The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems

RAKESH AGRAWAL, MICHAEL J. CAREY, MEMBER, IEEE, AND LAWRENCE W. McVOY

Abstract—**There is growing evidence that, for a fairly wide variety of database workloads and system configurations, locking is the concurrency control strategy of choice. With locking, of course, comes the possibility of deadlocks. Although the database literature is full of algorithms for dealing with deadlocks, very little in the way of practical performance information is available to a database system designer faced with the decision of choosing a good deadlock resolution strategy. This paper is an attempt to bridge this gap in our understanding of the behavior and performance of alternative deadlock resolution strategies. We employ a simulation model of a database environment to study the relative performance of several strategies based on deadlock detection, several strategies based on deadlock prevention, and a strategy based on timeouts. We show that the choice of the best deadlock resolution strategy depends upon the level of data contention, the resource utilization levels, and the types of transactions. We provide guidelines for selecting a deadlock resolution strategy for different operating regions.**

Index Terms—**Concurrency control, database systems, deadlock, modeling and simulation, transaction processing.**

I. INTRODUCTION

IT was shown in a recent performance study that, for a fairly wide variety of database workloads and system configurations, a concurrency control algorithm based on two-phase locking [8], [9], [15], [18] will outperform concurrency control algorithms that use transaction restarts rather than blocking to resolve conflicts between transactions [2]. This result was found to be particularly true under models of realistic hardware configurations with limited physical resources. With two-phase locking, of course, comes the possibility of deadlock [8], [9], [18]. Since locking realizes consistency by blocking transactions whose lock requests cannot be granted, two (or more) transactions may hold locks needed by each other, consequently blocking each other's execution forever.

Given that locking is the concurrency control strategy of choice, it is imperative that the problem of deadlock and the performance of various deadlock resolution strategies be well understood. The database literature is full of algorithms that either detect and resolve deadlocks or somehow prevent deadlocks from happening [1], [5]-[7], [12], [13], [16]-[18], [22], [23], [25]-[29], [31], [32], [37], [39]. There have been several analytical studies devoted to estimating the probability of deadlocks [19], [33], [40], [41], but very little in the way of practical performance information is available to the database system designer who is faced with the decision of choosing a good deadlock resolution strategy. The only significant performance studies of deadlock resolution strategies that we have seen are an early study by Munz and Krenz [30] and a more recent study by Balter, Berard, and Decitre [5]. In [30], eleven methods for selecting a transaction to restart once a deadlock has been detected were compared. Although the study involved some 600 simulation runs, the paper contains little in the way of details about the simulation model and the range of parameters considered. In [5], five deadlock resolution strategies were compared using simulation. Their simulation model description is also somewhat vague, and no details are given regarding the simulation parameter values used in the study. In addition, they did not model disk and CPU resources or their limits in any detail in their simulation model. Instead, they simply assumed that all transactions can be processed in parallel in the absence of concurrency control conflicts, an assumption that has since been shown to have a significant influence on the conclusions of concurrency control performance studies [2]. We will look more closely at the results of these two studies later on.

The intent of this paper is to bridge the gap in our understanding of the behavior and performance of alternative strategies for dealing with the problem of deadlock in centralized database systems. We begin by establishing a reasonably complete simulation model of a database management system. Our model captures the main elements of a database environment, including both users (terminals, the source of transactions) and physical resources for storing and processing the data (disks and CPU's) in addition to the usual components used in models for studying deadlock (workload and database characteristics). We then study and compare the relative performance of three different classes of deadlock resolution strategies [8], [18], including several strategies based on deadlock detection, several strategies based on deadlock prevention, and a strategy that deals with the problem through the use of timeouts. Our study examines the relative performance of these strategies over a wide range of multiprogramming levels (and thus conflict prob-

Manuscript received June 28, 1985. The work of M. J.Carey was supported in part by the National Science Foundation under Grant DCR-8402818, by an IBM Faculty Development Award, and by the Wisconsin Alumni Research Foundation.

R. Agrawal is with AT&T Bell Laboratories, Murray Hill, NJ 07974.

M. J. Carey and L. W. McVoy are with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8717408.

abilities), and both noninteractive and interactive workload types are considered.

The organization of the remainder of this paper is as follows. In Section II we describe the alternative deadlock resolution strategies that are examined in this paper. Our performance study is based on simulations of a closed queuing model of a single-site database system. The structure and characteristics of our simulator are described in Section III. Section IV presents the performance experiments and our results. Finally, Section V summarizes the main conclusions of this study.

II. Deadlock Resolution Strategies

Deadlocks are usually characterized in terms of a waits-for graph [14], [18], [21]. In a database context, a waits-for graph G is a directed graph where each vertex represents a transaction; each edge of the form $T_i \rightarrow T_j \in G$ means that transaction T_i is waiting for transaction T_j (e.g., because T_j owns a lock that T_i wants). It has been shown that there exists a deadlock if and only if there is a cycle in the waits-for graph [14], [21].

We classify deadlock resolution strategies into three broad classes based on whether they require explicit maintenance of the waits-for graph and on whether or not a transaction that is not necessarily involved in a deadlock may be restarted for deadlock resolution purposes. These three classes of deadlock resolution strategies, *detection*, *prevention*, and *timeout* strategies, are described in more detail in this section. We also describe the particular instances of these three classes of strategies whose performance is studied in this paper. Before describing the various strategies of interest, however, we elaborate briefly on the issue of how waits-for information is maintained.

In addition to the waits-for graph described above, a typical lock manager maintains a lock table for keeping track of locks. This table contains an entry for every locked object, including such information as a list of its current lockers, the mode that it is locked in, and a queue of transactions that are waiting to obtain a lock on the object [18]. To illustrate how waits-for information is derived from these entries, suppose that transaction T_1 holds a read lock on X, T_2 is waiting for a write lock on X, T_3 and T_4 are waiting behind T_2 for read locks on X,[1] and T_5 is at the back of the queue waiting for a write lock on X. This situation would be represented in the waits-for graph as the set of edges $\{T_2 \rightarrow T_1, T_3 \rightarrow T_2, T_4 \rightarrow T_2, T_5 \rightarrow T_3, T_5 \rightarrow T_4\}$. That is, each member of X's waiting queue is waiting for the first transaction or group of transactions in front of it with which it conflicts.

A. Deadlock Detection

Strategies based on deadlock detection require that the waits-for graph be explicitly built and maintained. In *continuous detection*, the waits-for graph is always kept cycle-free by checking for (and breaking) cycles every time a transaction blocks. The connected component of the waits-for graph involving the newly blocked transaction is searched using a cycle detection algorithm [4]. In *periodic detection*, the graph is searched only periodically for cycles, and all connected components of the graph must be searched in this case. Selecting an appropriate time interval for periodic detection is a problem, and we will study the effects of different time intervals for periodic deadlock detection.

An associated problem is that of selecting a transaction or set of transactions to restart in order to break any deadlock cycles that form. We will consider the following *victim selection* criteria:

1) Current Blocker: Pick the transaction that blocked the most recently (i.e., the one that just blocked, the current blocker, in the case of continuous detection).

2) Random Blocker: Pick a transaction at random from among the participants in the deadlock cycle.

3) Min Locks: Pick the transaction that is holding the fewest locks.

4) Youngest: Pick the transaction with the most recent initial startup time (i.e., the one that began running the most recently).

5) Min Work: Pick the transaction that has consumed the least amount of physical resources (CPU + I/O time) since it first began running.

The first of these five victim selection criteria picks the most convenient transaction to restart in the continuous detection case, and the second criterion simply picks any transaction. The latter three criteria for victim selection attempt to restart the least expensive transaction, the difference among these criteria being the metric used by each to predict expensiveness. Following the lead in [30], we have not considered any victim selection criteria that selects the most expensive transaction, such as the transaction that is holding the largest number of locks or that has consumed the largest amount of physical resources. It was shown in [30] that the least expensive counterparts of these criteria always have better performance.

One detail needs to be mentioned at this point. It is possible that multiple cycles may be encountered in the waits-for graph, in which case more than one victim may be selected. Our implementation of deadlock detection follows the advice given by Gray in [18], which suggests repeatedly selecting a victim from each detected cycle according to the victim selection criteria until the graph is free of cycles. This approach was suggested by Gray (and adopted for our use) because the problem of selecting the least expensive set of transactions to break all cycles is NP-hard [18].

B. Deadlock Prevention

In deadlock prevention, the waits-for graph is not explicitly maintained. Deadlocks are prevented by never allowing blocked states that can lead to circular waiting. We consider the following set of deadlock *prevention* algorithms:

[1] Fairness dictates that, even though the lock requests from T_3 and T_4 are compatible with the current lock on X, they must wait until T_2's request has been granted because their requests were made after T_2 made its request. Otherwise, read requests can lead to starvation of write requests [18].

1) Wound-Wait: If a lock request from transaction T_i leads to a conflict with another transaction T_j, resolve the conflict as follows. If T_i started running before T_j did, then restart T_j ("T_i wounds T_j"); otherwise, block T_i ("T_i waits for T_j"). This algorithm is due to Rosenkrantz, Stearns, and Lewis [37]. Deadlocks cannot occur, as a transaction can be blocked only by an older transaction, and therefore cycles cannot form in the waits-for graph. Thus, wound-wait is a preemptive algorithm in which older transactions run through the system killing any younger ones that they conflict with, waiting only for older conflicting transactions.

2) Wait-Die: If a lock request from transaction T_i leads to a conflict with another transaction T_j, resolve the conflict as follows. If T_i started up earlier than T_j, then block T_i ("T_i waits for T_j"); otherwise, restart T_i ("T_i dies"). This algorithm is also due to Rosenkrantz, Stearns, and Lewis [37]. Again, deadlocks are impossible, because a transaction can only be blocked by a younger transaction. Wait-die is a nonpreemptive counterpart to the wound-wait deadlock prevention algorithm.

3) Immediate-Restart: If a lock request from transaction T_i leads to a conflict with another transaction T_j, then simply restart T_i. In this algorithm, due to Tay [40], [41], deadlocks are impossible because no transaction is ever blocked.

4) Running-Priority: If a lock request from transaction T_i leads to a conflict with another transaction T_j, resolve the conflict as follows. If T_j is currently waiting due to some other conflict, then restart T_j and grant T_i's lock request; otherwise, if T_j is active, then block T_i. This algorithm is due to Franaszek and Robinson [16], and the idea is not to allow blocked transactions to impede the progress of active transactions. In this algorithm, then, a transaction is blocked only when waiting for an active transaction—it is similar in nature to wound-wait, except that running (instead of older) transactions are favored. Deadlock cycles are impossible because no transaction ever waits for another waiting transaction.

Observe that in deadlock prevention, a restarted transaction is not necessarily involved in an actual deadlock cycle. Deadlock prevention policies are thus conservative in nature, trading more restarts for the purpose of preventing possible deadlocks.

C. Timeout

In the case of the *timeout* strategy for dealing with deadlocks, a transaction whose lock request cannot be granted is simply placed in the blocked queue. The transaction is later restarted if its wait time exceeds some threshold value. Thus, like the detection strategies, timeout will restart transactions that are actually involved in deadlocks; like the prevention strategies, timeout may also restart some transactions that are not involved in any cycle (when the transaction has been waiting a long time but no cycle of waiting transactions actually exists). As we will see in Section IV, selecting an appropriate threshold value is a problem with the timeout approach. We will examine the effect of different threshold values on the performance of this strategy.

III. Simulation Model

Our simulator for studying the performance of the deadlock resolution strategies is based on the closed queueing model of a single-site database system depicted in Fig. 1. This model is the same simulation model that was used in [2], which is a model that has its origins in the models of [34]–[36] and [10], [11]. 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 lock requests. If the lock request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object is to be accessed, the transaction reenters the concurrency control queue and makes its next request prior to accessing the object. It is assumed for modeling convenience that a transaction performs all of its reads before performing any writes. We also examine the performance of deadlock resolution strategies under interactive workloads. The think path in the model provides an optional random delay that follows object accesses for this purpose.

If one of the transaction's lock requests leads to a lock conflict, the concurrency control module takes actions that are dependent upon the deadlock resolution strategy being used. For example, in the case of continuous deadlock detection, it checks whether blocking the current transaction will result in a cycle in the waits-for graph; if so, then the victim-selection criteria is applied to select the transaction(s) to be restarted. In the case of periodic deadlock detection, the transaction is simply added to the waits-for graph, with no check for cycles in the graph being made at this time. In the case of deadlock prevention, no waits-for graph is maintained; a decision is taken either to block the current transaction or to restart some transaction based on the prevention scheme being employed. In the case of timeout, the current transaction is always simply blocked.

If the result of a lock 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 a transaction, the restarted transaction goes to the back of the ready queue after a restart delay period. It then begins making all of the *same* concurrency control requests and object accesses over again. The duration of the restart delay is exponential with a mean equal to the running average of the transaction response times—that is, the duration of the delay is *adaptive*, depending on the observed

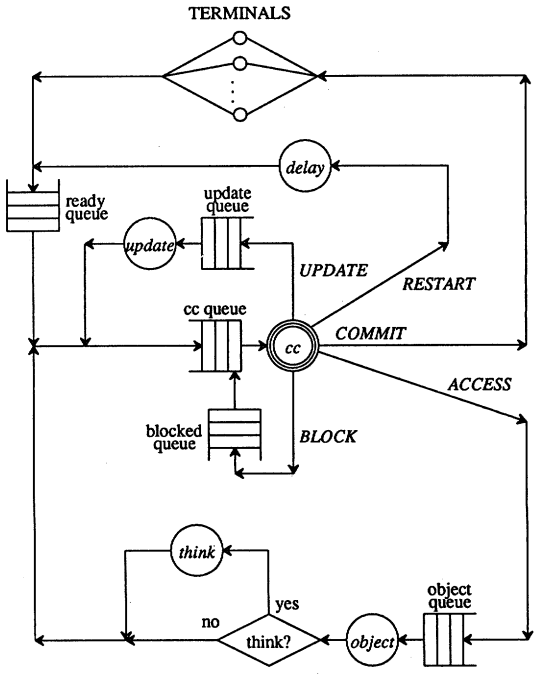


Fig. 1. Logical queueing model.

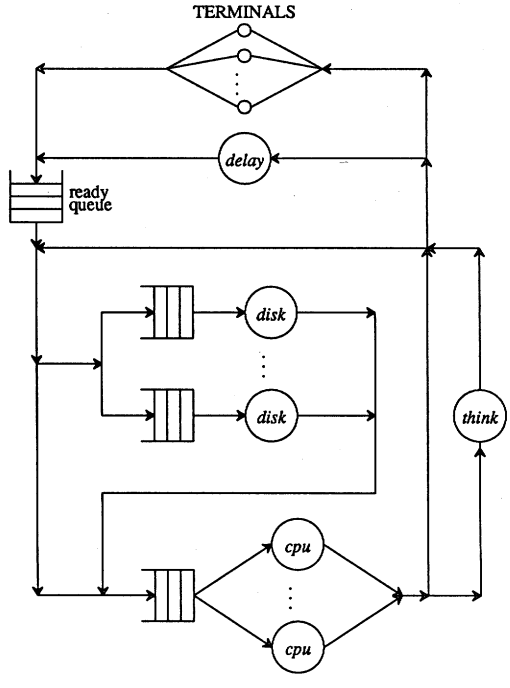


Fig. 2. Physical queueing model.

average response time. We chose to employ an adaptive delay after performing a sensitivity analysis that showed us that the performance of deadlock resolution strategies is sensitive to the restart delay time. Our experiments indicated that a delay of about one transaction time is best, and that throughput begins to drop off rapidly when the delay exceeds more than a few transaction times. To model periodic deadlock detection, an event is scheduled at fixed time intervals that checks for cycles in the waits-for graph and selects a victim (or victims) to break any and all cycles. To model timeouts, a timeout event is scheduled each time a transaction is blocked. If the blocked transaction is still blocked at the end of the threshold time period, it is restarted.

Eventually, when the transaction completes, one of two things happens. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, the transaction first enters the *update queue* and writes its deferred updates into the database. Locks are released together at end-of-transaction after any deferred updates have been performed.

Underlying the logical model of Fig. 1 are two physical resource types, CPU resources and I/O (disk) resources. Associated with the concurrency control, object access, and deferred update services in Fig. 1 are some use of one or both of these two resource types. The amounts of CPU and I/O time per logical service are specified as simulation parameters. The physical queueing model is depicted in Fig. 2, and Table I summarizes its associated simulation parameters. As shown, the physical model is a collection of terminals, CPU servers, and I/O servers. (While the model permits an arbitrary number of each class of server, as shown in Fig. 2, our experiments will all be based on a configuration with the one CPU server and two

TABLE I
SIMULATION 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)
num_terms	number of terminals
mpl	multiprogramming level
ext_think_time	mean time between transactions (per terminal)
int_think_time	mean intra-transaction think time (optional)
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	number of CPU servers
num_disks	number of I/O servers

I/O servers. The sufficiency of this configuration for our purposes will be addressed in Section IV.) The delay paths for the think and restart delays are also reflected in the physical queuing model. Requests in the CPU queue are served FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. Our I/O model is that of a database where the data is spread out evenly across all of the disks. There is a queue associated with each of the I/O servers; when a transaction needs service, it chooses an I/O server (at random, with all I/O servers being equally likely) and waits in the queue associated with the selected server. The service discipline for the I/O queues is FCFS.

The parameters *obj_io* and *obj_cpu* in Table I are the amounts of I/O and CPU time associated with reading or writing an object. Reading an object takes resources equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and *obj_io* at deferred update time, as it is assumed that transactions maintain deferred update lists in buffers in main memory. These parameters represent constant

service time requirements rather than stochastic ones for simplicity. The *ext_think_time* parameter is the mean of an exponential time distribution which determines the time delay between the completion of a transaction from a terminal. Finally, the *int_think_time* parameter is the mean of an exponential time distribution which determines the intratransaction think time for the model (if any). To model interactive workloads, transactions can be made to undergo a thinking period between finishing their reads and starting their writes.

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*.

IV. Performance Experiments

We ran a number of simulation experiments to study the performance of the alternative deadlock resolution strategies described in Section II. We present the experiments and their results in this section. Experiment 1 examines the performance of the different victim selection criteria for continuous deadlock detection. Experiment 2 studies periodic deadlock detection, looking at how the detection interval size affects its performance and at how it compares to continuous deadlock detection. Experiment 3 examines the use of timeouts for handling deadlocks, looking at the question of how to choose an appropriate timeout interval. Finally, Experiment 4 compares the performance of the best deadlock detection alternative to that of the timeout strategy and the four deadlock prevention strategies (wound-wait, wait-die, immediate-restart, and running-priority). At the end of this section we discuss how our results on alternative deadlock resolution schemes compare with those reported by others.

Table II gives the simulation values used for the experiments reported here. The database and transaction sizes were selected so as to jointly yield a region of operation which allows the interesting performance effects to be observed without necessitating impossibly long simulation times. These sizes are expressed in pages, as we equate objects and pages in this study. The multiprogramming level is varied between a limit of 5 transactions and a limit of the total number of terminals,[2] set to 200 in this study, to allow a range of conflict probabilities to be investigated. The object processing costs were chosen based on our notion of roughly what realistic values might be. We employed a modified form of the batch means method [38]

TABLE II
Simulation Parameter Settings

Parameter	Value
db_size	1000 pages
tran_size	8 page readset
max_size	12 page readset (maximum)
min_size	4 page readset (minimum)
write_prob	0.25
num_terms	200 terminals
mpl	5, 10, 25, 50, 75, 100, and 200 transactions
ext_think_time (non-interactive)	1 second
int_think_time (non-interactive)	0 seconds
ext_think_time (interactive)	21 seconds
int_think_time (interactive)	10 seconds
obj_io	35 milliseconds
obj_cpu	15 milliseconds
num_cpus	1
num_disks	2

for our statistical data analyses, 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 the throughput confidence intervals were typically in the range of plus or minus a few percent of the mean value, more than sufficient for our purposes. We discuss only the statistically significant performance differences when summarizing our results.

We investigated the performance of the algorithms in a configuration with one CPU server and two I/O servers. We consider two types of transaction workloads, a noninteractive workload, in which transactions have no intratransaction think time, and an interactive workload, in which transactions perform a number of reads, then think for some period of time (their "internal think," during which read locks are held), and then perform their writes. Besides being of interest in their own right, we found in [2] that an interactive transaction mix with a large intratransaction (internal) think time causes a system with finite physical resources (CPU's and disks) to behave as if the system had infinite resources. Based on the results in [2], it is safe to say that these two workload types are representative of the results that would be obtained over a wide range of resource configurations (i.e., numbers of CPU and I/O servers) and over a wide range of think times. Briefly, it was found that when the "useful" utilization[4] of the bottleneck resource type is moderately high (60–70 percent and above), the noninteractive workload results are good indicators of relative performance; for low utilizations (40–50 percent and below), interactive workloads with high intratransaction think times are good indicators of relative performance (regardless of why the utilizations are low, whether due to an interactive workload or to large numbers of CPU's and disks). We chose the interactive transaction approach to approximating infinite resource behavior for this study because it is very expensive to run simulations for system configurations that have many CPU and I/O servers (or truly infinite resources).

[2]Keep in mind that the multiprogramming level simply serves to bound the number of active transactions in the system, not to strictly determine it. It is likely that fewer than *mpl* transactions will actually be ready to run when *mpl* = *num_terms* because of the external think time for the terminals.

[3]More information on the details of the modified batch means method may be found in [10].

[4]We will explain the difference between total and useful resource utilization in Section IV-D.

We also found in [2] that for larger values of the database (for 10,000 objects), conflicts became very rare; when conflicts are rare, all concurrency control algorithms perform alike (as shown in [2], [3], [10], [11] and a number of other studies). Since we were interested in investigating *differences* in the performance of alternative deadlock resolution strategies, we have not varied the size of the database as a simulation parameter. The size used here provides a wide range of conflict probabilities over the large range of multiprogramming levels used. Also, we feel that it is important to study workloads with non-negligible conflict levels in order to understand how the algorithms will perform under heavy conflict workloads or when "hot spots" exist in the database. Finally, we have also not varied transaction size as a simulation parameter here because varying the multiprogramming level for a fixed size database provides a similar range of conflict levels of interest.

A. Experiment 1: Continuous Detection

Our first experiment examined the effectiveness of the alternative victim selection criteria for continuous deadlock detection. Figs. 3 and 4, respectively, show the throughputs (in transactions per second) obtained for the five criteria for the noninteractive type of transactions, where the resources are highly utilized, and for the interactive workload, where the resources have low utilizations. Although we also recorded the corresponding response time results, we omit them—they display the same relative trends, conveying no important additional information.

In both the noninteractive and interactive cases, for low levels of multiprogramming, the performance of the different victim selection criteria is almost identical. There are just not many deadlocks to differentiate the criteria. However, as the multiprogramming level is increased, the *random* selection criterion begins to lose out; so does the *current blocker* criterion, which is also a random selection in some sense. The *youngest*, *min locks*, and *min work* selection criteria, which attempt to restart the least expensive transaction, begin providing better performance as the multiprogramming level is increased. The reader may be surprised at first, as we were, by the superior performance of the *min locks* selection criterion as compared to the *min work* criterion. The explanation lies in the fact that a restarted transaction that is holding a large number of locks, although it may have consumed somewhat fewer resources as compared to some other transactions with fewer locks, would have to contend for all of its locks again—with the possibility of getting blocked or restarted at each lock request.[5] Figs. 5 and 6 give the blocking ratio (the ratio of blocked lock requests to the number of transactions executed) and the restart ratio (the ratio of restarted transactions to the number of transactions executed) for the interactive workload case. Note that the *min locks* blocking and restart ratios are the lowest among all of the victim selection criteria. Similar trends were seen in the noninteractive workload case.

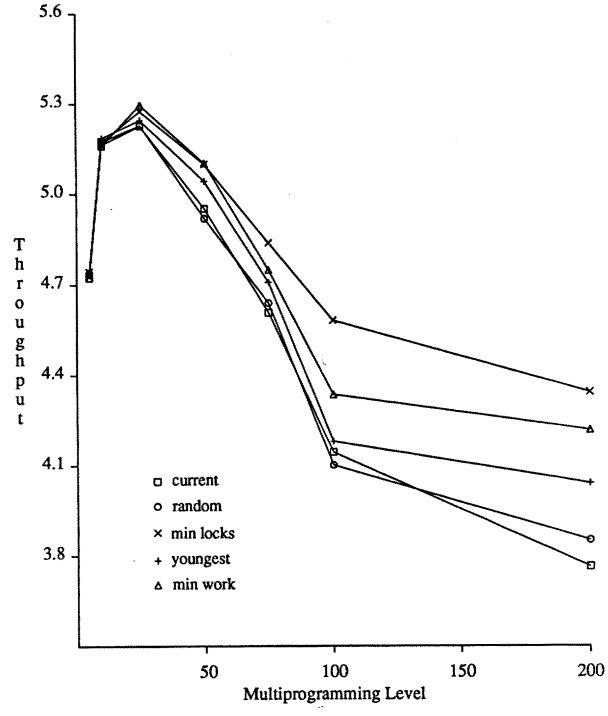


Fig. 3. Throughput, continuous detection (noninteractive workload).

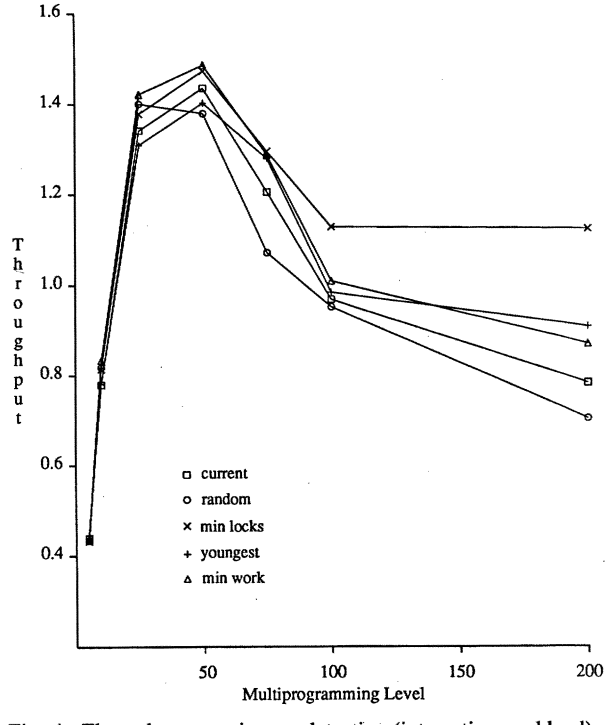


Fig. 4. Throughput, continuous detection (interactive workload).

In examining the results of this experiment, we came to the conclusion that there are several *stability* properties that a given deadlock resolution mechanism might have,

[5]It should be noted here that all objects in our database model have the same granularity. In a system that supports granularity hierarchies, such as System R [18], the simple lock counting scheme would probably have to be modified to account for the number of objects associated with each lock.

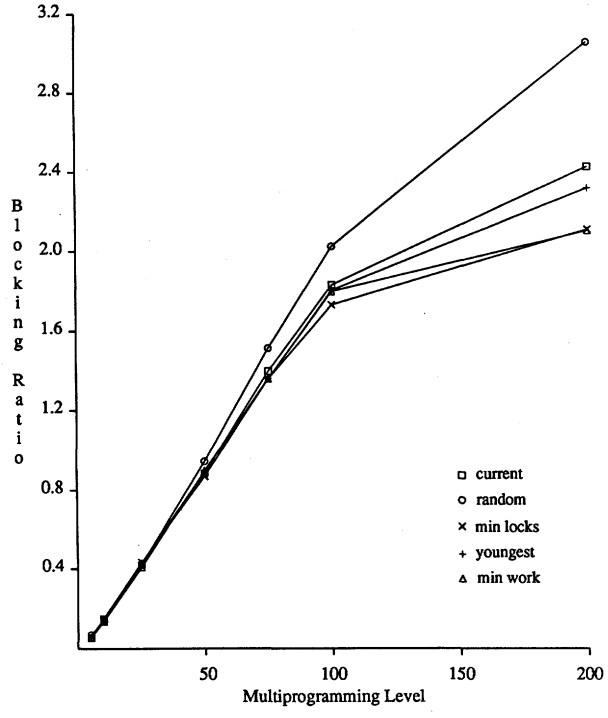


Fig. 5. Blocking ratio, continuous detection (interactive workload).

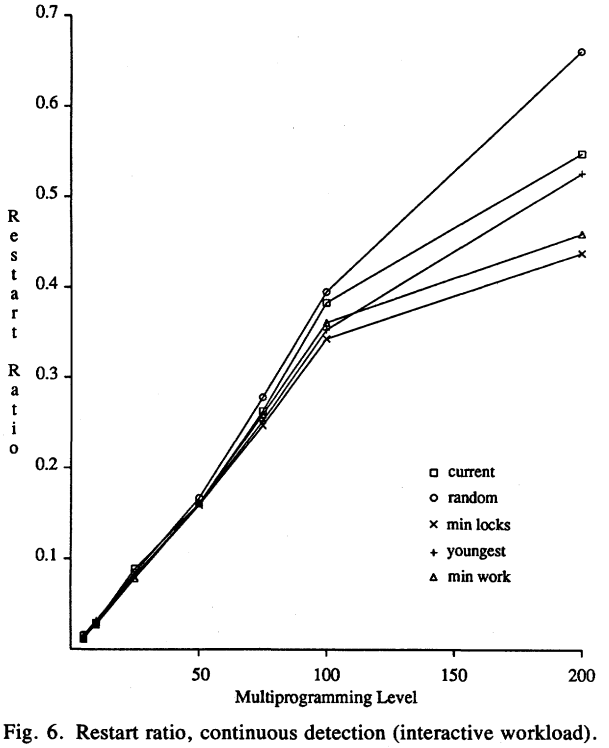


Fig. 6. Restart ratio, continuous detection (interactive workload).

and we hypothesize that these properties are desirable ones:

Guaranteed forward progress. At any time, there is at least one transaction in the system that can be guaranteed to finish.

No repeated restarts: No one transaction may be restarted over and over again an indefinite number of times.

While having these properties will not guarantee superior performance for an algorithm, the absence of these properties may make an algorithm unstable and therefore likely to perform poorly in high conflict situations. Our results bear out this hypothesis, at least to some extent. Of the victim selection criteria studied here, only the *youngest* criterion has these two stability properties—the oldest transaction in the system is ensured of being able to finish without being restarted, and (by induction) every transaction will eventually become the oldest one and then finish. As a result, the *youngest* criterion performs moderately well. Both the *random* and *current blocker* criteria lack both stability properties. With these criteria, any transaction with at least one lock request remaining can be restarted when it makes the request, and they do nothing to protect a transaction from being repeatedly restarted (particularly if two or more transactions are attempting to read and then write the same object). Consequently, the *random* and *current blocker* criteria have inferior performance at high multiprogramming levels. The *min work* and *min locks* criteria do not have these stability properties in the strict sense. This is because it is possible for the transaction with the most locks or having consumed the most resources to be "passed" by another transaction. However, they do have *pseudo-stability*—the number of locks held and the amount of resources consumed are likely to be closely correlated with the age of the transaction. Also, they seem to be probabilistically stable; for example, the transaction with the most locks is unlikely to be passed by many other transactions, so it is fairly safe from being restarted using *min locks* (like an old transaction in the *youngest* policy).

B. Experiment 2: Periodic Detection

As we mentioned in Section II, a problem with periodic deadlock detection is determining the time interval at which the detection algorithm should be periodically performed. Experiment 2 considered the choice of a deadlock detection interval and also compared the performance of periodic detection (with a good interval) to that of continuous detection. Based on the results of Experiment 1, we used the *min locks* victim selection policy here for both continuous and periodic detection. We found in several preliminary experiments (not reported here) that the choice of the best victim selection criteria is independent of whether continuous or periodic detection is performed, which is not surprising.

1) Deadlock Detection Interval: The deadlock detection interval selection portion of Experiment 2 was performed for a low level of multiprogramming (10) and for a high level of multiprogramming (100). Both the high resource utilization (noninteractive workload) and the low resource utilization (interactive workload) cases were considered. The throughput results are summarized in Fig. 7. As the detection interval is increased, performance degrades significantly, particularly for large multiprogramming levels. This is because deadlocked transactions that are blocked while holding locks required by other transactions increase contention further if they are restarted

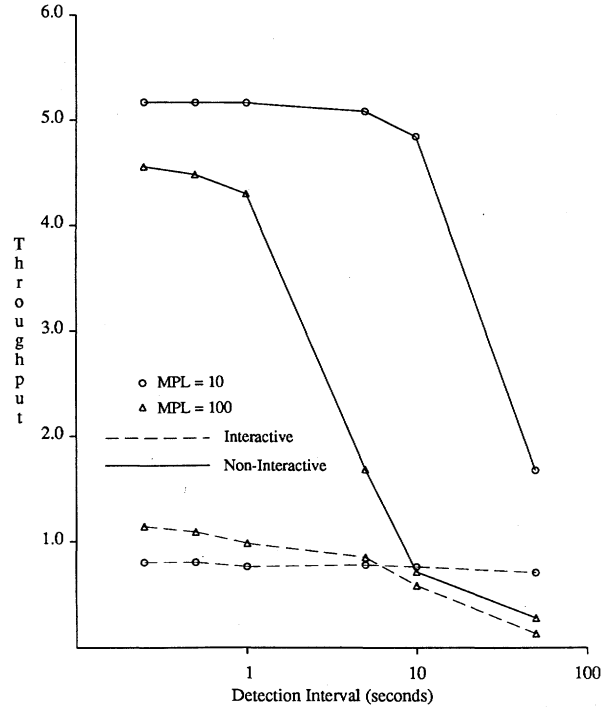


Fig. 7. Throughput, periodic detection, varying detection intervals.

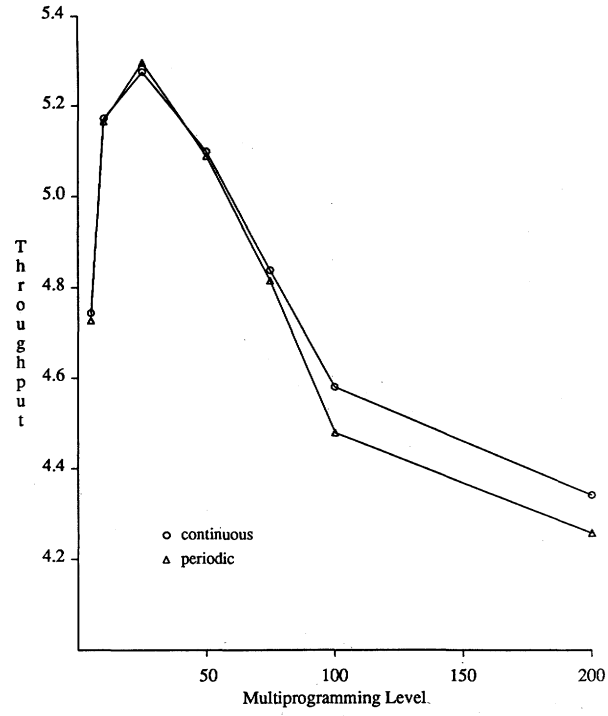


Fig. 8. Throughput, detection alternatives (noninteractive workload).

later. In this experiment, we assumed that the waits-for graph is maintained in main memory,[6] and hence that cycle detection can be performed very efficiently. We also ignored the context-switching overhead involved in taking a timer interrupt, turning control over to the deadlock detection process, and then returning control to one of the transactions. This is why throughput does not suffer at all when the detection interval becomes quite small. In an actual system, throughput would probably decrease slightly for the smallest interval sizes due to the cost of context switching (which is probably about a millisecond or so in a typical system, for two context switches, associated transient cache misses, etc.). We will comment further on these cost modeling issues shortly.

2) Periodic Versus Continuous Detection: The second half of Experiment 2 considered the performance of periodic versus continuous deadlock detection. Based on the preceding results, a time interval of 500 milliseconds was used for periodic detection in the noninteractive workload case, and a 1 second interval was used in the interactive case. The time interval for each class of workload was chosen by selecting the largest interval where neither the *MPL* = 10 nor the *MPL* = 100 throughput curves had dropped off significantly yet in Fig. 7. This provides for a reasonable time gap between detections while not handicapping the periodic deadlock detection algorithm due to an overly large time interval for either class of workload.

Figs. 8 and 9 show the throughput results obtained for the noninteractive and interactive workloads, respectively, and Figs. 10 and 11 show the associated conflict ratios. As shown, the performance of continuous detection is generally slightly better than that of periodic detection. The difference is due to the fact that failure to detect deadlocks early results in somewhat longer blocking times for transactions, leading to higher blocking and restart ratios (as shown in Figs. 10 and 11). The performance difference is more significant in the interactive case (Fig. 9), where continuous deadlock detection outperforms periodic detection by about 10 percent at the high-

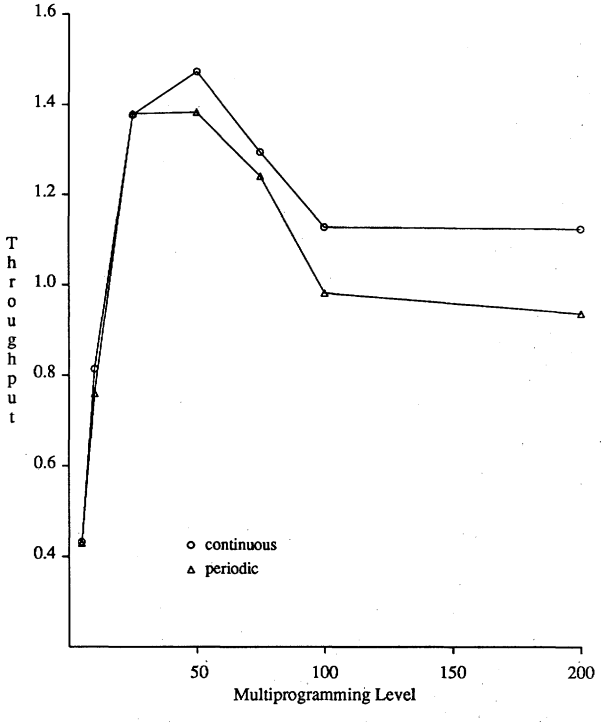


Fig. 9. Throughput, detection alternatives (interactive workload).

[6]In System R and IMS, for example, the lock table is held in main memory [20].

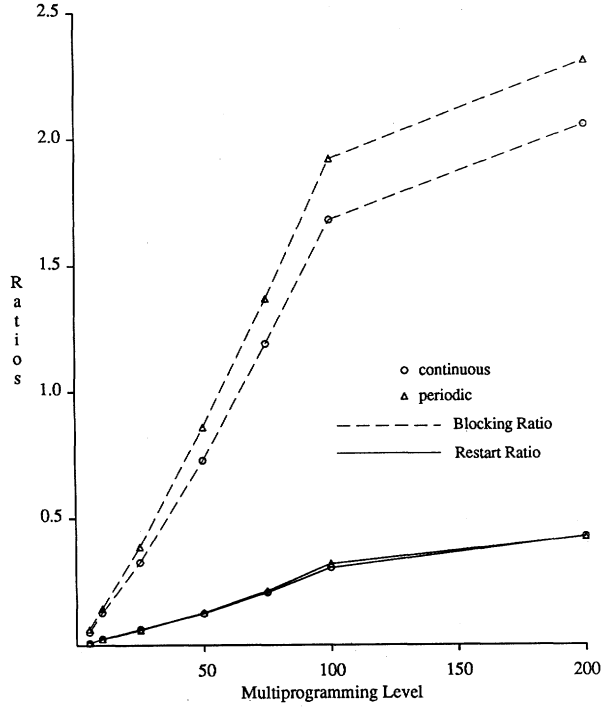


Fig. 10. Blocking and restart ratios, detection alternatives (noninteractive workload).

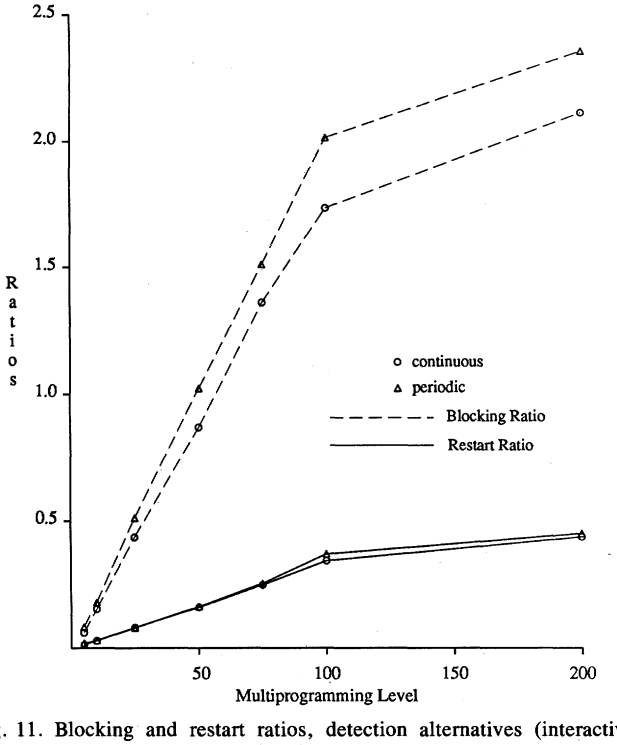


Fig. 11. Blocking and restart ratios, detection alternatives (interactive workload).

est multiprogramming levels. For most of the multiprogramming levels, however, the performance difference is fairly minimal.

Further discussion regarding detection costs is in order at this point. We chose not to include the cost of deadlock detection in our algorithm models for several reasons, not the least of which was that we did not believe we could model such an implementation-dependent cost realistically (so an attempt to do so would probably not lead to convincing results). However, we believe that continuous detection is probably cheaper than periodic detection for several reasons. In periodic detection, the high cost of context switching is likely to dominate the cost of running the detection routine itself—deadlock detection simply involves doing a depth-first search of each connected component of the waits-for graph, which costs $O(N^2)$ where N is the number of blocked transactions [1]. N is likely to be small, being just a fraction of the multiprogramming level. With continuous detection, no context switch is necessary—the blocking transaction is already inside the lock manager when detection is required. In addition, only the connected component containing the current blocking transaction needs to be searched for continuous detection, so the worst-case cycle search cost is just $O(N)$ [1]. Thus, although continuous detection will involve more frequent checking, it appears likely to be less costly overall.[7]

C. Experiment 3: Timeout Interval

We performed a sensitivity analysis to examine the impact of the timeout interval on the performance of the timeout method for deadlock resolution, and also to see if we could discover a reasonable heuristic for selecting the timeout interval. The sensitivity analysis was performed for a low multiprogramming level (10) and a high multiprogramming level (100) both for the noninteractive and interactive workloads. Fig. 12 summarizes the throughput results. As one would expect, the shape of the timeout throughput curve is convex. There is an optimal timeout value for a given level of multiprogramming and workload; smaller values degrade performance through too many restarts, and larger values degrade performance by blocking too many transactions for excessively long times. Note that the ideal timeout interval value varies with both the multiprogramming level and the nature of the workload.

One way of choosing the timeout interval would be to relate it in some way to the waiting time of a blocked request W. The basis for this heuristic is that ideally the timeout interval should be just long enough to allow a blocked request to complete its waiting. We experimented with an adaptive version of the timeout algorithm where the timeout interval was dynamically adjusted to a running estimate of the value $avg(W) + k * \sigma(W)$, where $\sigma(W)$ is the standard deviation of the request waiting time. We ran simulations where k was taken to be 0, 1, and 2. Fig. 13 summarizes the throughput results from this experiment for these values of k. The best overall results were obtained for the dynamically adjusted timeout interval of $avg(W) + \sigma(W)$ (i.e., for $k = 1$).

[7]We make this claim for a centralized system only, as message cost tradeoffs in the distributed case will make periodic detection cheaper there. We note that IBM's System R uses continuous detection, but that its distributed counterpart, System R*, uses periodic detection [24].

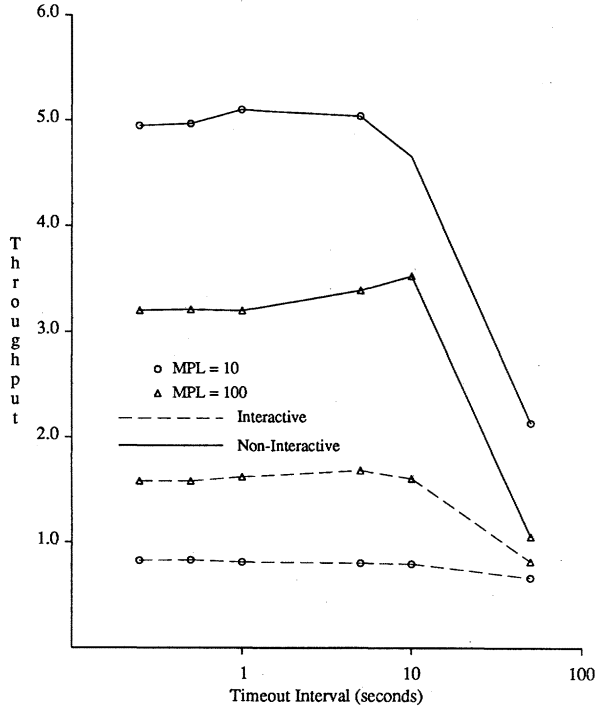


Fig. 12. Throughput, timeout algorithm, varying timeout intervals.

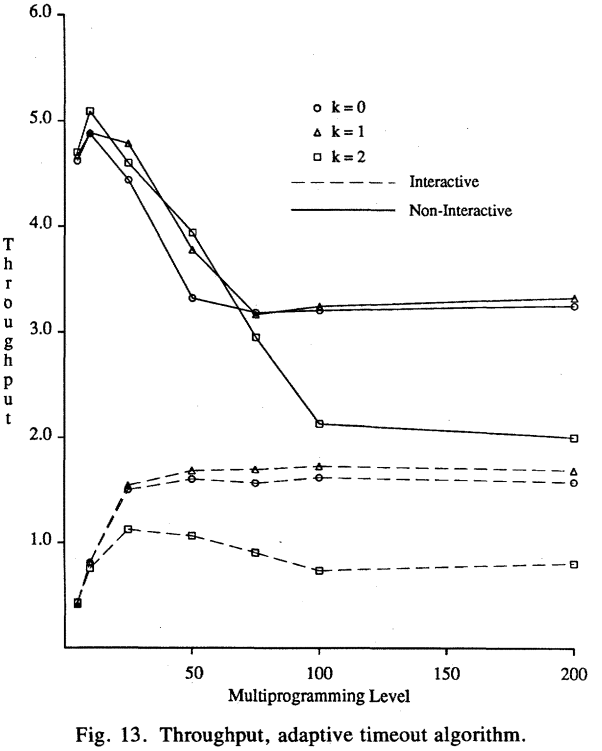


Fig. 13. Throughput, adaptive timeout algorithm.

D. Experiment 4: Detection, Prevention, and Timeout Performance

In Experiment 4, the most important of the experiments that we ran, we compared the performance of continuous deadlock detection, the four deadlock prevention strategies (wound-wait, wait-die, immediate-restart, and running-priority), and the timeout algorithm. Based on the results of Experiment 1, *min locks* was chosen to be the victim selection criteria for continuous deadlock detection. Periodic deadlock detection is not included here since it has already been dealt with separately. The adaptive version of the timeout algorithm was used in this experiment, with the timeout interval being dynamically adjusted to a value equal to $avg(W) + \sigma(W)$. We ran this experiment for both the noninteractive and interactive workloads. We will discuss the results for the two workload types separately here, as their results are significantly different.

1) The Noninteractive Workload: The throughputs obtained with the alternative deadlock resolution strategies are summarized in Fig. 14 for the noninteractive type of transactions. We again omit the response time results, as they display the same relative performance trends. For very low levels of multiprogramming, there are not enough lock conflicts to significantly differentiate between the alternative strategies. However, as the multiprogramming level is increased, differences begin to appear and a number of trends can be seen. Continuous detection emerges as the best strategy, and the immediate-restart algorithm has the worst performance of all. Wound-wait outperforms wait-die. For low levels of multiprogramming, running-priority provides somewhat better performance than wound-wait, but for high levels of multiprogramming, both wound-wait and wait-die perform significantly better than running-priority. The timeout strategy performs better than the immediate-restart strategy, but it performs worse than the other three deadlock prevention strategies. We analyze each of these trends in turn below.

The average utilization of the disks[8] (which happen to be the critical resource in our experiments) is shown in Figs. 15 and 16. The useful utilization is defined as the total utilization minus the fraction of the resource utilization that was used to process transactions that were later restarted. The figures show that, although the total disk utilization is nearly the same for all strategies, the useful utilization is strongly correlated with throughput. Figs. 17 and 18 show the blocking and restart ratios for the alternative strategies. There are clearly a spectrum of blocking and restart ratios for the strategies, ranging from a combination of a zero blocking ratio and the highest restart ratio for the immediate-restart strategy to the highest blocking ratio combined with the lowest restart ratio for the continuous detection algorithm. The conclusion that one can draw is that, in a resource-limited situation, a deadlock resolution strategy that attempts to minimize the number of restarts (and hence to minimize the waste of physical resources) outperforms a strategy that relies exclusively on restarts. It is also the case that the immediate-restart strategy lacks the stability properties that were described in Section IV-A.

Both wound-wait and wait-die offer a balance between the extreme blocking and restart ratios, and each has in-

[8]Average disk utilization is used here because there are two disks in the simulated system; it is defined as the sum of their individual utilizations divided by two.

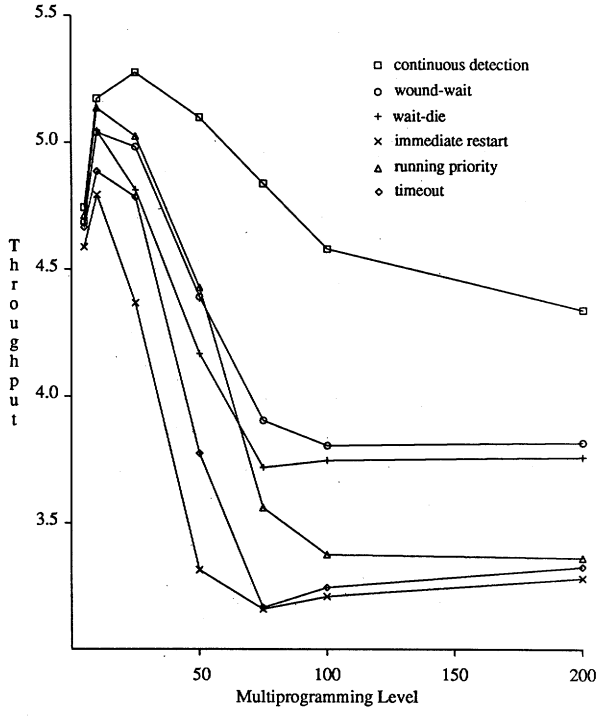


Fig. 14. Throughput, alternative strategies (noninteractive workload).

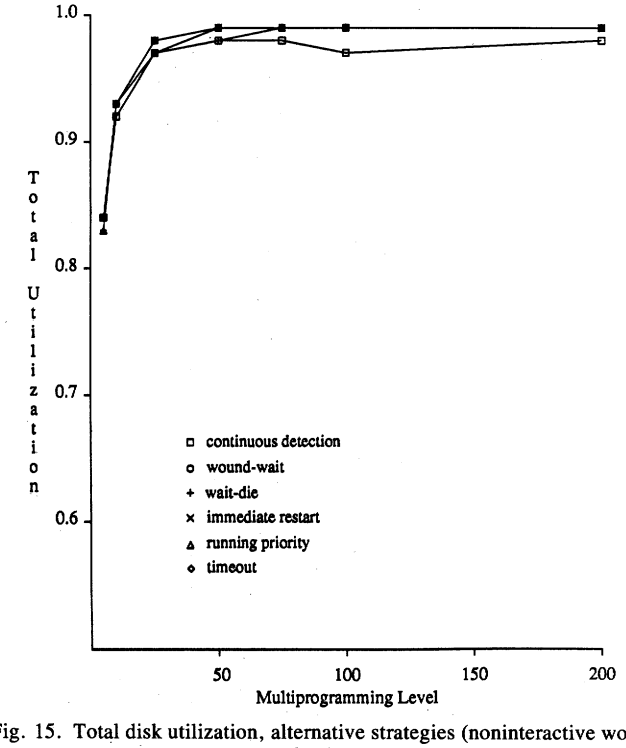


Fig. 15. Total disk utilization, alternative strategies (noninteractive workload).

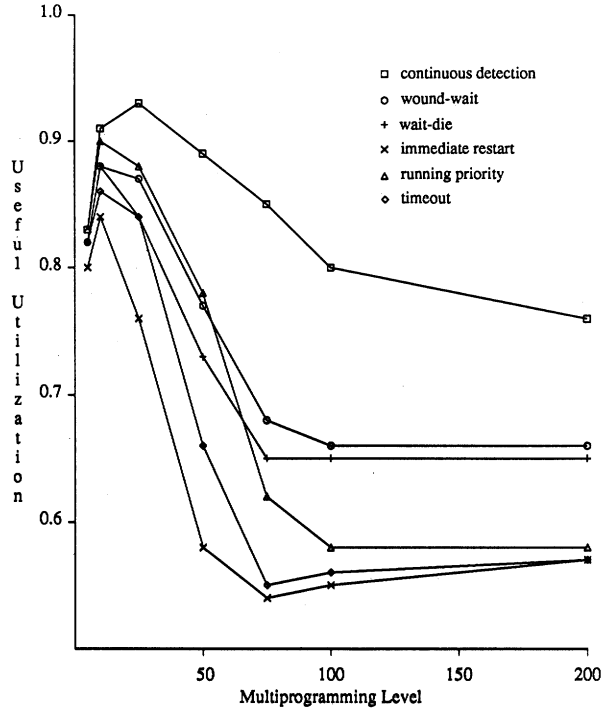


Fig. 16. Useful disk utilization, alternative strategies (noninteractive workload).

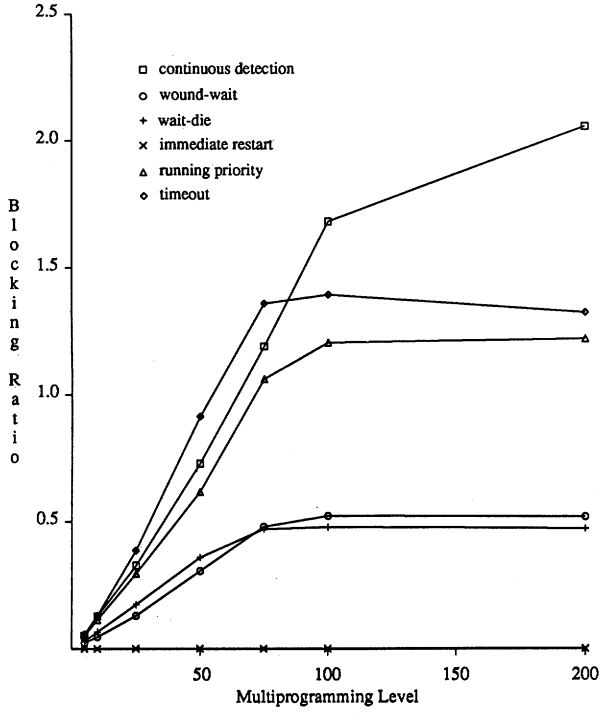


Fig. 17. Blocking ratio, alternative strategies (noninteractive workload).

termediate performance. Note that wound-wait has both of the desirable stability properties described earlier—a restarted transaction will not be restarted again due to the same conflict, and the oldest transaction in the system is guaranteed to complete (and every transaction will eventually become the oldest). Wait-die has the first stability property (guaranteed progress), but it suffers from the possibility of repeated transaction restarts. A younger transaction may be repeatedly restarted by an older transaction in wait-die, whereas in wound-wait the younger transaction will wait for the older transaction to complete after being wounded by it. However, the severity of the repeated restarts problem is alleviated somewhat by the restart delay in our model. In both wound-wait and wait-

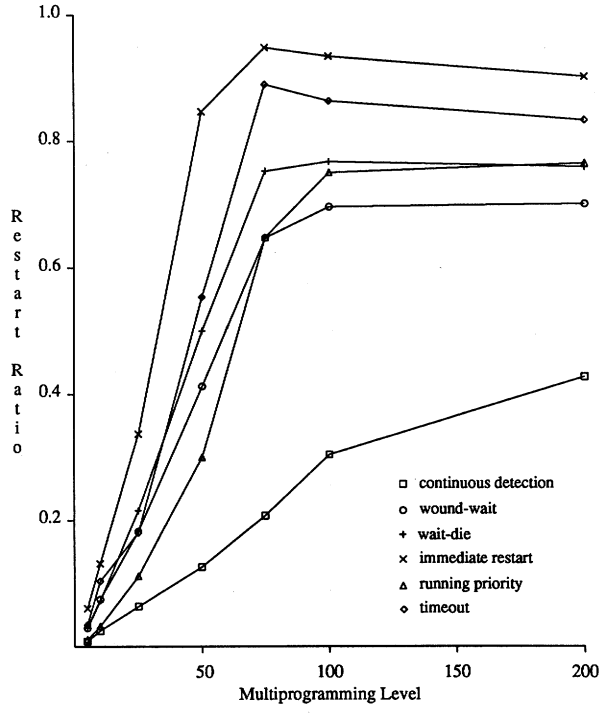


Fig. 18. Restart ratio, alternative strategies (noninteractive workload).

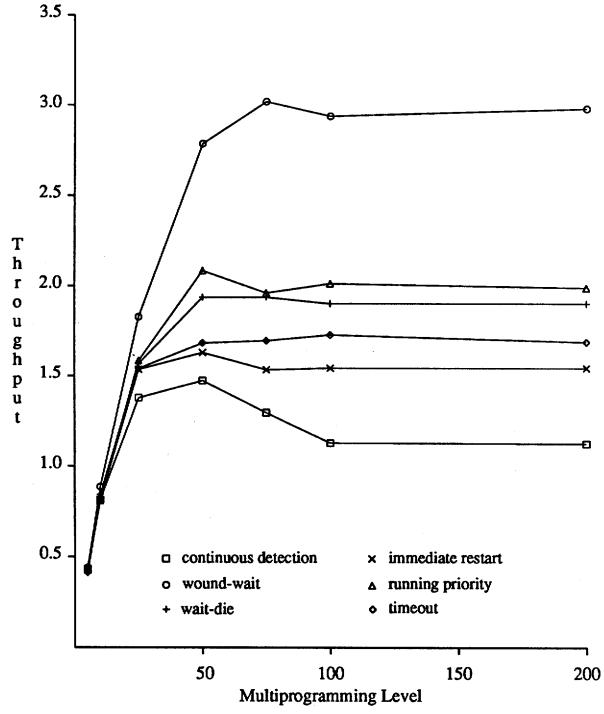


Fig. 19. Throughput, alternative strategies (interactive workload).

die, the oldest transaction in the system is always guaranteed to finish. The restart ratios for wait-die are higher than the restart ratios for wound-wait in Fig. 18, and wound-wait consequently has better performance.

The running-priority strategy is in some sense like continuous detection, but with the restriction that the wait-queue for each lock is limited to containing either one waiting writer or one set of waiting readers. It is thus not surprising that, for low multiprogramming levels where conflicts are rare and hence the average wait queue length is small, running-priority exhibits behavior like that of continuous deadlock detection. It has high blocking ratios and low restart ratios, and it outperforms wound-wait and wait-die in these cases. However, running-priority lacks the two stability properties. That is, a restarted transaction is basically random, being one that happened to be waiting at the time of a lock conflict rather than being chosen for age, lock count, or resource related reasons; consequently, no one transaction is assured of safely finishing, and there is no guarantee that an unlucky transaction will not be repeatedly restarted. For higher multiprogramming levels, then, running-priority is outperformed by wound-wait and wait-die because it lacks their stable, age-based selection of which transaction to restart.

The poor performance of the timeout strategy reflects the inherent problem of selecting a good timeout interval. Even with our adaptive version of the timeout algorithm, both the restart ratios and blocking ratios are much higher than those of wound-wait, wait-die, or running-priority, and consequently timeout has inferior performance. The timeout strategy also lacks the desired stability properties.

2) The Interactive Workload: Fig. 19 gives the throughput results for the alternative deadlock resolution strategies for the interactive workload case. As in the case of noninteractive transactions, the performance of the alternative strategies is not differentiable for low multiprogramming levels. As the multiprogramming level is increased, though, results very different from those for the noninteractive type workload are observed. Wound-wait emerges as the overall best strategy here, and the continuous deadlock detection strategy has the worst performance. Running-priority performs a little better than wait-die, and they both perform better than the timeout or immediate-restart algorithms. Timeout has slightly better performance than immediate-restart.

Figs. 20 and 21 show the total and useful average disk utilizations for the alternative strategies, and Figs. 22 and 23 give their blocking and restart ratios. On the average, transactions in our experiments require 150 milliseconds of CPU time and 350 milliseconds of disk time, so an internal think time of 10 seconds considerably reduces the average number of running (i.e., nonthinking) transactions. This reduces the demand on resources, and the resource utilizations decrease considerably. For high levels of multiprogramming, as the probability of conflicts increases, the blocking ratios increase; due to the fact that transactions think while holding locks, the lock waiting times also increase, which in turn further increases the amount of blocking. This excessive waiting in the system is the major reason for the degradation of the performance of continuous deadlock detection. Although its restart ratio increases due to deadlocks as the multiprogramming level is increased, blocking increases at a much faster rate than deadlocks. The immediate-restart strategy, which is based purely on restarting a conflicting transaction, does

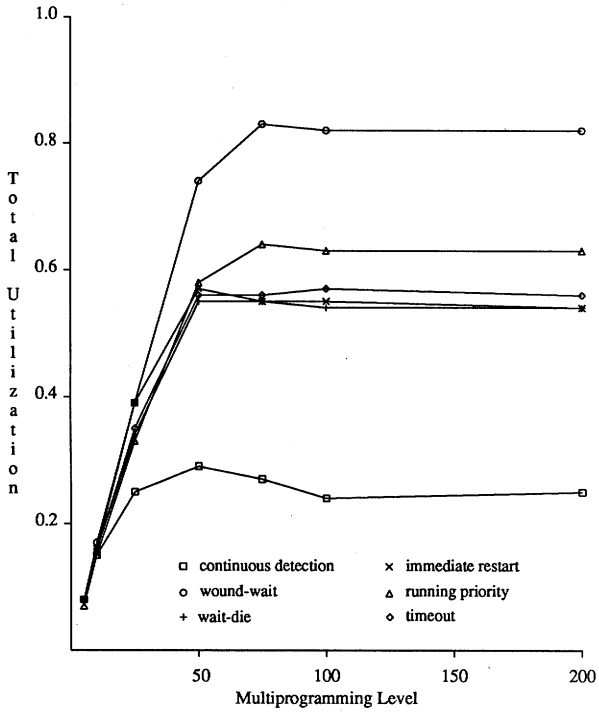


Fig. 20. Total disk utilization, alternative strategies (interactive workload).

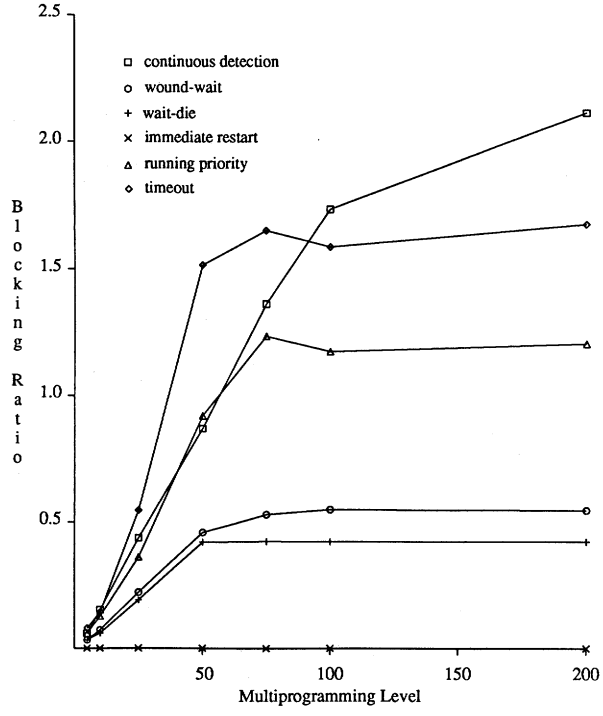


Fig. 22. Blocking ratio, alternative strategies (interactive workload).

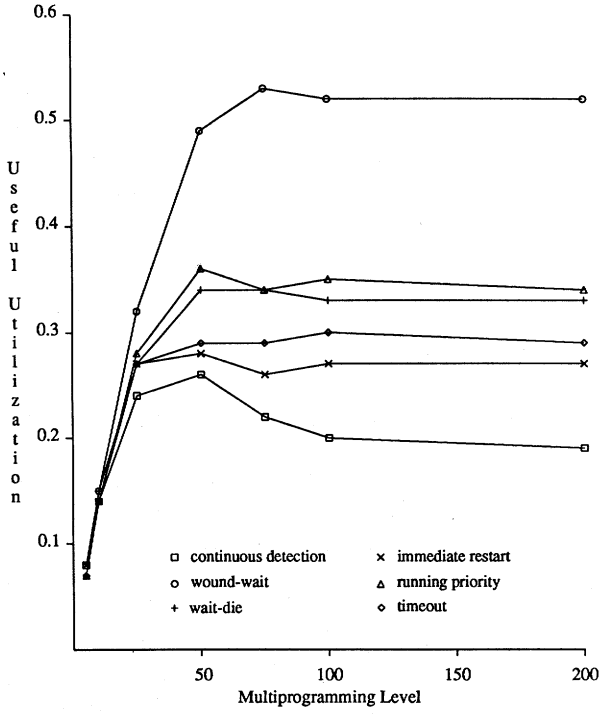


Fig. 21. Useful disk utilization, alternative strategies (interactive workload).

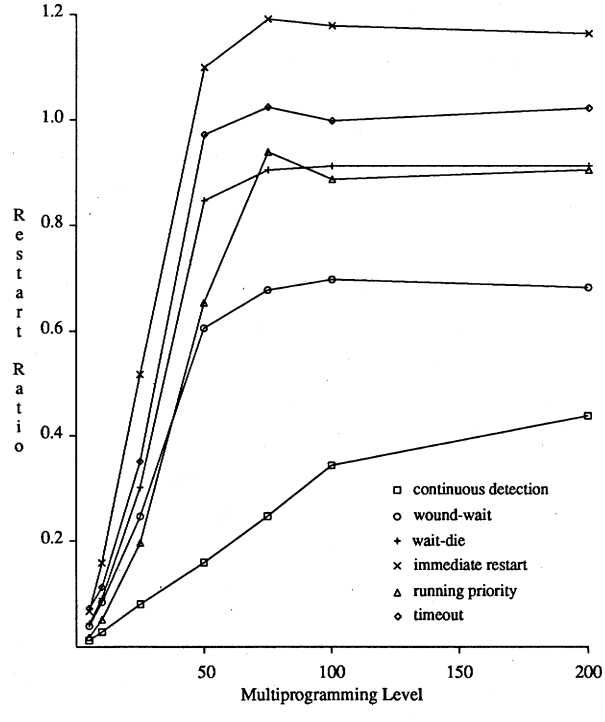


Fig. 23. Restart ratio, alternative strategies (interactive workload).

somewhat better—because the resource utilizations are so low in this environment, the benefits attained through not blocking other transactions by waiting and holding locks outweigh the cost of wasted resources due to restarts (as reported by Tay in [40], [41]). Due to the restart delay, which is needed to prevent transactions from being restarted repeatedly in the immediate-restart algorithm, and due to the high number of restarts, a plateau is reached where increasing the multiprogramming level does not increase the number of active transactions because all other transactions are either in a think state or a restart delay state.

The best performance for the interactive workload case is achieved using a deadlock resolution strategy that bal-

ances the restart and blocking ratios, as is the case with the wound-wait strategy. Observe that the total and useful disk utilizations are also largest with this strategy (see Figs. 20 and 21). Wait-die has higher restart ratios than wound-wait, causing wait-die to reach a plateau with fewer active transactions in the system than wound-wait. Running-priority attempts to limit waiting in the system by preempting blocked transactions in favor of active transactions. However, the blocking and restart ratios are much higher with running-priority compared to those for wound-wait, resulting in its relatively poorer performance. An analysis of simulation traces from running-priority and wound-wait runs showed that, in wound-wait, the major source of restarts is from write requests restarting conflicting younger read requests, which means that writers are pushing younger readers out of their way. In running-priority, however, it is common for a writer to block and wait for several active readers, and then to be subsequently restarted when a new reader arrives in the queue (since the reader will not wait for the blocked writer)—this leads to starvation of write requests under high conflicts. More restarts actually occur with wound-wait, but wound-wait has a lower restart ratio than running-priority because it also has many more commits (i.e., its restarts are productive ones). Running-priority has a higher blocking ratio because it blocks write requests before restarting them (as just described). Finally, timeout also has restart and blocking ratios which are higher than those of wound-wait, wait-die, and running-priority, explaining its lesser performance here.

E. Comparison to Other Results

It is interesting to compare our results on the performance of deadlock resolution strategies to the simulation results of Munz and Krenz [30] and those of Balter, Berard, and Decitre [5]. We consider each of these studies in turn.

In [30], the relative performance of eleven victim selection algorithms was studied. They found the following three methods, each of which uses some sort of minimal cost criteria, to be considerably better than the other criteria examined—pick the transaction with the fewest locks, pick the transaction with the smallest number of exclusive locks, and pick the transaction that has done the least work (using a cost function based on CPU time, main storage occupation, and I/O activity). No clear winner among these algorithms was found. They also examined strategies where the current blocker and the most costly transaction were chosen as the victim. Our victim selection results for the noninteractive workload case concur well with theirs. For the interactive case, we found that the criterion based on the number of locks was by far the best, whereas it was just marginally best in the noninteractive case.

In [5], deadlock detection (with an unspecified victim selection algorithm), wound-wait, an "improved" version of wound-wait, wait-die, and a "modified deadlock detection" scheme were compared. The modified deadlock detection scheme was a nonpreemptive version of the running-priority scheme that we have considered in this paper. It allows a transaction to wait only for an active transaction, restarting transactions that request locks that conflict with those held by waiting transactions. (Running-priority would restart the waiting transactions instead.) It was concluded that, under all conditions, for high multiprogramming levels, simple deadlock detection had the worst performance; wound-wait outperformed "improved" wound-wait and wait-die; and the modified deadlock detection performed the best of all algorithms studied. Unfortunately, the simulation model description in [5] is somewhat vague, and simulation parameter values were not specified.

Our conclusions regarding the relative performance of deadlock detection, wound-wait, and wait-die are similar to theirs under interactive workloads, but for transactions with no internal think times, our conclusions are very different. This is because Balter, Berard, and Decitre did not include physical resources (disks and CPUs) in their model, effectively assuming them to be infinite by assuming that all active transactions can be processed in parallel. As explained in [2] and in Section IV, performance trends for interactive workloads that lightly load a database system are similar to those that arise when infinite resources are assumed. As for the overall better performance of the modified deadlock detection algorithm in their paper, we found in our interactive workload experiments that wound-wait consistently outperformed a similar algorithm, running-priority. Our result here also seems to contradict the result of Franaszek and Robinson that running-priority outperforms wound-wait [16]. The explanation lies in the lock modes considered in the studies—in our experiments, both read and write locks were used, whereas the studies by Balter *et al.* and by Franaszek and Robinson considered only exclusive locks [5], [16]. The starvation problem that we described will not arise if transactions just use exclusive locks. Since real database systems usually provide both read locks and write locks, we feel that our results are probably more indicative of the real performance of the algorithms.

One further note on our performance results for wound-wait versus running-priority is in order. In our studies, all transactions were similar—they each read a number of objects, then wrote a subset of these objects, upgrading their read locks to write locks when making their write requests. Thus, virtually all transactions eventually needed to obtain one or more write locks, and write requests tended to be from older transactions than the conflicting read requests. Wound-wait is favored by such a situation, and running-priority's starvation problem is particularly significant under these conditions. In order to rule out the possibility that our results were situation-specific, we ran another interactive workload experiment with two classes of transactions, pure readers and pure writers. We again found that wound-wait outperformed running-priority, but the difference was a little less pronounced in this case (wound-wait was 40 percent better than running-priority

instead of 50 percent better at high multiprogramming levels). As a result of this experiment, we are quite confident that our results are indeed due to our having both read and write locks, and not to our particular choice of workload.

V. Conclusions

A major conclusion of this study is that the choice of the best deadlock resolution strategy is dependent upon the system's operating region. In a low conflict situation, the performance of all deadlock resolution strategies is basically identical. There are just not enough deadlocks in such situations to distinguish between the algorithms. In a situation where resources are fairly heavily utilized, continuous deadlock detection is the deadlock resolution strategy of choice. In this situation it is best to choose an algorithm that minimizes transaction restarts and hence wastes little in the way of resources. Such situations arise in database systems with physical resources and workloads such that a number of transactions tend to be competing for these resources at all times (i.e., primarily noninteractive workloads). If the transactions are primarily interactive, however, with long think times during which locks are held, then a deadlock detection strategy can cause excessive blocking and perform poorly. In such situations, an algorithm like wound-wait, which balances the blocking and restart ratios, provides the best performance. A deadlock resolution strategy such as immediate-restart, which exclusively relies on transaction restarts, was found to perform relatively poorly in both situations. Finally, continuous deadlock detection outperformed periodic deadlock detection due to the higher blocking and restart ratios associated with periodic detection, but the performance difference was generally quite small.

We presented two stability properties that we claim are desirable for deadlock resolution strategies, the first being that at any time there exists some transaction in the system that is guaranteed not be restarted (*guaranteed progress*), and the second being that transactions cannot be restarted over and over many times (*no repeated restarts*). While having these properties will not guarantee superior performance for an algorithm, the absence of these properties may cause an algorithm to become unstable in high conflict situations. Among the victim selection criteria that we considered, the random and current victim selection criteria lack these properties, the youngest criteria has both of these properties, and the minimum locks and the minimum resources selections have approximately these stability properties. Consequently, the performance of the random and current criteria were found to be inferior to that of the other victim selection criteria. The minimum number of locks criterion was found to be better than the other criteria because if a transaction that is holding a large number of locks is restarted, it has to contend for all of its locks again, increasing the probability of blocking and restarts. The wound-wait algorithm has both of the desired stability properties, whereas there may be repeated restarts with wait-die. Wound-wait was found to be consistently superior to wait-die. Running-priority has the nice property that it always chooses a blocked transaction in favor of a running transaction for restarting, but it lacks the stability properties. It thus has higher blocking and restart ratios as compared to wound-wait for high multiprogramming levels, resulting in inferior performance for high contention situations.

Our study highlighted the difficulty in choosing an appropriate timeout interval for the timeout strategy. It was demonstrated that the performance of timeout is sensitive to the timeout interval, and that the "right" timeout interval varies with the multiprogramming level and the transaction workload characteristics. We experimented with an adaptive version of the timeout strategy that outperformed the simple timeout strategy with its fixed timeout interval, but in no situation did timeout become the strategy of choice.

Several directions seem appropriate for future work. First, since our experiments were confined to the case of centralized systems, an interesting open problem is the extent to which these results will extend to the distributed database case. The tradeoffs are less clear there, as message overhead adds a new dimension to the problem. Second, it would be useful to validate our simulation model by repeating the experiments in a real database system. This would require access to the source code for a real database system with an architecture that allows transactions to be blocked, restarted while blocked, and restarted while not blocked (i.e., preempted while running), as otherwise it would be necessary to rewrite much of the system's lock manager and recovery subsystem. Unfortunately, we do not currently have access to a system that meets these requirements. Finally, since our results indicate that the best choice for a deadlock resolution strategy depends on the system's level of resource utilization, which often varies over time, it would be both interesting and useful to develop criteria for dynamically choosing a deadlock resolution strategy.

Acknowledgment

We wish to thank R. Canaday of AT&T Bell Laboratories for his encouragement and support of this work. The NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were used in the course of this study.

References

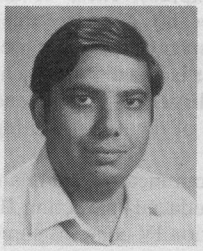
[1] R. Agrawal, M. Carey, and D. J. DeWitt, "Deadlock detection is cheap," *ACM-SIGMOD Rec.*, vol. 13, no. 2, pp. 19-34, Jan. 1983.

[2] R. Agrawal, M. J. Carey, and M. Livny, "Models for studying concurrency control performance: Alternatives and implications," in *Proc. ACM-SIGMOD 1985 Int Conf. Management of Data*, May 1985, pp. 108-121; to appear in extended form as "Concurrency control performance modeling: Alternatives and implications," *ACM Trans. Database Syst.*, vol. 12, no. 4, Dec. 1987.

[3] R. Agrawal and D. J. DeWitt, "Integrated concurrency control and recovery mechanisms: Design and performance evaluation," *ACM Trans. Database Syst.*, vol. 10, no. 4, pp. 529-564, Dec. 1985.

[4] A. V. Aho, E. Hopcraft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1975, ch. 5.

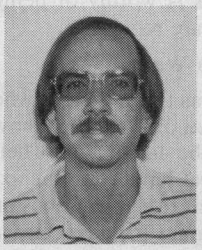
[5] R. Balter, P. Berard, and P. Decitre, "Why the control of concurrency level in distributed systems is more fundamental than deadlock management," in *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1982, pp. 183–193.
[6] C. Beeri and R. Obermarck, "A resource class independent deadlock detection algorithm," in *Proc. 7th Int. Conf. Very Large Data Bases*, Sept. 1981.
[7] P. A. Bernstein, J. B. Rothnie, N. Goodman, and C. H. Papadimitriou, "The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case)," *IEEE Trans. Software Eng.*, SE-4, no. 3, pp. 154–168, May 1978.
[8] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, June 1981.
[9] ——, "A sophisticate's introduction to distributed database concurrency control," in *Proc. 8th Int. Conf. Very Large Data Bases*, Sept. 1982, pp. 62–76.
[10] M. J. Carey, "Modeling and evaluation of database concurrency control algorithms," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Berkeley, Dec. 1983.
[11] M. J. Carey and M. R. Stonebraker, "The performance of concurrency control algorithms for database management systems," in *Proc. 10th Int. Conf. Very Large Data Bases*, Aug. 1984, pp. 107–118.
[12] D. D. Chamberlain, R. F. Boyce, and I. L. Traiger, "A deadlock-free scheme for resource locking in a data base environment," in *Proc. IFIP 1974*, pp. 340–343.
[13] W. N. Chin, "Some comments on 'deadlock detection is cheap' in *SIGMOD Record* Jan. 83," *ACM SIGMOD Record*, pp. 61–63, Mar. 1984.
[14] E. G. Coffman, M. J. Elphic, and A. Shoshani, "System deadlocks," *ACM Comput. Surveys*, vol. 3, no. 2, pp. 67–78, June 1971.
[15] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.
[16] P. Franaszek and J. T. Robinson, "Limitations of concurrency in transaction processing," *ACM Trans. Database Syst.*, vol. 10, no. 1, pp. 1–28, Mar. 1985.
[17] V. D. Gligor and S. H. Shattuck, "On deadlock detection in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-6, no. 5, Sept. 1980.
[18] J. N. Gray, "Notes on database operating systems," in *Operating Systems: An Advanced Course*, (*Lecture Notes in Computer Science* 60), R. Bayer, R. M. Graham, and G. Seegmuller, Eds. New York: Springer-Verlag, 1979.
[19] J. N. Gray, P. Homan, H. Korth, and R. Obermarck, "A straw man analysis of the probability of waiting and deadlock in a database system," IBM Res. Lab., San Jose, CA, Rep. RJ3066, Feb. 1981.
[20] J. N. Gray, personal communication to D. DeWitt, Apr. 1982.
[21] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179–196, Sept. 1972.
[22] P. F. King and A. J. Collmeyer, "Database sharing—An efficient method for supporting concurrent processes," in *Proc. AFIPS 1973 Nat. Comput. Conf.*, 1973, pp. 271–275.
[23] H. F. Korth, "Edge locks and deadlock avoidance in distributed systems," in *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1982, pp. 173–182.
[24] B. Lindsay, personal communication, May 1985.
[25] D. B. Lomet, "A practical deadlock avoidance algorithm for data base systems," in *Proc. ACM-SIGMOD 1977 Int. Conf. Management of Data*, Aug. 1977, pp. 122–127.
[26] D. B. Lomet, "Deadlock avoidance in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, Mar. 1980.
[27] P. P. Macri, "Deadlock detection and resolution in a CODASYL based data management system," in *Proc. ACM-SIGMOD 1976 Int. Conf. Management of Data*, June 1976, pp. 45–49.
[28] D. A. Menasce and R. R. Muntz, "Locking and deadlock detection in distributed databases," in *Proc. 3rd Berkeley Workshop Distributed Data Management and Computer Networks*, Aug. 1978.
[29] D. Mitchell and M. J. Merritt, "Distributed algorithm for deadlock detection and resolution," in *Proc. ACM-SIGACT-SIGMOD Conf. Principles of Distributed Computing*, Aug. 1984.
[30] R. Munz and G. Krenz, "Concurrency in database systems—A simulation study," in *Proc. ACM-SIGMOD 1977 Int. Conf. Management of Data*, Aug. 1977, pp. 111–120.
[31] G. Newton, "Deadlock prevention, detection and resolution: An annotated bibliography," *ACM-SIGOPS Operating Syst. Rev.*, vol. 13, no. 4, pp. 33–44, Apr. 1979.
[32] R. Obermarck, "Distributed deadlock detection algorithm," *ACM Trans. Database Syst.*, vol. 7, no. 2, June 1982.
[33] A. Reuter, "An analytic model of transaction interference in database systems," Univ. Kaiserslautern, West Germany, Rep. IB 68/83, 1983.
[34] D. R. Ries and M. R. Stonebraker, "Effects of locking granularity in a database management system," *ACM Trans. Database Syst.*, vol. 2, no. 3, pp. 233–246, Sept. 1977.
[35] D. R. Ries, "The effects of concurrency control on database management system performance," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Berkeley, Apr. 1979.
[36] D. R. Ries and M. R. Stonebraker, "Locking granularity revisited," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 210–227, June 1979.
[37] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.*, vol. 3, no. 2, pp. 178–198, June 1978.
[38] R. Sargent, "Statistical analysis of simulation output data," in *Proc. 4th Annu. Symp. Simulation of Computer Systems*, 1976.
[39] M. R. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. Software Eng.*, vol. SE-5, no. 3, pp. 188–194, May 1979.
[40] Y. C. Tay, "A mean value performance model for locking in databases," Ph.D. dissertation, Harvard Univ., Cambridge, MA, Tech. Rep. 04-84, Feb. 1984.
[41] Y. C. Tay, N. Goodman, and R. Suri, "Locking performance in centralized databases," *ACM Trans. Database Syst.*, vol. 10, no. 4, pp. 415–462, Dec. 1985.



Rakesh Agrawal received the B.E. degree in electronics and communication engineering from the University of Roorkee, Roorkee, India, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison, in 1983.

Since then, he has been with AT&T Bell Laboratories, Murray Hill, NJ, where he is a member of the Technical Staff in the Computer Technology Research Laboratory. His current research interests include distributed and parallel processing, database management systems, and computer architecture.

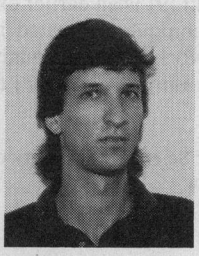
Dr. Agrawal is a member of the Association for Computing Machinery and the IEEE Computer Society.



Michael J. Carey (S'77–M'79) received the B.S. degree in electrical engineering and mathematics and the M.S. degree in electrical engineering (computer engineering) from Carnegie-Mellon University, Pittsburgh, PA, in 1979 and 1981, respectively, and the Ph.D. degree in computer science from the University of California, Berkeley, in 1983.

Since then he has been an Assistant Professor of Computer Sciences at the University of Wisconsin, Madison. His research interests include database management systems, distributed and parallel processing, and applied performance evaluation.

Dr. Carey received an IBM Faculty Development Award in 1984. He is a member of Eta Kappa Nu, Tau Beta Pi, the Association for Computing Machinery, and the IEEE Computer Society.



Lawrence W. McVoy received the B.S. degree in computer sciences from the University of Wisconsin, Madison, in 1985, and he is currently working toward the M.S. degree in the computer sciences graduate program there. His interests include operating systems, networks, distributed processing, and database performance modeling.

Mr. McVoy is a member of the Association for Computing Machinery and the IEEE Computer Society.