THE PERFORMANCE OF ALTERNATIVE STRATEGIES FOR DEALING WITH DEADLOCKS IN DATABASE MANAGEMENT SYSTEMS

by

Rakesh Agrawal
Michael J. Carey
Larry W. McVoy

Computer Sciences Technical Report #590

March 1985

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

Rakesh Agrawal

AT&T Bell Laboratories
Murray Hill, NJ 07974

Michael J. Carey
Larry W. McVoy

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

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

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

Rakesh Agrawal

AT&T Bell Laboratories
Murray Hill, NJ 07974

Michael J. Carey
Larry W. McVoy

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

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 "complete" and "realistic" model of a database environment to study the relative performance of a number of 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.

1. 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 [Eswa76, Gray79, Bern81, Bern82] will outperform concurrency control algorithms that use transaction restarts rather than blocking to resolve conflicts between transactions [Agra85]. 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 [Gray79, Bern81, Bern82]. 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.

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

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 prevent deadlocks from happening [King73, Cham74, Macr76, Lome77, Bern78, Rose78, Mena78, Gray79, Ston79, Newt79, Lome80, Glig80, Beer81, Balt82, Ober82, Kort82, Agra83a, Fran83, Chin84, Mitc84]. There have been several analytical studies devoted to estimating the probability of deadlocks [Gray81, Reut83, Tay84], 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 [Munz77] and a more recent study by Balter, Berard and Decitre [Balt82]. In [Munz77], 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 [Balt82], five deadlock resolution strategies were compared using simulation. Unfortunately, the paper does not give any details of the simulation model used, the range of operating regions considered, or the values of their simulation parameters. 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 complete and (we believe) realistic 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 CPUs) 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 [Gray79, Bern81], including a number of 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 probabilities), and both non-interactive and interactive workload types are considered.

The organization of the remainder of the paper is as follows. In Section 2 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 3. Section 4 presents the performance experiments and our results. Finally, Section 5 summarizes the main conclusions of this study.

2. DEADLOCK RESOLUTION STRATEGIES

Deadlocks are usually characterized in terms of a waits-for graph [Coff71, Holt72, Gray79]. In a database context, a waits-for graph G is a directed graph where each vertex represents a transaction, and each edge of the form $(T_i, T_j) \in G$ means that transaction T_i is waiting for a lock owned by transaction T_j. It has been shown that there exists a deadlock if and only if there is a cycle in the waits-for graph [Coff71, Holt72]. We classify deadlock resolution strategies into three broad classes depending upon whether they require explicit maintenance of the waits-for graph and whether or not just transactions that are actually involved in deadlocks are 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.

2.1. 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 [Aho75]. In *periodic detection*, the graph is searched only periodically for cycles, and all connected components of the graph must be searched in this case. We will study the effect 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 [Munz77], we have not considered any victim selection criteria that selects the most "expensive" transaction, such as the transaction that is holding largest number of locks or that has consumed largest amount of physical resources. It was shown in [Munz77] that the least "expensive" counterpart 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 [Gray79], 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 [Gray79].

2.2. 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) *Wound-Wait*: If a transaction T_i requests a lock that conflicts with a lock held by 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 [Rose78]. 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 transaction T_i requests a lock that conflicts with a lock held by 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 [Rose78]. Again, deadlocks are impossible because a transaction can only be blocked by an older transaction. Wait-die is a non-preemptive counterpart to the wound-wait deadlock prevention algorithm.

(3) *Immediate-Restart*: If a transaction T_i requires a lock that conflicts with a lock held by another transaction T_j, then simply restart T_i. In this algorithm, due to Tay [Tay84], deadlocks are impossible because no transaction is ever blocked.

(4) *Running-Priority*: If a transaction T_i requests a lock that conflicts with a lock held by another transaction T_j, resolve the conflict as follows: If T_j is waiting for some other transaction T_k, restart T_j. This algorithm is due to Franaszek and Robinson [Fran83], 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 a lock held by 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.

2.3. 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 4, 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.

3. SIMULATION MODEL

Our simulator for studying the performance of the deadlock resolution strategies is based on the closed queuing model of a single-site database system depicted in Figure 1. This model is the same simulation model that was used in [Agra85], which is a model that has its origins in the models of [Ries77, Ries79a, Ries79b] and [Care83, Care84]. 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 re-enters the concurrency control queue and makes the lock 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 conflicts with a lock held by another transaction, the concurrency control module takes actions that are dependent upon the deadlock resolution strategy being used. For example, in the

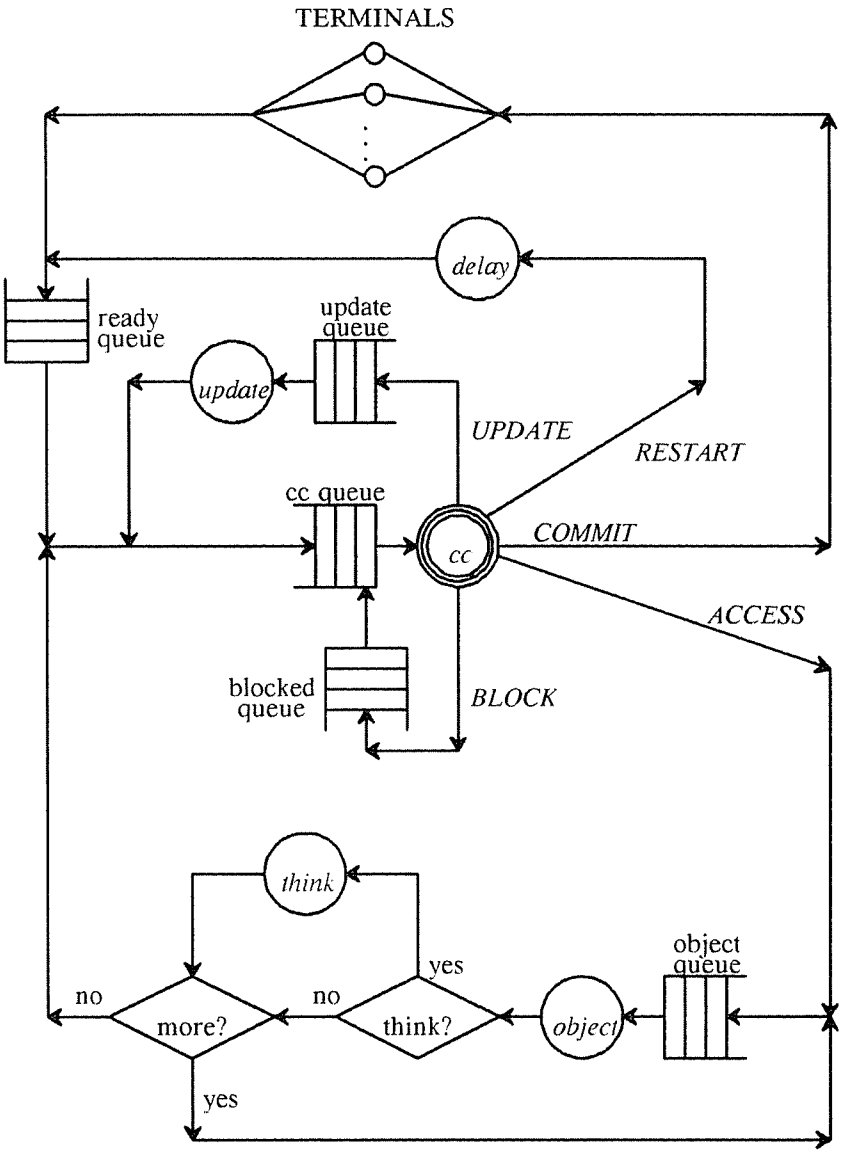

Figure 1: Logical Queuing Model.

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 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 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 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 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 the transaction will complete. 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. Locks are released together at end-of-transaction after the deferred updates have been performed.

Underlying the logical model of Figure 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 Figure 1 are some use of one or both of these two resources. The amounts of CPU and I/O time per logical service are specified as simulation parameters. The physical queuing model is depicted in Figure 2, and Table 1 summarizes its associated simulation parameters. As shown, the physical model is a collection of terminals, CPU servers, and I/O servers. (While the model permits arbitrary numbers of each class of server, as shown in Figure 2, our experiments will all be based on a configuration with one CPU server and two I/O servers. The sufficiency of this configuration for our purposes will be addressed in Section 4.) The delay paths for the think and restart delays are also reflected in the physical queuing model. Requests in the CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. Our I/O model is that of a partitioned database, where the data in the database is spread out evenly across all of the I/O servers. There is a

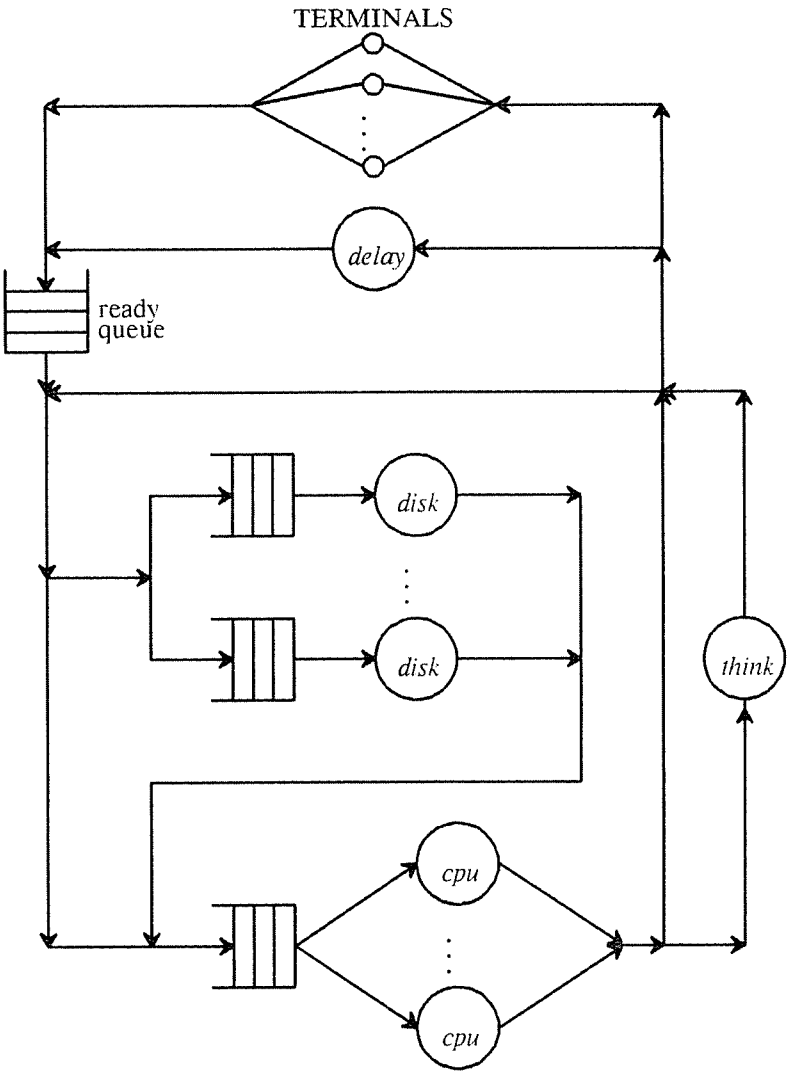

Figure 2: Physical Queuing Model.

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 an I/O queue associated with the selected server. The service discipline for the I/O queues is also FCFS.

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

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

Table 1: Simulation Parameters.

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

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

4. PERFORMANCE EXPERIMENTS

We performed a number of simulation experiments to study the performance of the alternative deadlock resolution strategies described in Section 2. Table 2 gives the simulation parameter values used for the experiments reported here. The parameters that vary from experiment to experiment are not listed in the table, but will instead be given in the description of the experiments.

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

Table 2: Simulation Parameter Settings.

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, set to 200 in this study, to allow a range of conflict probabilities to be investigated. The object processing costs were chosen based on our notion of roughly what realistic values might be. We employed a modified form of the batch means method [Sarg76] for our statistical data analyses, and each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90% confidence intervals.[1] 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 considered two types of transaction workloads, a non-interactive workload, in which transactions have no intra-transaction think time, and an interactive workload, in which transactions perform a number of reads, then think for some period of time (their "internal think time", during which read locks are held), and then perform their

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

writes. Besides being of interest in their own right, we found in [Agra85] that an interactive transaction mix with a large intra-transaction (internal) think time causes a system with finite physical resources (CPUs and disks) to behave as if the system had infinite resources. Based on the results in [Agra85], 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[2] of the bottleneck resource type is moderately high (60-70% and above), the non-interactive workload results are good indicators of relative performance; for low utilizations (40-50% and below), interactive workloads with high intra-transaction 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 CPUs 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).

We also found in [Agra85] that for larger values of the database size (for 10,000 objects), conflicts became very rare; when conflicts are rare, all concurrency control algorithms perform alike (as shown in [Agra83b, Care83, Care84, Agra85] 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. We have also not varied transaction size as a simulation parameter, again because varying the multiprogramming level as we do for a fixed size database provides us with a similar range of conflict situations of interest.

4.1. Experiment 1: Continuous Detection

Our first experiment examined the effectiveness of the alternative victim selection algorithms for continuous deadlock detection. Tables 3 and 4, respectively, show the throughputs (in transactions per second) obtained for the five selection algorithms for the non-interactive 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

[2] We will explain the difference between total and useful resource utilization in Section 4.4.

victim selection criteria	Multiprogramming Level						
	5	10	25	50	75	100	200
current	4.724	5.163	5.226	4.952	4.607	4.144	3.763
random	4.740	5.173	5.228	4.920	4.639	4.101	3.852
youngest	4.741	5.186	5.245	5.043	4.708	4.180	4.040
min locks	4.744	5.172	5.275	5.099	4.839	4.581	4.342
min work	4.724	5.160	5.295	5.102	4.749	4.335	4.217

Table 3: Throughput, Continuous Detection (non-interactive workload).

Victim Selection Criteria	Multiprogramming Level						
	5	10	25	50	75	100	200
current	0.439	0.779	1.341	1.435	1.205	0.968	0.783
random	0.434	0.818	1.400	1.379	1.071	0.951	0.704
youngest	0.446	0.781	1.309	1.402	1.279	0.983	0.908
min locks	0.432	0.814	1.378	1.473	1.295	1.127	1.123
min work	0.436	0.834	1.421	1.486	1.287	1.008	0.870

Table 4: Throughput, Continuous Detection (interactive workload).

information. In both the non-interactive and interactive cases, for low levels of multiprogramming, the performance of the different victim selection algorithms is almost identical. There are just not many deadlocks to differentiate the algorithms. 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 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.[3] Table 5 gives the blocking ratio (the ratio of blocked lock requests to the number of transactions executed) and the restart ratio (the ratio of restarted transac-

[3] 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, the simple lock counting scheme would probably have to be modified to account for the number of objects associated with each lock.

tions to the number of transactions executed) for the interactive workload case. Note that the *fewest locks* blocking and restart ratios are the lowest among all of the victim selection criteria. Similar trends were seen in the non-interactive workload case.

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, 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 given 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 the algorithm unstable, and the algorithm would then be likely to perform poorly in high conflict situations. Our results bear out this hypothesis, at least to some extent. Of the algorithms studied here, only the *youngest* victim selection criterion has both of these stability properties; it performs moderately well. Both the *random* and *current blocker* criteria lack these stability properties, and consequently they have inferior performance at high multiprogramming levels. The *min work* and *min locks* criteria don't have these properties in the strict sense, as it is possible for the transaction holding the largest number of locks or having consumed the most

Victim Selection Criteria		Multiprogramming Level						
		5	10	25	50	75	100	200
current	block	0.055	0.148	0.422	0.886	1.399	1.833	2.431
	restart	0.011	0.029	0.089	0.161	0.263	0.383	0.548
random	block	0.070	0.143	0.426	0.945	1.514	2.027	3.063
	restart	0.016	0.029	0.085	0.167	0.278	0.395	0.661
youngest	block	0.059	0.154	0.434	0.894	1.363	1.807	2.323
	restart	0.015	0.032	0.082	0.160	0.252	0.353	0.526
min locks	block	0.059	0.152	0.436	0.870	1.361	1.734	2.114
	restart	0.012	0.028	0.080	0.159	0.247	0.343	0.438
min work	block	0.055	0.138	0.410	0.899	1.361	1.801	2.107
	restart	0.013	0.027	0.078	0.161	0.259	0.361	0.459

Table 5. Blocking and Restart Ratios, Continuous Detection (interactive workload).

resources to be "passed" by another transaction. They do have *pseudo-stability*, however — 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 probabilistically stable. For example, the transaction holding the most locks is unlikely to be passed by a large number of other transactions, so a transaction with a large number of locks is fairly safe from being restarted using *min locks* (as is an old transaction under the *youngest* policy).

4.2. Experiment 2: Periodic Detection

As was mentioned in Section 2, a problem with periodic deadlock detection is determining the time interval at which the detection algorithm should be periodically performed. Experiment 2 examined the performance of periodic detection for different time intervals. Based on the results of Experiment 1, the *min locks* policy was used for victim selection: we found in several earlier experiments (not reported here) that the best victim selection criteria is the best independent of whether continuous or periodic detection is performed, which is not surprising. Experiment 2 was performed for a low level of multiprogramming (10) and for a high level of multiprogramming (100). Both the high resource utilization (non-interactive workload) and the low resource utilization (interactive workload) cases were considered. The throughput results for this experiment are summarized in Table 6.

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 further increase contention if they are restarted later. In this experiment, we assumed that the waits-for graph is maintained in main memory,[4] and hence that cycle detection can be performed very efficiently. This is why

Type of Transactions	MPL	Detection Interval					
		250 msec.	500 msec.	1 sec.	5 sec.	10 sec.	50 sec.
Non-Interactive	10	5.170	5.166	5.159	5.080	4.840	1.675
	100	4.557	4.480	4.293	1.683	0.713	0.279
Interactive	10	0.804	0.804	0.760	0.779	0.761	0.708
	100	1.144	1.092	0.981	0.853	0.583	0.132

Table 6: Throughput, Periodic Detection, Varying Detection Intervals.

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

throughput does not decrease as the interval becomes fairly small.

4.3. 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 non-interactive and interactive workloads. Table 7 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 * σ(W), where σ(W) is the standard deviation of the request waiting time. We ran simulations where k was taken to be 0, 1, and 2. Table 8 summarizes the throughput results from this experiment for these three values of k. The best results were obtained for the dynamically adjusted timeout interval of avg(W) + σ(W) (i.e., for k = 1).

Type of Transactions	MPL	Timeout Interval					
		250 msec.	500 msec.	1 sec.	5 sec.	10 sec.	50 sec.
Non-Interactive	10	4.945	4.963	5.097	5.037	4.651	2.134
	100	3.195	3.204	3.193	3.385	3.521	1.054
Interactive	10	0.826	0.830	0.810	0.801	0.794	0.663
	100	1.580	1.581	1.622	1.683	1.605	0.818

Table 7: Throughput, Timeout Algorithm, Varying Timeout Intervals.

Type of Transactions	k	Multiprogramming Level						
		5	10	25	50	75	100	200
Non-Interactive	0	4.619	4.877	4.436	3.318	3.181	3.206	3.249
	1	4.666	4.886	4.784	3.777	3.165	3.244	3.327
	2	4.701	5.090	4.597	3.938	2.949	2.132	2.003
Interactive	0	0.434	0.810	1.498	1.602	1.565	1.619	1.574
	1	0.413	0.815	1.541	1.682	1.695	1.728	1.687
	2	0.429	0.755	1.122	1.066	0.909	0.736	0.804

Table 8: Throughput, Adaptive Timeout Algorithm.

4.4. 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 and periodic 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 and periodic detection. Based on the results of Experiment 2, the time interval for periodic detection was taken to be 500 milliseconds for the non-interactive workload case and 1 second for the interactive case. These time intervals were selected so that there was time gap between detections, and yet the algorithm was not seriously handicapped due to a large time interval for either class of workload. 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) + σ(W). We ran this experiment for both the non-interactive and interactive workloads. We will discuss the results for the two workload types separately here, as the results are significantly different for the two workload types.

4.4.1. The Non-Interactive Workload

The throughputs obtained with the alternative deadlock resolution strategies are summarized in Table 9 for the non-interactive type of transactions. We 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. Periodic detection has slightly inferior performance

as compared to continuous detection. 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 (which happen to be the critical resource in our experiments) is shown in Table 10. The useful utilization figures measure the total utilization minus the fraction of the resource utilization that was used to process transactions that were later restarted. Table 10 shows that, although the total disk utilization is basically the same for all strategies, the useful utilization is strongly correlated with throughput. Table 11 shows 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 4.1.

Both wound-wait and wait-die offer a balance between the extreme blocking and restart ratios, and each has intermediate performance. Note that wound-wait has both of the desirable stability properties described earlier.

Deadlock Resolution Strategy	Multiprogramming Level						
	5	10	25	50	75	100	200
continuous detection	4.744	5.172	5.275	5.099	4.839	4.581	4.342
periodic detection	4.728	5.166	5.296	5.089	4.816	4.480	4.258
wound-wait	4.688	5.040	4.982	4.392	3.906	3.807	3.819
wait-die	4.677	5.047	4.813	4.168	3.721	3.748	3.761
immediate-restart	4.588	4.794	4.368	3.315	3.158	3.208	3.281
running-priority	4.714	5.136	5.025	4.429	3.561	3.375	3.362
timeout	4.666	4.886	4.784	3.777	3.165	3.244	3.327

Table 9: Throughput, Alternative Strategies (non-interactive workload).

Deadlock Resolution Strategy		Multiprogramming Level						
		5	10	25	50	75	100	200
continuous detection	total	.84	.92	.97	.98	.98	.97	.98
	useful	.83	.91	.93	.89	.85	.80	.76
periodic detection	total	.84	.92	.97	.98	.98	.96	.96
	useful	.83	.91	.93	.89	.84	.79	.74
wound-wait	total	.84	.93	.98	.99	.99	.99	.99
	useful	.82	.88	.87	.77	.68	.66	.66
wait-die	total	.83	.93	.97	.98	.99	.99	.99
	useful	.82	.88	.84	.73	.65	.65	.65
immediate restart	total	.84	.93	.98	.99	.99	.99	.99
	useful	.80	.84	.76	.58	.54	.55	.57
running priority	total	.83	.93	.97	.99	.99	.99	.99
	useful	.83	.90	.88	.78	.62	.58	.58
timeout	total	.84	.93	.97	.99	.99	.99	.99
	useful	.82	.86	.84	.66	.55	.56	.57

Table 10: Disk Utilization, Alternative Strategies (non-interactive workload).

Victim Selection Criteria		Multiprogramming Level						
		5	10	25	50	75	100	200
continuous detection	block	0.052	0.128	0.326	0.728	1.190	1.681	2.057
	restart	0.008	0.025	0.063	0.125	0.207	0.304	0.427
periodic detection	block	0.064	0.145	0.388	0.860	1.368	1.921	2.310
	restart	0.010	0.024	0.060	0.128	0.213	0.320	0.425
wound-wait	block	0.023	0.046	0.130	0.305	0.480	0.521	0.519
	restart	0.029	0.074	0.182	0.412	0.647	0.696	0.700
wait-die	block	0.029	0.066	0.173	0.358	0.470	0.477	0.473
	restart	0.029	0.072	0.215	0.500	0.752	0.767	0.758
immediate restart	block	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	restart	0.060	0.131	0.337	0.846	0.948	0.934	0.901
running priority	block	0.050	0.114	0.295	0.616	1.060	1.204	1.221
	restart	0.011	0.032	0.111	0.300	0.648	0.750	0.764
timeout	block	0.057	0.131	0.386	0.913	1.359	1.394	1.326
	restart	0.034	0.103	0.180	0.553	0.889	0.863	0.832

Table 11: Blocking and Restart Ratios, Alternative Strategies (non-interactive workload).

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-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 Table 11, 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 one group of compatible lock requests. It is thus not surprising that, for low multiprogramming levels where conflicts are rare and hence the average wait queue length is likely to be small, running-priority exhibits behavior identical to continuous deadlock detection. It has high blocking ratios and low restart ratios, and it outperforms wound-wait or wait-die in these cases. However, running-priority lacks the desired stability properties — a restarted transaction is random, being the transaction which happened to be waiting at the time of a lock conflict (not a transaction chosen for age, lock, or resource related reasons). Consequently, for high multiprogramming levels, running-priority behaves similar to wound-wait or wait-die, but with random selection of the transaction to restart; thus, it has inferior performance as compared to these algorithms.

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.

Finally, the reason that the performance of periodic detection is somewhat inferior to that of continuous detection is its relatively higher blocking and restart ratios. A deadlocked transaction that has not been restarted and that is holding locks increases the probabilities of waiting and deadlocks.

4.4.2. The Interactive Workload

Table 12 gives the throughput results for the alternative deadlock resolution strategies for the interactive workload case. As in the case of non-interactive 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 non-interactive type workload are obtained. Wound-wait emerges as the overall best strategy here, and the two deadlock detection strategies have the worst performance. Running-priority performs a bit better

Deadlock Resolution Strategy	Multiprogramming Level						
	5	10	25	50	75	100	200
continuous detection	0.432	0.814	1.378	1.473	1.295	1.127	1.123
periodic detection	0.430	0.760	1.380	1.383	1.241	0.981	0.936
wound-wait	0.439	0.886	1.827	2.786	3.018	2.939	2.981
wait-die	0.426	0.843	1.567	1.936	1.936	1.900	1.900
immediate-restart	0.433	0.809	1.535	1.628	1.534	1.543	1.543
running-priority	0.425	0.808	1.584	2.084	1.960	2.011	1.988
timeout	0.413	0.815	1.541	1.682	1.695	1.728	1.687

Table 12: Throughput, Alternative Strategies (interactive workload).

than wait-die, and they both perform better than the timeout or immediate-restart algorithms. Timeout has slightly better performance than immediate-restart.

Table 13 shows the blocking and restart ratios for the alternative strategies, and Table 14 gives their total and useful disk utilizations. On the average, transactions in our experiments require 150 milliseconds of CPU time and 350 milliseconds of disk time and, so an internal think time of 10 seconds considerably reduces the average number of running (i.e., non-thinking) transactions. This reduces the demand on resources, and the resource utilizations

Victim Selection Criteria		Multiprogramming Level						
		5	10	25	50	75	100	200
continuous detection	block	0.059	0.152	0.436	0.870	1.361	1.734	2.114
	restart	0.012	0.028	0.080	0.159	0.247	0.343	0.438
periodic detection	block	0.082	0.178	0.512	1.023	1.511	2.016	2.358
	restart	0.019	0.029	0.078	0.161	0.254	0.370	0.451
wound-wait	block	0.033	0.073	0.221	0.458	0.529	0.549	0.545
	restart	0.039	0.084	0.246	0.605	0.677	0.697	0.682
wait-die	block	0.033	0.060	0.191	0.420	0.423	0.422	0.422
	restart	0.043	0.091	0.301	0.846	0.905	0.912	0.912
immediate restart	block	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	restart	0.067	0.158	0.516	1.099	1.191	1.178	1.163
running priority	block	0.059	0.126	0.360	0.920	1.234	1.172	1.203
	restart	0.018	0.051	0.196	0.653	0.939	0.887	0.905
timeout	block	0.080	0.143	0.545	1.515	1.651	1.586	1.676
	restart	0.072	0.112	0.350	0.972	1.024	0.998	1.022

Table 13: Blocking And Restart Ratios, Alternative Strategies (interactive workload).

Deadlock Resolution Strategy		Multiprogramming Level						
		5	10	25	50	75	100	200
continuous detection	total	.08	.15	.25	.29	.27	.24	.25
	useful	.08	.14	.24	.26	.22	.20	.19
periodic detection	total	.08	.14	.25	.27	.26	.21	.21
	useful	.08	.13	.24	.24	.22	.17	.16
wound-wait	total	.08	.17	.39	.74	.83	.82	.82
	useful	.08	.15	.32	.49	.53	.52	.52
wait-die	total	.08	.16	.34	.55	.55	.54	.54
	useful	.07	.15	.27	.34	.34	.33	.33
immediate restart	total	.08	.16	.39	.57	.55	.55	.54
	useful	.08	.14	.27	.28	.26	.27	.27
running priority	total	.07	.15	.33	.58	.64	.63	.63
	useful	.07	.14	.28	.36	.34	.35	.34
timeout	total	.08	.16	.35	.56	.56	.57	.56
	useful	.07	.14	.27	.29	.29	.30	.29

Table 14: Disk Utilization, Alternative Strategies (interactive workload).

decrease considerably. For high levels of multiprogramming, as the probability of conflicts increases, the blocking ratios increase; due to the fact that the transactions think while holding locks, the 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 deadlock detection. Although the restart ratio increases due to deadlocks as the multiprogramming level is increased, blocking increases at much faster rate than deadlocks. The immediate-restart strategy, which is based purely on restarting a conflicting transaction, does 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 [Tay84]). 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 strategy that balances the restart and the blocking ratios, as is the case with the wound-wait strategy. Observe that the total and useful disk utiliza-

tions are also largest with this strategy (see Table 14). 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 an 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 waiting for several active readers, and then to be subsequently restarted when a new reader arrives in the queue — 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.

4.5. Comparison With Other Results

It is interesting to compare our results on the performance of deadlock resolution strategies with the simulation results of Munz and Krenz [Munz77], and those of Balter, Berard and Decitre [Balt82]. We consider each of these studies in turn.

In [Munz77], the relative performance of 11 victim selection algorithms were 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 non-interactive 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 non-interactive case.

In [Balt82], the relative performance of deadlock detection (with an unspecified victim selection algorithm), wound-wait, an "improved" version of wound-wait, wait-die, and a "modified deadlock detection" scheme was compared. Their modified deadlock detection scheme was a non-preemptive 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, neither the details of the simulation model nor any simulation parameters were specified in [Balt82].

Our conclusions on 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. We conjecture that Balter, Berard and Decitre either did not consider physical resources in their model, assumed them to be infinite, or studied only interactive workloads. 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 [Fran83]. 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 [Balt82, Fran83]. 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 older than 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 bit less pronounced in this case (wound-wait was 40% better than running-priority instead of 50% 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.

5. 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 that have finite physical resources and workloads such that a number of transactions tend to be competing for these resources at all times (i.e., primarily non-interactive workloads). If the transactions are primarily interactive, however, with long think times during which locks are held, then a deadlock detection strategy causes excessive blocking and performs 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 consistently outperformed periodic deadlock detection due to the higher blocking and restart ratios associated with periodic detection.

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 to be restarted (*guaranteed progress*), and the second being that transactions cannot be restarted over and over arbitrarily many times (*no repeated restarts*). The absence of these properties may cause an algorithm to become unstable in high conflict situations. Among the victim-selection algorithms 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 selection 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 criteria was was found to be better than the other critera 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 probabiity 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 very 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.

Our experiments were confined to the case of centralized database 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.

ACKNOWLEDGEMENTS

We wish to thank Rudd Canaday 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

[Agra83a] R. Agrawal, M. Carey and D. J. DeWitt, "Deadlock Detection Is Cheap", *ACM-SIGMOD Record* 13, 2 (Jan. 1983), 19-34.

[Agra83b] R. Agrawal and D. J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", Computer Sciences Tech. Rep. #497, Univ. Wisconsin, Madison, Feb. 1983.

[Agra85] R. Agrawal, M. J. Carey and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications", *Proc. ACM-SIGMOD 1985 Int'l Conf. on Management of Data*, May 1985, to appear. Also available as Computer Sciences Tech. Rep. #567, Univ. Wisconsin, Madison, Dec. 1984.

[Aho75] A.V. Aho, E. Hopcraft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Chapter 5), Addison-Wesley, Reading, Mass., 1975.

[Balt82] R. Balter, P. Berard and P. Decitre, "Why the Control of Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management, *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Aug. 1982, 183-193.

[Beer81] C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Proc. 7th Int'l. Conf. on Very Large Data Bases*, Sept. 1981.

[Bern78] 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, 3 (May 1978), 154-168.

[Bern81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys* 13, 2 (June 1981), 185-221.

[Bern82] P. A. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control", *Proc. 8th Int'l. Conf. on Very Large Data Bases*, Sept. 1982, 62-76.

[Care83] M.J. Carey, *Modeling and Evaluation of Database Concurrency Control Algorithms*, Computer Sciences Dept., Univ. California, Berkeley, Dec. 1983, Ph.D. Thesis.

[Care84] M.J. Carey and M.R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proc. 10th Int'l. Conf. on Very Large Data Bases*, Aug. 1984, 107-118.

[Cham74] D. D. Chamberlain, R. F. Boyce and I. L. Traiger, "A Deadlock-Free Scheme for Resource Locking in a Data Base Environment", *Proc. IFIP 1974*, , 340-343.

[Chin84] W. N. Chin, "Some Comments on 'Deadlock Detection is Cheap' in SIGMOD Record Jan. 83", *ACM-SIGMOD Record*, March 1984, 61-63.

[Coff71] E. G. Coffman, M. J. Elphic and A. Shoshani, "System Deadlocks", *ACM Computing Surveys* 3, 2 (June 1971), 67- 78.

[Eswa76] 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* 19, 11 (Nov. 1976), 624-633.

[Fran83] P. Franaszek and J.T. Robinson, *Limitations of Concurrency in Transaction Processing*, Rep. RC-10151, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Aug. 1983

[Glig80] V. D. Gligor and S. H. Shattuck, "On Deadlock Detection in Distributed Systems", *IEEE Trans. Software Eng.* SE-6, 5 (Sept. 1980).

[Gray79] J. N. Gray, "Notes on Database Operating Systems", in Lecture Notes in Computer Science 60, *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmuller (ed.), Springer-Verlag, New York, 1979.

[Gray81] 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*, Rep. RJ3066, IBM Research Lab., San Jose, California, Feb. 1981.

[Holt72] R. C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys* 4, 3 (Sept. 1972), 179- 196.

[King73] P. F. King and A. J. Collmeyer, "Database Sharing - An Efficient Method for Supporting Concurrent Processes", *Proc. AFIPS 1973 Natl. Computer Conf.*, 1973, 271-275.

[Kort82] H. F. Korth, "Edge Locks and Deadlock Avoidance in Distributed Systems", *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Aug. 1982, 173-182.

[Lome77] D. B. Lomet, "A Practical Deadlock Avoidance Algorithm for Data Base Systems", *Proc. ACM-SIGMOD 1977 Int'l Conf. on Management of Data*, Aug. 1977, 122-127.

[Lome80] D. B. Lomet, "Deadlock Avoidance in Distributed Systems", *IEEE Trans. Software Eng.* SE-6, 3 (March 1980). .

[Macr76] P. P. Macri, "Deadlock Detection and Resolution in a CODASYL Based Data Management System", *Proc. ACM-SIGMOD 1976 Int'l Conf. on Management of Data*, June 1976, 45- 49.

[Mena78] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Databases", *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, Aug. 1978.

[Mitc84] D. Mitchell and M. J. Merritt, "Distributed Algorithm for Deadlock Detection and Resolution", *Proc. of ACM-SIGACT-SIGMOD Conf. on Principles of Distributed Computing*, Aug. 1984.

[Munz77] R. Munz and G. Krenz, "Concurrency in Database Systems - A Simulation Study", *Proc. ACM-SIGMOD 1977 Int'l Conf. on Management of Data*, Aug. 1977, 111-120.

[Newt79] G. Newton, "Deadlock Prevention, Detection and Resolution: An Annotated Bibliography", *ACM-SIGOPS Operating Systems Review* 13, 4 (April 1979), 33-44.

[Ober82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Trans. Database Syst.* 7, 2 (June 1982). .

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

[Ries77] D.R. Ries and M.R. Stonebraker, "Effects of Locking Granularity in a Database Management System", *ACM Trans. Database Syst.*, 2, 3, (Sept. 1977), 233-246.

[Ries79a] D.R. Ries, *The Effects of Concurrency Control on Database Management System Performance*, Computer Sciences Dept., Univ. California, Berkeley, April 1979, Ph.D. Thesis.

[Ries79b] D.R. Ries and M.R. Stonebraker, "Locking Granularity Revisited", *ACM Trans. Database Syst.*, 4, 2, (June 1979), 210-227.

[Rose78] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems", *ACM Trans. Database Syst.* 3, 2 (June 1978), 178-198.

[Ston79] M. R. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Trans. Software Eng.* SE-5, 3 (May 1979), 188-194.

[Tay84] Y. C. Tay, *A Mean Value Performance Model for Locking in Databases*, Tech. Rep.-04-84, Harvard Univ., Cambridge, Mass., Feb. 1984. Ph.D. Thesis.