system resources. Thus, although it is possible to study the effects of data contention and resource contention separately in some models [50, 51], and although such a separation may be useful in iterative approximation methods for solving concurrency control performance models [M. Vernon, personal communication, 1985], it is clear that one cannot select a concurrency control algorithm for a real system on the basis of such a separation—the proper algorithm choice is strongly resource dependent. A reasonable model of database system resources is a crucial ingredient for studies in which algorithm selection is the goal.

Another interesting 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, which controls the number of transactions that may concurrently compete for data, CPU, and I/O services (as opposed to the number of users that may be attached to the system). 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 [6, 20, 50, and 51] for locking in the low resource contention case, but more important 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, an adaptive algorithm that dynamically adjusts the multiprogramming level in order to maximize system throughput needs to be designed. Some performance indicators that might be used in the design of such an algorithm are useful resource utilization or running averages of throughput, response time, or conflict ratios. The design of such an adaptive load control algorithm is an open problem.

In addition to our conclusions about the impact of resources in determining concurrency control algorithm performance, we also investigated the effects of two transaction behavior modeling assumptions. With respect to fake versus real restarts, we found that concurrency control algorithms differ somewhat in their sensitivity to this modeling assumption; the results with fake restarts tended to be somewhat biased in favor of the restart-oriented algorithms. However, the overall conclusions about which algorithm performed the best relative to the other algorithms were not altered significantly by this assumption. With respect to the issue of how write-lock acquisition is modeled, we found relative algorithm performance to be more sensitive to this assumption than to the fake-restarts

assumption. The performance of the blocking algorithm was particularly sensitive to the no-lock upgrades assumption in the infinite resource case, with its throughput being underestimated by as much as a factor of two at the higher multiprogramming levels.

In closing, we wish to leave the reader with the following thoughts about computer system resources and the future, due to Bill Wulf:

> 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. [54, p. 41]

ACKNOWLEDGMENTS

The authors wish to acknowledge the anonymous referees for their many insightful comments. We also wish to acknowledge helpful discussions that one or more of us have had with Mary Vernon, Nat Goodman, and (especially) Y. C. Tay. Comments from Rudd Canaday on an earlier version of this paper helped us to improve the presentation. 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

1. AGRAWAL, R. Concurrency control and recovery in multiprocessor database machines: Design and performance evaluation, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisc., 1983.
2. AGRAWAL, R., AND DEWITT, D. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 529–564.
3. AGRAWAL, R., CAREY, M., AND DEWITT, D. Deadlock detection is cheap. *ACM-SIGMOD Record 13*, 2 (Jan. 1983).
4. AGRAWAL, R., CAREY, M., AND MCVOY, L. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Trans. Softw. Eng.* To be published.
5. BADAL, D. Correctness of concurrency control and implications in distributed databases. In *Proceedings of the COMPSAC '79 Conference* (Chicago, Nov. 1979). IEEE, New York, 1979, pp. 588–593.
6. BALTER, R., BERARD, P., AND DECITRE, P. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In *Proceedings of the 1st ACM SIGACT SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Aug. 18–20, 1982). ACM, New York, 1982, pp. 183–193.
7. BERNSTEIN, P., AND GOODMAN, N. Fundamental algorithms for concurrency control in distributed database systems. Tech. Rep., Computer Corporation of America, Cambridge, Mass., 1980.
8. BERNSTEIN, P., AND GOODMAN, N. "Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1980), pp. 285–300.
9. BERNSTEIN, P., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–222.
10. BERNSTEIN, P., AND GOODMAN, N. A sophisticate's introduction to distributed database concurrency control. In *Proceedings of the 8th International Conference on Very Large Data Bases* (Mexico City, Sept. 1982), pp. 62–76.
11. BERNSTEIN, P., SHIPMAN, D., AND WONG, S. Formal aspects in serializability of database concurrency control. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979).

12. CAREY, M. Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, Sept. 1983.
13. CAREY, M. An abstract model of database concurrency control algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Jose, Calif., May 23–26, 1983). ACM, New York, 1983, pp. 97–107.
14. CAREY, M., AND MUHANNA, W. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst. 4*, 4 (Nov. 1986), 338–378.
15. CAREY, M., AND STONEBRAKER, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984), pp. 107–118.
16. CASANOVA, M. The concurrency control problem for database systems. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass. 1979.
17. CERI, S., AND OWICKI, S. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Calif., Feb. 1982), ACM, IEEE, New York, 1982.
18. ELHARD, K., AND BAYER, R. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 503–525.
19. ESWAREN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
20. FRANASZEK, P., AND ROBINSON, J. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst. 10*, 1 (Mar. 1985), 1–28.
21. GALLER, B. Concurrency control performance issues. Ph.D. dissertation, Computer Science Department, University of Toronto, Ontario, Sept. 1982.
22. GOODMAN, N., SURI, R., AND TAY, Y. A simple analytic model for performance of exclusive locking in database systems. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23, 1983). ACM, New York, 1983 pp. 203–215.
23. GRAY, J. Notes on database operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, 1979.
24. GRAY, J., HOMAN, P., KORTH, H., AND OBERMARCK, R. A straw man analysis of the probability of waiting and deadlock in a database system. Tech. Rep. RJ3066, IBM San Jose Research Laboratory, San Jose, Calif., Feb. 1981.
25. HAERDER, T., AND PEINL, P. Evaluating multiple server DBMS in general purpose operating system environments. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug. 1984).
26. IRANI, K., AND LIN, H. Queuing network models for concurrent transaction processing in a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, May 30–June 1, 1979). ACM, New York, 1979.
27. KUNG, H., AND ROBINSON, J. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
28. LIN, W., AND NOLTE, J. Distributed database control and allocation: Semi-annual report. Tech. Rep., Computer Corporation of America, Cambridge, Mass., Jan. 1982.
29. LIN, W., AND NOLTE, J. Performance of two phase locking. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Feb. 1982), ACM, IEEE, New York, 1982, pp. 131–160.
30. LIN, W., AND NOLTE, J. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983).
31. LINDSAY, B., ET AL. Notes on distributed databases, Tech. Rep. RJ2571, IBM San Jose Research Laboratory, San Jose, Calif., 1979.
32. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed databases. In *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, Aug. 1978). ACM, IEEE, New York, 1978, pp. 215–232.
33. PAPADIMITRIOU, C. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
34. PEINL, P., AND REUTER, A. Empirical comparison of database concurrency control schemes. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983), pp. 97–108.

35. POTIER, D., AND LEBLANC, P. Analysis of locking policies in database management systems. *Commun. ACM 23*, 10 (Oct. 1980), 584–593.
36. REED, D. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1978.
37. REUTER, A. An analytic model of transaction interference in database systems. IB 68/83, University of Kaiserslautern, West Germany, 1983.
38. REUTER, A. Performance analysis of recovery techniques. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 526–559.
39. RIES, D. The effects of concurrency control on database management system performance. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, Calif., 1979.
40. RIES, D., AND STONEBRAKER, M. Effects of locking granularity on database management system performance. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
41. RIES, D., AND STONEBRAKER, M. Locking granularity revisited. *ACM Trans. Database Syst. 4*, 2 (June 1979), 210–227.
42. ROBINSON, J. Design of concurrency controls for transaction processing systems. Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 1982.
43. ROBINSON, J. Experiments with transaction processing on a multi-microprocessor. Tech. Rep. RC9725, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1982.
44. ROSENKRANTZ, D., STEARNS, R., AND LEWIS, P., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
45. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. In *The INGRES Papers: Anatomy of a Relational Database System*, M. Stonebraker, Ed. Addison-Wesley, Reading, Mass. 1986.
46. SARGENT, R. Statistical analysis of simulation output data. In *Proceedings of the 4th Annual Symposium on the Simulation of Computer Systems* (Aug. 1976), pp. 39–50.
47. SPITZER, J. Performance prototyping of data management applications. In *Proceedings of the ACM '76 Annual Conference* (Houston, Tx., Oct. 20–22, 1976). ACM, New York, 1976, pp. 287–292.
48. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979).
49. STONEBRAKER, M., AND ROWE, L. The Design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 28–30, 1986). ACM, New York, 1986, pp. 340–355.
50. TAY, Y. A mean value performance model for locking in databases. Ph.D. dissertation, Computer Science Department, Harvard University, Cambridge, Mass. Feb. 1984.
51. TAY, Y., GOODMAN, N., AND SURI, R. Locking performance in centralized databases. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 415–462.
52. THOMAS, R. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
53. THOMASIAN, A., AND RYU, I. A decomposition solution to the queuing network model of the centralized DBMS with static locking. In *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Minneapolis, Minn., Aug. 29–31, 1983). ACM, New York, 1983, pp. 82–92.
54. WULF, W. Compilers and computer architecture. *IEEE Computer* (July 1981).

Received August 1985; revised August 1986; accepted May 1987