Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

ON THE PERFORMANCE OF THE IMMEDIATE RESTART CONCURRENCY CONTROL POLICY

Abbas Asadi-Arbabi

Dimension Software Systems
Las Colins, Texas 75038

Sayed A. Banawan

Department of Computer Science
University of Houston
Houston, Texas 77204

ABSTRACT

In database systems several transactions are usually allowed to proceed concurrently to exploit the potential parallelism. However, such concurrent execution must be controlled to maintain the consistency and integrity of the data. This paper studies the performance of a concurrency control algorithm based on locking called the *Immediate Restart*. Under this policy a transaction that fails in acquiring a lock is immediately aborted (and replaced by another transaction to maintain the same multiprogramming level). The aborted transaction is attempted once again after some delay. We discuss the factors that influence the algorithm performance, namely data contention and resource contention. We argue that selecting the appropriate delay is crucial to the performance of the policy. If the delay is too short the transaction is likely to fail again because the locks that caused it to be aborted are still being held by other transactions. On the other hand, a very long delay, while it increases the probability of success, it also increases the transaction response time. In both cases, the overall performance may not be optimal. We propose a new method for estimating the optimal delay. It is based on examing the current state of system resources. Our results indicate that it performs well and is better than the traditional algorithm that uses the "running average" transaction response time as the restart delay.

1 INTRODUCTION

A database is a collection of data objects that may be shared by many users who read and update them through transactions. A transaction is a sequence of atomic actions, so called *actions*. An action is the smallest unit of work (i.e., read or write). When transactions run concurrently the system must control the timing of read and write operations of transactions to maintain the consistency and integrity of the data. A conflict arises between two transactions if both attempt to access the same data object and one of them writes it. The mechanism by which the system controls the behavior and timing of transactions so that it may detect and resolve conflicts is called the *concurrency control* mechanism.

Concurrency control has been the focus of much research that produced many algorithms. Most of these algorithms uses one of three approaches: locking, timestamps and optimistic. The various concurrency control algorithms differ in the time when they *detect* a conflict and the way they *resolve* it. A commonly used algorithm that is based on locking is called *Immediate Restart*. Under this policy, transactions set read locks on objects that they read, and these locks are later upgraded to write locks for objects they also write. If a lock request is denied, the requesting transaction is immediately aborted and may be replaced by another transaction immediately. The aborted transaction is restarted after some delay. This delay is necessary to prevent the same conflict from occurring repeatedly. The main advantage of this policy is that it does not cause transaction to blocks, which can potentially lead to deadlocks that are expensive to detect and resolve. However, as some researchers (Agrawal, Carey, and Livny 1982), have reported the immediate restart policy may not perform as well as other policies. In this paper, we focus on the performance aspects of the immediate restart policy. First, we study the factors that influence the algorithm performance and quantify their effect using a simulation model. Then, we argue that using the running average response time may not yield optimal performance. Finally, we propose a new method for selecting the restart delay that is based on analyzing the current state of system resources and show that it consistently performs better than the traditional method.

The paper is organized as follows. Section 2 de-

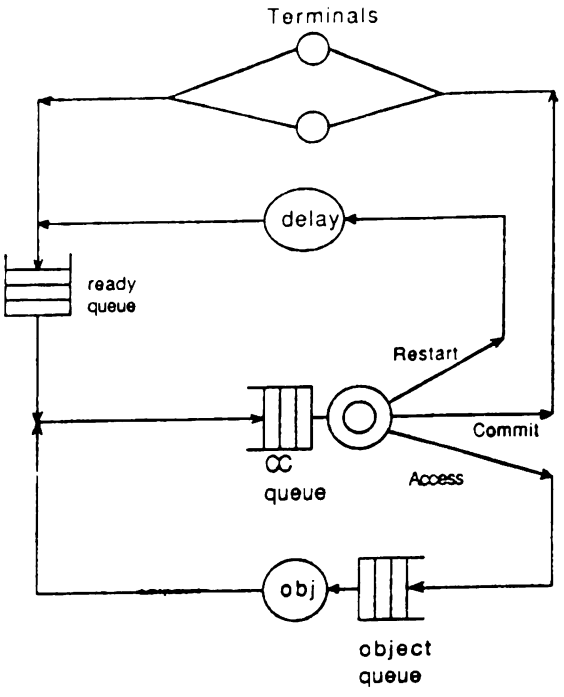

Figure 1: Logical View of the Model

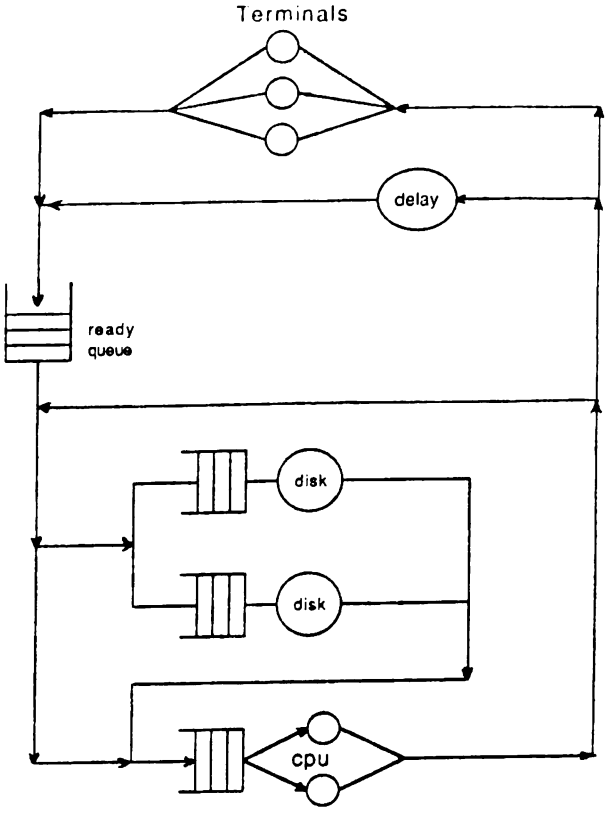

Figure 2: Queuing Model

scribes the logical view of the system and the underlying queuing model. Section 3 defines and lists the model parameters. In Section 4 we evaluate system performance using simulation. Section 5 presents the new method for selecting the restart delay and shows how it improves system performance. Finally, Section 6 summarizes the paper.

2 SYSTEM MODEL

Our model is a variation of that proposed by Ries and Stonebraker (Ries and Stonebraker 1977) that has been used as a baseline model in a number of studies, e.g., Agrawal, Carey and Livny (1987). It captures the basic elements of a database environment, including both the *users* who issue transactions, and the *hardware resources* that execute them.

The logical view of the model is shown in Figure 1. We assume a fixed number of terminals that generate transactions. There is a limit to the number of *active* transactions allowed in the system determined by the multiprogramming level (MPL). A transaction is considered active if it is either receiving or waiting for service at a resource, but not waiting for locks. A new transaction is allowed to execute if the number of currently active transactions is less than the multiprogramming level, otherwise the transaction is placed at the end of the *ready queue* where it waits until an active transaction terminates.

A newly issued transaction that becomes active is placed at the end of the *concurrency control queue* (CC) where it makes its first request. A concurrency control request is a request to lock an object. If the request is granted (i.e., the transaction acquires the lock) it proceeds to the object queue so that it may access the data followed by some processing. If the transaction has further requests it returns to the end of the concurrency control queue. A transaction that fails to acquire a lock, because its request has been denied, is aborted and sent to the ready queue after a restart delay. Eventually, a transaction will access all its objects and commit.

Underlying the logical view in Figure 1 are two types of hardware resources: processing units and I/O devices. The corresponding queuing model is shown in Figure 2. It consists of a collection of terminals, CPUs, and I/O disks.

When a transaction needs CPU service it is assigned a free server if one exists, otherwise the transaction joins the CPU queue and waits until one becomes available. In other words, the CPU may be thought of as being a pool of identical servers that serve a single queue, All service disciplines are as-

sumed to be FCFS.

3 MODEL PARAMETERS

We assume that performing an I/O operation to read or write a data object has an average service time *tdisk* while data processing at the CPU has an average service time *tcpu*. Having completed a transaction, a user waits on the average an amount of time equal to *think_time* before issuing the next one. All service times as well as the think time are assumed to be exponentially distributed. A transaction has a length *tran_size* determined by the number of data objects it accesses. The objects are chosen randomly (without replacement) from among all the objects in the database. The "without replacement" method is required since it does not make sense for a transaction to request a lock which it already holds. The percentage of I/O operations that are write requests is given by *wr_pr*. Unless otherwise mentioned, a transaction accesses all the objects of a database with the same probability. A summary of the model parameters and their definitions is shown in Table 1.

To evaluate system performance it is desirable to use an analytic approach based on *queuing network models* (Lazowska et al. 1985). Unfortunately, the underlying queuing model doesn't have a product-form solution due to locking. Because of the intractability of solving general queuing network models, simulation was deemed necessary to study the performance. As such, a simulation model based on the event-driven approach was coded in C on top of a special package called *SMPL* (MacDougall 1987) modified properly to handle concurrency control.

In the simulation model, each transaction issued from a terminal comes with a list of object indices in the database that it accesses. This list is called a *script*. Associated with each object index in the script list is a type that distinguishes the lock requested: shared (read) or exclusive (write). All acquired locks are released together at the transaction completion or abortion time. While a transaction is active a backup copy of the original objects accessed by the transaction is saved. If a lock request is denied the transaction is aborted and the original objects that have been updated by this transaction are restored to their original values in the database from the backup copy. On the other hand, if the transaction commits, the backup copy is discarded. This recovery scheme is based on the *write-ahead* policy that has been shown to preserve the database consistency in both single and multiple user environments (Elmasri and Navathe 1989).

Table 2: Simulation Parameter Settings

Parameter	Value
db_size	1000 or 10,000 pages
tran_size	8 pages
wr_pr	25%
think_time	1 second
num_terms	200
MPL	2 to 200
tcpu	18 millisecond
tdisk	35 millisecond
num_cpus	1, 5, 10, 50, ∞
num_disks	2, 10, 20, 100, ∞
num_units	2
gran_size	1 page

4 SIMULATION RESULTS

In this section we study the performance of the immediate restart policy that uses the running average service time as the restart delay. The effect of data contention and resource contention individually and together on performance is studied.

Performance Metrics

The primary performance metric used in this study is system throughput, which is directly related to resource utilizations. In some cases, the conflict ratio is also reported. The conflict ratio is defined as the average number a transaction is aborted. For example, a conflict ratio of 0.16 can be interpreted as follows: each 6 transactions 1 is aborted once while the others commit in the first run.

Parameter Setting

Some model parameters have fixed values during all simulation experiments. These values are listed in Table 2.

Hardware resources are assumed to come in units. Each unit consists of a CPU server and *num_units* disks. The parameter *num_units* is set to 2. In other words, for each CPU server in the system, there are 2 disks. The disk service time was chosen to be close to typical values provided by the manufacturer. The CPU service time was chosen so that both the CPU and disks have more or less the same utilization. Thus, these values result in a balanced system so that no device becomes a bottleneck. The restart delay has an exponential distribution with mean equal to the running average transaction response time.

Table 1: Model Parameters

Parameter	Definition
db_size	number of objects in database
tran_size	size of transaction
wr_pr	proportion of write operations to transaction size
delay	mean transaction restart delay
think_time	mean time between transactions
num_terms	number of terminals
MPL	multiprogramming level
tcpu	mean CPU time for processing a data object
tdisk	mean I/O time for accessing a data object
num_cpus	number of CPU servers
num_disks	number of I/O disks
num_units	number of disks in each unit
gran_size	size of a granule

In general, database systems performance is affected by two types of contention: *data contention* and *resource contention*. Data contention arises when transactions attempt to access a data object whose lock cannot be shared. In this case, the youngest transaction is usually aborted. Resource contention arises when there are more transactions ready to execute than hardware resource (e.g., memory, CPU or I/O disks). Four different cases can be identified:

- low data contention and low resource contention
- low data contention and high resource contention
- high data contention and low resource contention
- high data and high resource contention

These cases are discussed below.

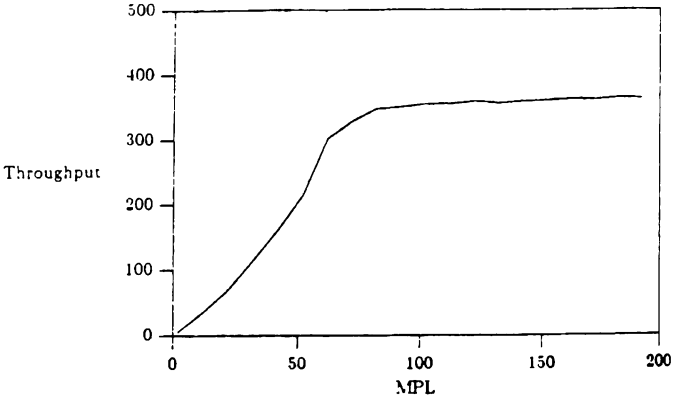

Figure 3: Throughput of a Large Database with Infinite Resources

4.1 Data Contention

As was stated earlier, data contention arises when transactions attempt to acquire locks that cannot be shared. Let *req_lock* be the maximum number of locks that can be held concurrently by all transactions. *req_lock* is related to the multiprogramming level and the transaction size as follows:

$$req\_lock(\text{MPL}) = \text{MPL} * tran\_size.$$

The data contention can be estimated as the average number of locks per object, that is:

$$data\_con(\text{MPL}) = req\_lock(\text{MPL})/num\_gran$$

where $num\_gran = db\_size/gran\_size$.

A database system performs best when both data contention and resource contention are low. Data contention is low at large systems, i.e., systems with large number of objects compared to the multiprogramming level. Resource contention can be eliminated by providing enough hardware resources so that no transaction needs to wait for service. Figures 3 and 4 present the results of simulating a large database system with an infinite number of resources so that both types of contention are low. The system has 10,000 data objects. Figure 3 shows the throughput vs. MPL, while Figure 4 shows the corresponding conflict ratio. Note that the throughput curve levels off after certain MPL ($\approx$ 80) is reached. This is due to the fact that beyond 80, all issued (ready) transactions are already active and the rest are waiting for users' input.

The effect of data contention can be isolated by examining the performance of a database with a smaller number of objects. Figure 5 shows the result of simulating a system with only 1000 pages, while keeping

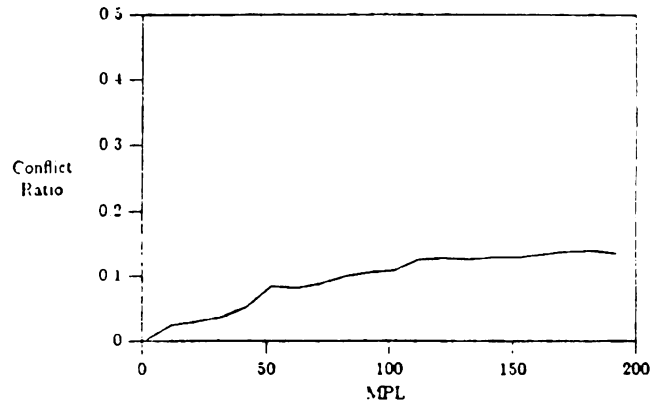

Figure 4: Conflict Ratio of a Large Database with Infinite Resources

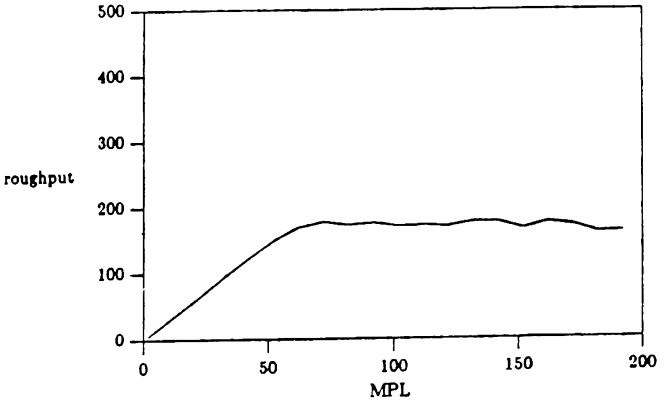

Figure 5: Throughput of a Small Database with Infinite Resources

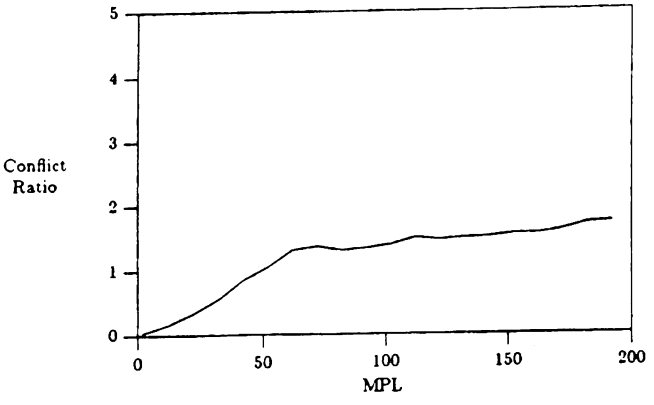

Figure 6: Conflict Ratio of a Small Database with Infinite Resources

the number of hardware resources infinite. The corresponding conflict ratio is shown in Figure 6. Compared with Figure 4, high data contention yields a large conflict ratio that reduces system throughput substantially.

4.2 Resource Contention

The previous results were obtained under the assumption of infinite resources which eliminates resource contention entirely. The effect of resource contention on performance can be isolated by studying a system with a *large* number of data objects, and a small number of hardware resources. Figure 7 shows the throughput as a function of MPL in a system with only 1 unit of hardware resources. We observe that while the throughput is still a non-decreasing function of MPL, it levels off at a much smaller value of MPL.

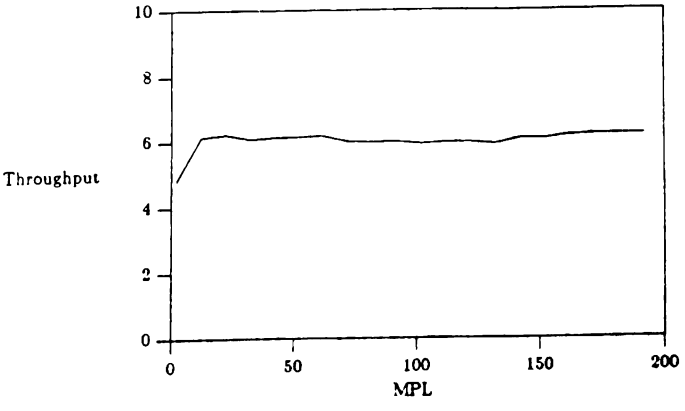

Figure 7: Throughput of a Large Database with 1 Resource Unit

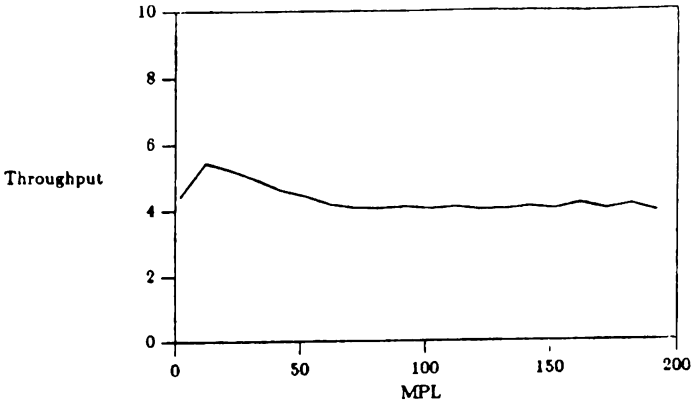

Figure 8: Throughput of a Small Database with 1 Resource Unit

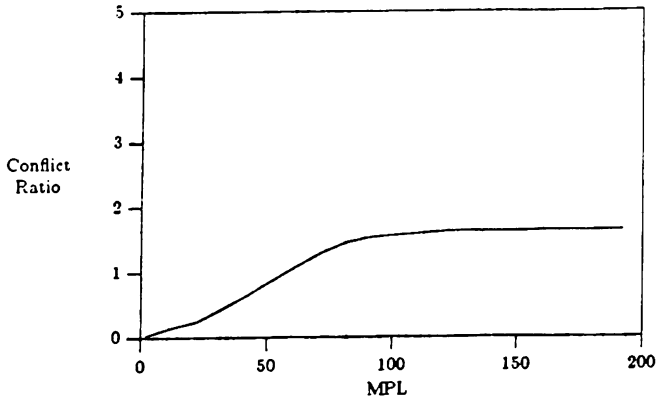

Figure 9: Conflict Ratio of a Small Database with 1 Resource Unit

4.3 Resource Contention And Data Contention

To study the effect of both types of contention on system performance we assume a database of 1000 objects and allow 1 unit of hardware resources. The simulation results of this case are shown in Figures 8 and 9.

In this case, the observed system throughput first increases as MPL increases until certain value, after which it decreases for a while then it becomes flat. This behavior can be explained as follows. Initially, both data and resource contention are low so the throughput increases with MPL until data contention reaches a level in which conflicts becomes so high that a form of thrashing takes place. As MPL increases further, resource contention stabilizes the throughput by making transaction wait longer at resource queues instead of being aborted and rescheduled.

Multiple Resource Case

Clearly, to eliminate resource contention the system does not have to have an infinite number of resources. Indeed, it is possible to eliminate resource contention by providing enough resources so that no transaction has to wait for service. Since the maximum number of active transactions cannot exceed MPL, providing MPL units of resources should eliminate resource contention. The question becomes whether we can eliminate resource contention by fewer number of resources. In other words, adding one resource unit at a time, what is the smallest number of resources needed so that the system exhibits performance similar to one with infinite resources, where no resource contention exists. To answer this question, we simulated a system with 5, 10 and 50 resources units. It was found that the maximum throughput achievable with 50 units is the same as with 10 units, which means that resource contention can be eliminated with only 10 units. This is true because data contention limits the number of concurrently active transaction regardless of MPL.

5 DYNAMIC RESTART DELAY

In the immediate restart algorithm, when a transaction is aborted due to a lock conflict, it is rescheduled after some delay. Selecting the right value to delay an aborted transaction before its next attempt for execution is very critical to the performance of restart-oriented concurrency control algorithms. On the one hand, an unreasonably large value of delay will put some transactions in the waiting state while they could execute, which wastes system resources and degrades performance. On the other hand, an extremely short delay will cause a transaction to restart before its lock requests can be granted so it has to be aborted again. This also wastes system resources and degrades performance.

In the traditional immediate restart algorithm, the delay is usually set equal to the observed average response time. The rationale behind this choice is that during that time an active transaction is likely to commit and release the locks that are needed by the aborted transaction. Note that this delay increases the average response time of transaction which, in turn, causes an additional increase to the restart delay. Consequently, this approach to calculating the restart delay may not yield optimal performance as it limits the potential parallelism unnecessarily. This explains the inferior performance of the algorithm compared with blocking as reported in some previous studies (Agrawal, Carey and Livny 1987).

In this section we propose a new method for calculating the value of the restart delay that dynamically sets this value according to the system workload and resource contention. It differs from the traditional method in that instead of using the average transaction response time, the time until next completion is determined from examining the system state. This time depends on system load, transaction size, resource contention and service demands at hardware resources. It is used as the current value of the restart delay.

To understand the new method, let us assume that transaction A has just been aborted because of a conflict with transaction B that is currently holding locks needed by transaction A. If we can determine the time required for transaction B to terminate, then this time would be the *exact* time that transaction A should be delayed before restarting it once again.

Recall that a transaction consists of a sequence of I/O operations, called actions, interleaved with CPU processing. Furthermore, the resource *residence time*, that is the time to receive service at a resource, depends on the number of transactions currently queued for that resource and their service demands. Assuming that the system can be approximated by a *separable* queuing network (Lazowska et al. 1984), the residence time at resource k can be calculated using the formula:

$$R_k(\text{MPL}) = D_k * [1 + A_k(\text{MPL})]$$

where R_k is the residence time, D_k is the average service demand at resource k, and A_k is the number of transactions seen at resource k when a new transaction arrives. It follows that the CPU and disk residence times can be estimated as follows:

$$R_{cpu}(\text{MPL}) = tcpu * [1 + A_{cpu}(\text{MPL})]$$

and

$$R_{disk}(\text{MPL}) = tdisk * [1 + A_{disk}(\text{MPL})]$$

Now the total execution time of a transaction, *exe_time*, at multiprogramming level MPL can be calculated using the formula:

$$exe\_time(\text{MPL}) = [R_{cpu}(\text{MPL}) + R_{disk}(\text{MPL})] * tran\_size$$

This value is used as the delay to reschedule the aborted transaction. The advantage of this method for calculating the restart delay is that it takes into account the "current" state of different resources in the system, not just the "long term" average response time.

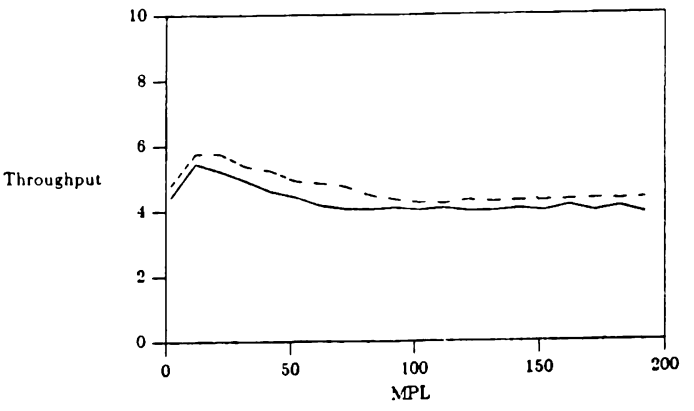

Figure 10: Comparison of Throughput under Both Algorithms

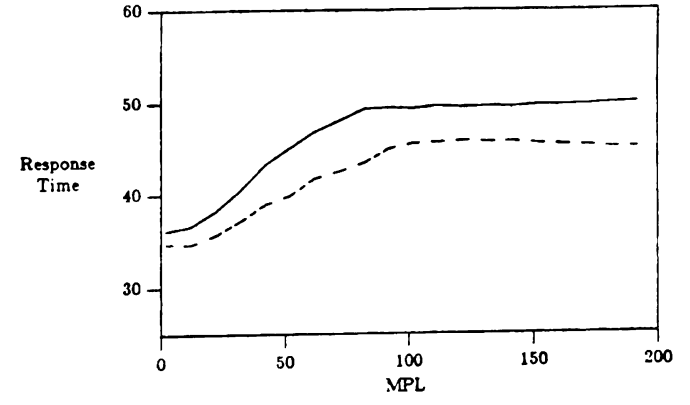

Figure 11: Comparison of Response Time under Both Algorithms

Figures 10 and 11 show the throughput and response time of the new algorithm (shown as dashed-lines) and compare them with those of the old algorithm (shown as solid lines). The new algorithm consistently perform better over the entire range of multiprogramming level.

Non-Uniform Access To Database

Up to this point, database access was assumed to be uniform, which means that all objects are accessed with equal probabilities. To study system performance under non-uniform access we simulated a system under the 80-20 access pattern (Lin and Notle 1982). In this case, 80% of the transactions access a small region of the database, called the *hot spot*. The other 20% access objects in the remaining large part. Therefore, data contention becomes very high for the hot spot and very low for the rest of the database. The high level of data contention causes an increase in the number of restarts which degrades performance. Figure 12 compares system throughput under non-

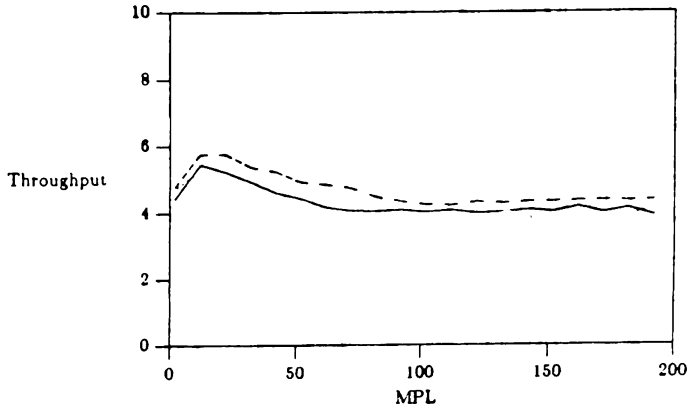

Figure 12: Comparison of Throughput under Non-uniform Access

uniform access using both the old and new method of computing the restart delay. Again, the new method yields higher throughput over the entire range of multiprogramming.

6 SUMMARY

In this paper, we used a simulation model that captures most realistic aspects of database systems to study the performance of a concurrency control policy that aborts the transactions that fail to acquire their locks and reschedule them after some delay. The effect of data contention and resource contention individually and together on system performance was presented. A new method for calculating the restart delay that takes into account the current state of system resources was proposed and shown to yield better performance than the traditional method which uses the running average of transaction response time under both uniform and non-uniform access.

REFERENCES

Agrawal, R., M. Carey, and M. Livny. 1982. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems* 12: 609-654.

Carey, M., and M. Stonebraker. 1984. The performance of concurrency control algorithms for DBMSs. In *Proceedings of the 1984 International Conference on Very Large Database*, 107-118. Singapore.

Devor, C., and C. Carlson. 1982. Structural locking mechanisms and their effects on database management system performance. *Information Systems* 7: 345-358.

Elmasri, R., and S. Navathe. 1989. *Fundamentals of Database Systems.* New York: BenjaminCummings.

Lin, W., and J. Notle. 1982. Read only transactions and two phase locking. In *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*, 85-93. Pittsburgh, Pennsylvania.

Lazowska, E., J. Zahorjan, J., G. S. Graham, and K. Sevcik. 1984. Quantitative system performance: Computer system analysis using queueing network models. Englewood Cliffs: Prentice-Hall.

MacDougall, M. 1987. Simulating Computer Systems: Techniques and Tools. Cambridge: MIT press.

Ries, D., and M. Stonebraker. 1977. Effect of locking granularity in database management system. *ACM Transactions on Database Systems* 2: 233-246.

Tay, Y., N. Goodman, and R. Suri. 1985. Locking performance in centralized database management. *ACM Transactions on Database Systems* 10: 415-462.

ACKNOWLEDGMENTS

This work has been partially supported by a grant from the University of Houston (RIG 88008).

AUTHOR BIOGRAPHIES

Abbas Asadi-Arbabi received a Master in computer science from the University of Houston in May 1990. His research interest is database system performance. He is currently with Dimension Software Systems in Las Colins, Texas.

Sayed A. Banawan is an assistant professor in the Department of Computer Science at the University of Houston. His research interests are performance evaluation of computer systems and load sharing in distributed and parallel systems.