On Being Optimistic About Real-Time Constraints

by

Jayant R. Haritsa
Michael J. Carey
Miron Livny

Computer Sciences Technical Report #906
January 1990

On Being Optimistic about Real-Time Constraints

Jayant R. Haritsa
Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

To appear in
Proceedings of the 1990 ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems (PODS)

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

On Being Optimistic about Real-Time Constraints

Jayant R. Haritsa

Michael J. Carey

Miron Livny

Computer Sciences Department

University of Wisconsin

Madison, WI 53706

ABSTRACT — Performance studies of concurrency control algorithms for conventional database systems have shown that, under most operating circumstances, locking protocols outperform optimistic techniques. Real-time database systems have special characteristics – timing constraints are associated with transactions, performance criteria are based on satisfaction of these timing constraints, and scheduling algorithms are priority driven. In light of these special characteristics, results regarding the performance of concurrency control algorithms need to be re-evaluated. We show in this paper that the following parameters of the real-time database system – its policy for dealing with transactions whose constraints are not met, its knowledge of transaction resource requirements, and the availability of resources – have a significant impact on the relative performance of the concurrency control algorithms. In particular, we demonstrate that under a policy that discards transactions whose constraints are not met, optimistic concurrency control outperforms locking over a wide range of system utilization. We also outline why, for a variety of reasons, optimistic algorithms appear well-suited to real-time database systems.

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

1. INTRODUCTION

We define a Real-Time Database System (RTDBS) to be a transaction processing system which attempts to satisfy the timing constraints associated with each incoming transaction. Typically, a constraint is expressed in the form of a *deadline*, that is, the user submitting the transaction would like it to be completed before a certain time in the future. Thus, greater value is associated with processing transactions before their deadlines as compared to completing them late. How early a transaction completes relative to its deadline is not as important as whether or not it completes by the deadline. Therefore, in contrast to a conventional DBMS where the goal is to minimize response times, here the emphasis is on satisfying timing constraints of transactions.

At any given time, all transactions in the system can be divided into two categories: *feasible* transactions and *late* transactions. A feasible transaction still has a possibility of meeting its deadline. A late transaction has either already missed its deadline or has no chance of successfully meeting it. The system executes feasible transactions until they either complete before their deadline or are detected to be late. Various application-dependent policies exist to deal with late transactions. As mentioned earlier, satisfaction of transaction timing constraints is the primary goal, rather than considerations of fairness. Therefore, the algorithms which resolve resource contention and data contention can be reasonably expected to be priority-driven, with the transaction priority assignment scheme being tuned to reducing the number of late transactions. In summary, RTDBSs differ from conventional DBMSs in that transactions have deadlines, the primary performance criterion is number of deadlines met and not response time, and the scheduling algorithms are driven by priority considerations rather than fairness considerations.

Recent papers by Abbott and Garcia-Molina [Abbo88, Abbo89] have addressed the problem of scheduling transactions in an RTDBS with the objective of minimizing the percentage of late transactions. In their work, they focus primarily on the scheduling issue, and use locking as the underlying concurrency control mechanism. In this paper we shift the focus to studying the relative performance of two well-known classes of concurrency control algorithms – locking protocols and optimistic techniques – in an RTDBS environment.[1] Earlier studies of these concurrency control algorithm classes for conventional DBMSs (e.g. [Agra87, Care88]) have concluded that, under most operating circumstances, locking algorithms outperform optimistic algorithms. In light of the significant differences between RTDBSs and conventional DBMSs outlined above, it is possible that these previous results will

[1] We assume here that *serializability* [Eswa76] is the required level of database correctness.

not hold true in an RTDBS environment. This possibility motivated our investigation of the performance of concurrency control algorithms in real-time database systems.

We show in this paper that for a real-time database system, the policy for dealing with late transactions, apriori knowledge of transaction resource requirements, and the availability of resources, all have a significant impact on the relative performance of the concurrency control algorithms. In terms of the late policy, late transactions may either have to be run to completion or alternatively, they may be considered as having lost all value and hence be discarded from the system. In the former case, the system is said to have *soft* deadlines, and in the latter, we say that the system has *firm* deadlines.[2] Real-world examples of systems having these different types of deadlines are given in [Abbo88]. Regarding workload information, apriori knowledge of individual transaction resource requirements can help the algorithms that resolve resource and data contention to make better decisions. Depending on the nature of the application, this information may or may not be available. Finally, locking and optimistic algorithms behave very differently with regard to resource utilization, and hence the availability of resources has an impact on their relative behavior.

The major result of this paper is that in a system with firm deadlines, *optimistic concurrency control outperforms locking* over a wide range of system utilization. We also show that when transaction resource requirements are known in advance, the performance of both algorithms is considerably improved and their performance difference shrinks significantly. In contrast, for a system with soft deadlines, the situation is not as clear-cut, in terms of how performance is to be measured, due to the presence of more than one performance metric. We provide some insight into the issues involved in this scenario. We also show that when resources are plentiful, thus making resource contention a non-issue, optimistic concurrency control performs much better than locking. A detailed simulation model of an RTDBS was used to derive the performance statistics.

The paper is organized in the following fashion: Section 2 describes the functioning of the concurrency control algorithms chosen for study. In Section 3, we provide the motivation behind why one might expect optimistic algorithms to perform better than locking in an RTDBS environment. Then, in Section 4, we describe our simulation model, while Section 5 highlights the results of our experiments. Finally, Section 6 summarizes the main conclusions of the study and outlines future avenues to explore.

[2] A *hard* deadline system guarantees satisfaction of *all* transaction deadlines, while a *firm* deadline system only guarantees that late transactions are not provided any further service.

2. CONCURRENCY CONTROL ALGORITHMS

Locking and optimistic algorithms both come in several flavors. In our study, we will compare a specific locking protocol with a specific optimistic technique. These particular instances were chosen because they are well suited to the RTDBS environment, are of comparable complexity, and are general in their applicability. The details of the selected algorithms are explained below.

In classical two-phase locking (2PL) [Eswa76], transactions set read locks on objects that they read, and these locks are later upgraded to write locks for the objects that are updated. If a lock request is denied, the requesting transaction is blocked until the lock is released. Read locks can be shared, while write locks are exclusive. For an RTDBS, two-phase locking needs to be augmented with a priority scheme to ensure that higher priority transactions are not delayed by lower priority transactions. In the *High Priority* scheme [Abbo88], all data conflicts are resolved in favor of the transaction with the higher priority. When a transaction requests a lock on an object held by other transactions in a conflicting lock mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the object.[3] A secondary benefit of the High Priority scheme is that it also serves as a deadlock prevention mechanism. We hereafter refer to this scheme as **2PL-HP**.

In classical optimistic concurrency control (OPT) [Kung81], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. A transaction is restarted at its commit point if it fails its validation test. This test checks that there is no conflict of the validating transaction with transactions that committed since it began execution. For the RTDBS scenario, we use the *Broadcast Commit* variant of this algorithm [Mena82, Robi82]. Here, when a transaction commits, it notifies other currently running transactions which conflict with it and these conflicting transactions are immediately restarted. Note that there is no need to check for conflicts with already committed transactions since any such transaction would have, in the event of a conflict, restarted the validating transaction at its (the committed transaction's) own earlier commit time. This also means that a validating transaction is always guaranteed to commit. The broadcast commit method detects conflicts earlier than the basic OPT algorithm, resulting in both less wasted resources and earlier restarts; this increases the chances of meeting transaction deadlines. We hereafter refer to this scheme as **OPT-BC**. A point to note is that no use of tran-

[3] In addition, a new reader can join a group of read-lockers only if its priority is higher than that of all waiting writers.

saction priorities is made here in resolving data contention. We will return to this issue later in the paper.

3. 2PL-HP VERSUS OPT-BC

Locking and optimistic concurrency control represent the two extremes in terms of data conflict detection and conflict resolution – locking detects conflicts as soon as they occur and resolves them using blocking; optimistic concurrency control detects conflicts only at transaction commit times and resolves them using restarts. Earlier comparative studies of locking and optimistic algorithms for a conventional DBMS (e.g. [Agra87, Care88]) have shown that, under operating circumstances of limited resources, locking provides significantly better performance than optimistic concurrency control. Some fundamental aspects of the RTDBS world, outlined below, indicate a potential for these previous results to be altered in this unique environment.

3.1. Blocking

The main reason for the good performance of locking in a conventional DBMS is that its blocking-based conflict resolution policy results in conservation of resources, while the optimistic algorithm with its restart-based conflict resolution policy wastes more resources. In an RTDBS environment, however, we expect to see a smaller difference between the useful resource utilizations of the two algorithms, thus reducing the advantage that locking has over optimistic algorithms. This is because OPT-BC implicitly derives a blocking effect *due to resource contention* [Care89] -- low priority transactions wait when resources are captured by high priority transactions. Low priority transactions that may conflict with high priority transactions are effectively prevented from making progress by the priority-based resource scheduling, thus decreasing the chances of data conflicts; if a conflict does occur and the low priority transaction has to be restarted, the amount of wasted resource utilization is at least reduced. Conversely, 2PL-HP loses some of the basic 2PL algorithm's blocking factor due to the partially restart-based nature of the High Priority scheme.

3.2. Restarts

In locking algorithms, data conflicts are resolved as soon as they occur, while in optimistic algorithms, data conflicts are resolved only when a transaction attempts to commit. In a conventional DBMS, this delayed conflict resolution results in optimistic algorithms wasting more resources than locking algorithms. Although this is still true in the RTDBS scenario, the delayed conflict resolution of optimistic algorithms aids in making *better decisions*,

since more information about the conflicting transactions is available at this later stage. For example, in 2PL-HP, a transaction could be restarted by, or wait for, another transaction which is later discarded. Such restarts or waits are useless and cause performance degradation. In OPT-BC, however, we are *guaranteed* the commit, and hence completion, of any transaction which reaches the validation stage. Since only validating transactions can cause restarts of other transactions, *all* restarts generated by the OPT-BC algorithm are useful.

3.3. Priority Reversals

In an RTDBS, for certain types of transaction priority assignment schemes (e.g. Least Slack [Abbo88]), it is possible for a pair of concurrently running transactions to have opposite priorities relative to each other at different points in time during their execution. We will refer to this phenomenon as "priority reversal".[4] In 2PL-HP, data conflicts between members of such a pair could result in mutual restarts[5], as shown in Figure 1. There we show the priority profile as a function of time for two concurrently executing transactions A and B, with deadlines D_A and D_B, respectively. From the profiles it is evident that, although A initially has higher priority than B, with the passage of time, the situation gets reversed because B has a higher rate of priority increase. At time $t = X_1$, transaction B locks object X. At time $t = X_2$, transaction A attempts to access the same object in a conflicting mode. Since, at this time, the priority of A is greater than that of B, B is restarted and A is granted the lock. B begins re-executing with a new

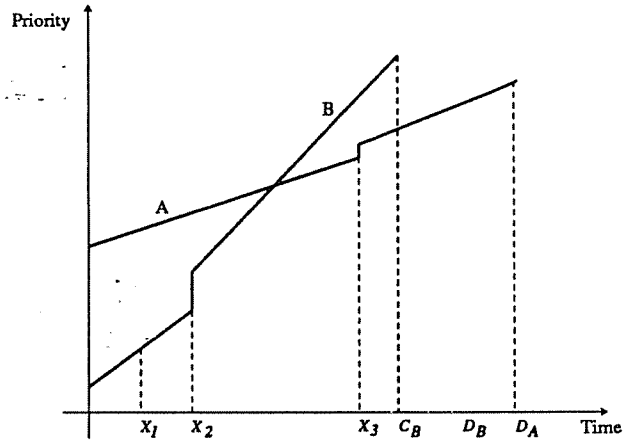

Figure 1: Mutual Restarts with 2PL-HP.

[4] This is different from *priority inversion* [Sha87], which refers to the situation where a transaction is blocked (due to data or resource conflict) by another transaction with a lower priority.

[5] The concept of mutual restarts between a pair of transactions can be extended to cyclic restarts among a set of transactions.

priority profile and attempts to access object X at time $t = X_3$. At this time, A is still holding the lock on object X. However, as B now has a higher priority than A, it is A's turn to be restarted while B is given the lock. Finally, B completes at time $t = C_B$ before its deadline D_B, while A misses its deadline D_A and is discarded. Given this final outcome, we observe that the restart of B at time $t = X_2$ was beneficial to neither transaction A nor transaction B.

Based on the above example, we can envision scenarios where, because of dynamically shifting transaction priorities, mutual restarts take place leading to wasted resources and an increased number of late transactions. Also, depending on the dynamics of the priority profile, we may have to "constantly" poll all locked data objects to check that lock holders still maintain higher priority over their associated lock waiters.[6] If a priority reversals is found, the regular High Priority scheme is applied between the waiters and the lock holders. Since OPT-BC does not make use of priorities in resolving data conflict, such priority related problems simply do not arise.

We have conducted experiments to evaluate the performance effects of the above-mentioned factors, and the following sections describe our experimental framework and the results that were obtained.

4. MODELING A REAL-TIME DBMS

We developed a detailed model of an RTDBS for studying the performance of the aforementioned concurrency control algorithms. In our model, the system consists of a shared-memory multiprocessor operating on disk resident data.[7] The database itself is modeled as a collection of pages. Transactions arrive in a Poisson stream and each transaction has an associated deadline time. A transaction consists of a sequence of read and write accesses. A read access involves a concurrency control request to get access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Write requests are handled similarly except for their disk I/O – their disk activity is deferred until the transaction has committed. The organization of the model is loosely based on the design in [Care88] and is shown in Figure 2.

The model has five components: a *source* that generates transactions; a *transaction manager* that models the execution of transactions; a *concurrency control (CC) manager* that implements the details of the concurrency control algorithms; a *resource manager* that models the CPU and I/O resources; and a *sink* that gathers statistics on

[6] This is required not only to ensure that higher priority transactions are not held up by lower priority transactions, but also for deadlock prevention – the High Priority scheme does not work as a deadlock prevention mechanism if priority reversals are possible.

[7] We assume that all data is accessed from disk and ignore buffer pool considerations.

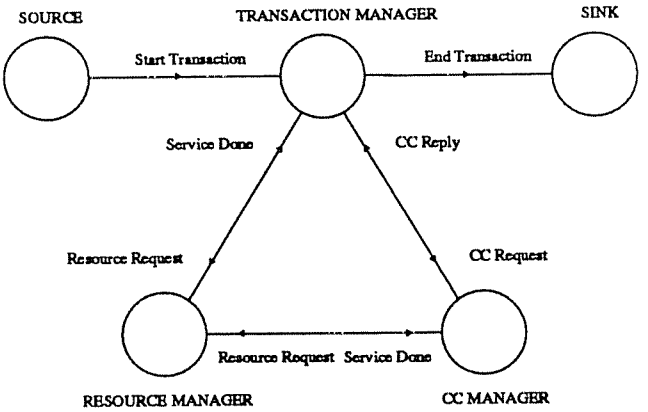

Figure 2: RTDBS Model Structure

completed transactions. The workload model characterizes transactions in terms of the pages that they access and the number of pages that they update. Table 1 summarizes the key parameters of the workload model. The *ArrivalRate* parameter specifies the rate of transaction arrivals. The *DatabaseSize* parameter fixes the number of pages in the database. The actual number of pages accessed by a transaction varies uniformly between half and one-and-a-half times the value of the *PageCount* parameter. The page requests themselves are generated from a uniform distribution spanning the entire database. The *WriteProb* parameter gives the probability that a page which is read will also be updated.

In our experiments, we used the following formula for deadline assignment :

$$D_T = A_T + SF * R_T$$

where

D_T = Deadline time of transaction T
A_T = Arrival time of transaction T
R_T = Resource time of transaction T
SF = Slack factor

The *slack factor* is a multiplicative constant greater than one, and *resource time* is the total service time at the resources that the transaction requires for its data processing. The formula is designed to ensure that all transactions, independent of their service requirement, have the same chance of making their deadline. By changing the value of the slack factor we can smoothly vary the tightness/slackness of deadlines. The *SlackFactor* parameter sets this value. The transaction priority assignment scheme used for the majority of the study was *Earliest Deadline* – transactions with earlier deadlines have higher priority than transactions with later deadlines. For one experiment, which was designed to investigate the impact of priority reversals, we used the *Least Slack* priority assignment

scheme – transactions with smaller slack times have higher priority than transactions with larger slack times. There the slack time of a transaction T at any time t was computed as $(D_T - t)$. The *PriorityPolicy* parameter fixes the transaction priority scheme. Finally, the *LatePolicy* parameter determines whether the system operates under soft or firm deadlines.

The physical resources in our model consist of multiple CPUs and multiple disks. There is a single queue for the CPUs and the service discipline is preemptive-resume, with the preemption being based on transaction priorities. Each of the disks has its own queue and is scheduled according to the priority-based variant of the elevator disk scheduling algorithm described in [Care89]. Table 2 summarizes the key parameters of the resource model. Requests at each disk are grouped into *NumDiskPrio* priority levels and the elevator algorithm is applied within each priority level.[8] The requests at a priority level are served only when there are no pending requests at higher priority levels. The priority levels are periodically reconfigured to reflect the change in transaction priorities as a function of time. The data itself is modeled as being uniformly distributed across all the disks and across all tracks within a disk, with the number of tracks on each disk being set by the *NumTracks* parameter. The *PageCpu*, *Disk-Delay* and *SeekFactor* parameters capture CPU and disk processing times per data page, and are detailed in [Care89].

Parameter	Meaning
ArrivalRate	Poisson rate of transaction arrivals
DatabaseSize	Number of pages in database
PageCount	Average pages accessed/transaction
WriteProb	Write probability/ accessed page
SlackFactor	Slack factor in deadline formula
PriorityPolicy	Transaction priority policy
LatePolicy	Firm or Soft deadlines

Table 1 : Workload Model Parameters

[8] The assignment of requests to disk priority levels, for a transaction T, uses the following mapping :

$$DiskLevel_T = NumDiskPrio * \left\lfloor 1 - \frac{Prio_T}{MaxPrio} \right\rfloor$$

where $Prio_T$ is transaction T's current priority computed according to the transaction priority assignment scheme, and *MaxPrio* is a normalizing constant set equal to the highest possible transaction priority.

Parameter	Meaning
NumCPUs	Number of processors
NumDisks	Number of disks
NumDiskPrio	No. of priority levels at each disk
NumTracks	Number of tracks per disk
PageCpu	CPU time for processing data page
DiskDelay	Disk rotational + transfer delays
SeekFactor	Factor relating seek time to distance

Table 2 : Resource Model Parameters

5. EXPERIMENTS AND RESULTS

In this section, we present performance results for our experiments comparing 2PL-HP and OPT-BC in a real-time database system environment.[9] The simulator used to obtain the results was written in the Modula-2-based DeNet simulation language [Livn88]. We first describe the performance metrics and then list the baseline values for the system parameters. Subsequently, we discuss our results with regard to the impact of data contention, resource contention, deadline tightness/slackness, priority reversals, knowledge of transaction resource requirements, and various deadline policies. All of our experiments evaluate the system over a wide range of loads. To serve as a basis for comparison, the performance levels achieveable in the *absence* of any concurrency conflicts are also shown on the graphs, under the title NO-CC. The NO-CC curve should be interpreted as the contribution of resource contention alone towards performance degradation. In the Appendix, we provide a theoretical basis for the shapes of the curves seen in our simulation results.

5.1. Performance Metrics

The primary performance metric used is *MissPercent*, which is the percentage of transactions that do not complete before their deadline. MissPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loadings, while MissPercent values in the range of 20 to 100 percent represent system performance under "heavy" loading.[10] A secondary performance metric, *MeanLateness*, used only when we consider soft deadlines, measures the average time by which transactions miss their deadlines. The simulations also generated a host of other statistical information, including CPU and disk utilizations, number of transaction restarts,

[9]All graphs in this paper show mean values with relative half-widths about the mean of less than 10% at 90% confidence interval, with each experiment having been run until at least 5000 transactions had been processed by the system. Only statistically significant differences are discussed here.

[10]Any long-term operating region where the miss percent is large is obviously unrealistic for a viable RTDBS. Exercising the system to high miss levels, however, provides valuable information on the response of the algorithms to brief periods of stress loading.

average transaction blocked time, etc. These secondary measures help explain the behavior of the algorithms under various loading conditions.

5.2. Parameter Settings

The resource parameter settings were such that the CPU time to process a page was 10 milliseconds while disk access times were between 15 and 30 milliseconds, depending on the level of disk utilization. These settings made the CPU utilization and the disk utilization almost balanced, and in particular, the system was slightly disk-bound. We started our experiments by first developing a baseline model around which we then constructed further experiments by varying a few parameters at a time. The settings of the workload parameters and resource parameters for our baseline model are listed in Tables 3 and 4. These settings were chosen with the objective of having a high amount of data and resource contention in the system, thus helping to bring out the differences between the algorithms. The baseline model represents an RTDBS system that has firm deadlines, and it therefore discards late transactions. Also, the system lacks any knowledge of transaction resource requirements. This implies that a transaction is detected as being late only when it actually misses its deadline.

Parameter	Value
DatabaseSize	1000 pages
PageCount	16 pages
WriteProb	0.25
SlackFactor	4.0
PriorityPolicy	Earliest Deadline
LatePolicy	Firm Deadlines

Table 3 : Baseline Model Workload Settings

Parameter	Value
NumCPUs	10
NumDisks	20
NumDiskPrio	5
NumTracks	1000
PageCpu	10 ms
DiskDelay	15 ms
SeekFactor	0.5 ms

Table 4 : Baseline Model Resource Settings

5.2.1. Baseline Model

For the baseline model, Figures 3 and 4 show MissPercent behavior under normal load and heavy load conditions, respectively. From these graphs, it is clear that for very low arrival rates, 2PL-HP and OPT-BC perform almost identically, but as the arrival rate increases, OPT-BC does progressively better than 2PL-HP. The cause for the better performance of OPT-BC is its lower number of restarts, as shown in Figure 5.[11] In OPT-BC, only a *committing* transaction can generate restarts. In 2PL-HP, however, a transaction can generate restarts at any time during the course of its execution. Therefore, even a transaction which is later discarded can cause restarts. At higher loads, when many transactions miss their deadline and have to be discarded, 2PL-HP has significantly more restarts than OPT-BC. This is brought out clearly in Figure 5, where we observe a large difference between the "useful restarts" curve for 2PL-HP, which shows the number of restarts caused only by eventually committed transactions, and the "total restarts" curve for 2PL-HP, which shows the total number of restarts caused by all transactions. The restarts decrease after a certain load because resource contention, rather than data contention, becomes the dominant reason for transactions getting discarded.

Figure 6 shows at what stage during their execution, on the average, transactions were restarted due to data conflicts. We observe that 2PL-HP consistently detects conflicts earlier than OPT-BC. We might therefore expect 2PL-HP to waste less resources than OPT-BC, but since OPT-BC has much fewer restarts, it actually makes better overall use of resources than 2PL-HP. This concept is quantified in Figure 7, where the total utilization and the useful utilization of resources are shown. Useful utilization is computed as the resource usage made by only those transactions which eventually met their deadlines. We see that OPT-BC makes better use of the resources than 2PL-HP. Note that, for a conventional DBMS, this workload scenario would result in exactly the *opposite* results – a locking algorithm would do better than an optimistic algorithm since the useless restarts problem would not exist and resources would be better conserved by the locking algorithm.

5.2.2. Data and Resource Contention

Under conditions of low data contention, we would expect performance to primarily be determined by the resource scheduling algorithm rather than the concurrency control algorithm. This is borne out in Figure 8, where

[11] All "restart" graphs in this paper are normalized on a per-transaction basis; that is, they are computed as the number of restarts divided by the number of processed transactions.

we observe that both 2PL-HP and OPT-BC perform almost indistinguishably when the number of data pages is increased ten-fold from the baseline value of 1000 to 10000. Also, their performance comes close to that of NO-CC.

To obtain a condition of high data contention and low resource contention, we approximately simulated an "infinite" resource situation [Agra87], that is, where there is no queueing for resources. This was done by increasing twenty-fold the number of processors and the number of disks, from their baseline values of 10 and 20 to 200 and 400, respectively.[12] The performance results for this scenario are shown in Figures 9 and 10, where we see that OPT-BC does much better than 2PL-HP in terms of both normal load performance and heavy load performance. There are two reasons for OPT-BC outperforming 2PL-HP: First, the basic useless restarts problem of 2PL-HP, as outlined before for the baseline model, occurs here too. Second, blocking in 2PL-HP reduces the number of transactions available to run and make progress, whereas in OPT-BC, transactions are always trying to move ahead. Blocking causes an increase in the number of transactions in the system, thus generating more conflicts and more restarts. Figures 11 and 12 show these effects quantitatively. It should be noted that optimistic algorithms perform better than locking under infinite resource conditions in a conventional DBMS setting too[Fran85, Agra87].

The baseline model is representative of the high data contention and high resource contention scenario, and as stated before, we see that OPT-BC does noticeably better than 2PL-HP under heavy-load conditions and slightly better under normal-load conditions. An interesting observation is that although OPT-BC does not make use of priorities, leaving the resource scheduler to handle all priority-related decisions, it still does better than 2PL-HP, which uses priorities to "help" transactions make their deadlines. Another important point to note here is that while resource contention can be reduced by purchasing more resources and/or faster resources, there exists no equally simple mechanism to reduce data contention. While abundant resources are usually not to be expected in conventional database systems, they may be more common in RTDBS environments since many real-time systems are sized to handle transient heavy loading. This directly relates to the application domain of RTDBSs, where functionality, rather than cost, is usually the driving consideration.

[12]These resource levels ensured that the total resource utilization did not go beyond 25 percent even at the highest load of 100 transactions/sec.

5.2.3. Deadline Tightness / Slackness

The next set of experiments examined the effect that deadline tightness/slackness had on the relative performance of the algorithms. To do this we varied the SlackFactor from 1 to 10 while keeping all the other parameters the same as those of the baseline model. We conducted the experiment for arrival rates of 10 and 30 transactions/sec and the corresponding graphs are shown in Figures 13 and 14. At low slack factors, both algorithms show a rapid increase in MissPercent since transactions are operating under very tight deadlines. OPT-BC is slightly more stable than 2PL-HP in terms of the steepness of the increase. As the slack factor increases, transactions are given more time to complete, and the MissPercent decreases sharply. For the arrival rate of 10 transactions/sec, the MissPercent goes down all the way to zero. For the arrival rate of 30 transactions/sec, however, we observe that the MissPercent stays virtually constant beyond a slack factor of 5, for both 2PL-HP and OPT-BC. This behavior is explained as follows: Since increasing the slack factor provides transactions with more time to complete, it results in more transactions concurrently running in the system. As the number in the system increases, the resources in the system eventually saturate and the MissPercent then becomes constant for a fixed arrival rate. Figure 15 shows that the lower number of restarts for OPT-BC is again the cause for its better performance when compared to 2PL-HP.

Based on the foregoing experiments, we conclude that *OPT-BC is preferred to 2PL-HP* for a RTDBS system that has firm deadlines and no apriori knowledge of transaction resource requirements. This is especially true under conditions of heavy loading, high data contention, or low resource contention. We have also studied the effects of changes in page write probabilities, transaction sizes and database sizes, although space constraints preclude their inclusion here. These experiments reinforced the general conclusions given above. An interesting observation was that when the database size was very large, making data conflicts infrequent, basic 2PL did better than both OPT-BC and 2PL-HP. This is because, in the absence of significant data contention, the concurrency control algorithm which best conserves resources provides the best performance.

5.2.4. Priority Reversals

In order to examine the performance effect of priority reversals, we conducted an experiment where *Least Slack* was used as the transaction priority assignment policy, while keeping the other parameters the same as those of the baseline model. The slack of a transaction was computed when it arrived, and this remained the transaction's

priority as long as it was executing; if the transaction was restarted, it's slack was then recomputed. This slack evaluation scheme is called *static evaluation* in [Abbo89].[13] Figure 16 shows the results of this experiment. In the corresponding experiment using the Earliest Deadline policy (see Figures 3 and 4), we had observed that 2PL-HP had performed comparably to OPT-BC at very low and very high loads, and noticeably worse at intermediate loads. Here we notice, however, that 2PL-HP does significantly worse than OPT-BC over virtually the entire range of loadings. The restart curves in Figure 17 show the reasons for the performance degradation of 2PL-HP: it now suffers from not only the useless restarts problem but also from the mutual restarts problem. The "NO-MUTUAL" curve in Figure 17 shows the total number of restarts discounting those caused due to priority reversals.[14] As we can see, mutual restarts make a perceptible contribution to the total number of restarts. A point to note is that the static evaluation scheme produces limited fluctuation in transaction priorities since these priorities are recomputed only at transaction restart times. For schemes which generate greater fluctuations in priority (e.g. *continuous evaluation* [Abbo89]), the mutual restarts problem could be expected to have a greater impact on the performance of 2PL-HP.

5.3. Knowledge of Transaction Resource Requirements

Some real-time systems, such as manufacturing plants, are characterized by having a few well-defined actions which are done repetitively. In such systems, it may be possible to have a good knowledge of transaction behavior, thus enabling reasonably accurate estimates of transaction resource requirements. We conducted experiments to evaluate the impact of having such knowledge. In particular, the estimates were used to aid in early detection of transactions that were destined to become late. A transaction was discarded whenever it was realized that its remaining service requirement was larger than the time remaining to its deadline. This is because even if the transaction were to run alone in the system, it is guaranteed not to complete before its deadline. This policy for detecting late transactions is called *Feasible Deadlines* in [Abbo88].

We conducted an experiment to evaluate the impact of the Feasible Deadlines policy, while keeping all the parameters the same as those of the baseline model. Figure 18 shows the result of this experiment. We observe that

[13] The 2PL-HP algorithm is altered here in the following manner: The priority of the requesting transaction is compared not with the current priority of the lock holders, but with the priority the lock holders would have *if they were to be restarted.* The reason for this change is to prevent immediate mutual restarts – a detailed explanation is given in [Abbo89].

[14] Here we disregard cyclic restarts, and take into account only mutual restarts. The mutual restart counter was incremented whenever a transaction A was restarted by another transaction B, with A itself having restarted B at an earlier time.

the performance of both 2PL-HP and OPT-BC is much improved when compared to Figures 3 and 4, and that the performance difference between them has shrunk greatly. The restart curves shown in Figure 19 and the utilization curves shown in Figure 20, when compared to the corresponding curves in Figure 5 and Figure 7, highlight the cause for the performance improvements. Both OPT-BC and 2PL-HP benefit from the savings on wasted resource utilization due to the early detection of late transactions. In addition, 2PL-HP benefits by the elimination of the useless restarts which could otherwise have been caused by these "soon-to-be-late" transactions. Figures 20 and 21 show the MissPercent and restart graphs for the infinite resources scenario. Here, since resource utilization is not an issue, only 2PL-HP improves its performance due to the elimination of many useless restarts.

5.4. Soft Deadline Policy

All of the previous experiments assumed a firm deadline policy; that is, late transactions were immediately discarded from the system. In this section, we look into the impact of having a soft deadline policy, where all transactions have to be run to completion. While the single metric of MissPercent was sufficient to characterize the firm deadline policy, here we need an auxiliary metric – MeanLateness, which captures the tardy behavior of late transactions. A transaction that commits within its deadline has a lateness of zero. A transaction that completes after its deadline has a lateness of $(C_T - D_T)$, where C_T and D_T are the transaction's completion time and deadline time, respectively. The presence of two metrics complicates matters since we now have to decide upon the relative importance of the metrics. For example, is it worthwhile to trade a ten percent increase in Mean Lateness for a five percent improvement in MissPercent? Also, since MissPercent and MeanLateness are adversarial metrics, in the sense that a decrease in one will usually result in an increase in the other, it is difficult to simultaneously improve both metrics. While the tradeoff to be established between the two metrics is ultimately completely application-dependent, we describe below one possible policy, which we will subsequently refer to as *LateHigh*.

In the LateHigh policy, late transactions are given higher priority than feasible transactions so that, although they complete late, they complete with minimum delay. Of course, this preferential treatment for late transactions may cause some feasible transactions to miss their deadline which essentially means that we are willing to trade MissPercent for any improvement in MeanLateness. The results in [Abbo89] assume such a system.

We conducted an experiment to investigate the impact of a soft deadline policy by using the LateHigh mechanism to deal with late transactions, while keeping all the other parameters the same as those of the original

baseline model. An important point to note here is that 2PL-HP no longer suffers from the useless restarts problem, as all transactions are run to completion. Figures 23 and 24 show the MissPercent and MeanLateness results for this experiment. We see that OPT-BC saturates slightly earlier than 2PL-HP and has worse MeanLateness performance. Figures 25 and 26 show the same graphs under conditions of infinite resources. Here we see the opposite results – OPT-BC saturates much later than 2PL-HP and has correspondingly better MeanLateness performance.

Based on these experiments, we conclude that 2PL-HP is better than OPT-BC for a soft deadline system with finite resources (assuming LateHigh policy), but for systems with plentiful resources, OPT-BC is much better. Therefore, the behavior here is similar to that seen in conventional database systems.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a quantitative study of the relative performance of locking and optimistic concurrency control techniques in the context of a real-time database system (RTDBS). The performance metric used here is the percentage of deadlines made, unlike a conventional DBMS where response time or throughput is the performance criterion. In a conventional DBMS, optimistic algorithms generally perform worse than locking. In an RTDBS, however, optimistic algorithms show improved performance because they derive a blocking effect from the priority-based handling of resource contention. Also, their delayed data conflict resolution policy aids them in making better decisions. To evaluate the effect of these factors, detailed experiments were carried out on a simulated RTDBS with two representative algorithms: two-phase locking with high priority conflict resolution (2PL-HP) and broadcast optimistic (OPT-BC).

We showed that the policy for dealing with late transactions, knowledge of transaction resource requirements, and the availability of resources all have a significant impact on the relative behavior of the algorithms. In particular, for a system where late transactions are discarded, we demonstrated that OPT-BC outperforms 2PL-HP over a wide range of system loading and resource availability. We also showed that 2PL-HP was more sensitive than OPT-BC to the dynamics of transaction priority profiles. When the system had advance knowledge of transaction resource requirements, both OPT-BC and 2PL-HP performed much better, with their performance difference shrinking significantly. Under a policy where late transactions are run to completion, the picture was not as clear-cut; with certain caveats, we showed that when late transactions are given higher priority than feasible transactions, 2PL-HP does better than OPT-BC under finite resources, and OPT-BC does better than 2PL-HP when resources are plentiful.

In conclusion, from a performance standpoint, we can say that optimistic schemes appear generally better suited than locking to the RTDBS environment.[15]

OPT-BC does not make use of transaction priorities in resolving data conflicts. While this protects it from problems related to priority dynamics, it also prevents it from making smarter decisions which could help in decreasing the number of missed deadlines. We are currently working on developing an optimistic algorithm which allows for the use of priorities to improve decision making but which is yet safeguarded from the problems arising out of priority dynamics. We also intend to look into the issues involved in the performance of concurrency algorithms in a distributed RTDBS environment.

Acknowledgments

The authors thank Rajiv Jauhari for helpful discussions and assistance in developing the RTDBS simulator.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions : a Performance Evaluation," *Proc. of the 14th Conference on Very Large Database Systems*, Aug. 1988.
[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data," *Proc. of the 15th Conference on Very Large Database Systems* , Aug. 1989.
[Agra87] Agrawal, R., Carey, M., and Livny,M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Systems* , Dec. 1987.
[Care88] Carey, M., and Livny, M., "Distributed Concurrency Control Performance : A Study of Algorithms, Distribution, and Replication," *Proc. of the 14th Conference on Very Large Database Systems*, Aug. 1988.
[Care89] Carey, M., Jauhari, R., and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. of the 15th Conference on Very Large Database Systems*, Aug. 1989.
[Eswa76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM* , Nov. 1976.
[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems* , June 1981.
[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1988.
[Mena82] Menasce, D., and Nakanishi, T., "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," *Information Systems* , vol. 7-1, 1982.
[Robi82] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems," *Ph.D. Thesis* , Carnegie Mellon University, 1982.
[Sha87] Sha, L., Rajkumar, R., and Lehoczky, J., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *Tech. Report* No. CMU-CS-87-181, Carnegie Mellon University, Dec. 1987.
[Jack63] Jackson, J.R., "Jobshop-like Queuing Systems," *Management Science* , No. 10-1, Oct. 1963.
[Klei75] Kleinrock, L., "Queueing Systems," Vol. I, John Wiley & Sons, New York.

[15] Open problems do remain in order to make optimistic schemes truly practical, particularly in the areas of recovery methods and index management. We hope that our results will encourage research in this direction.

APPENDIX

In this section we try to provide a theoretical basis for the observed shapes of the performance curves discussed in Section 5. Using the terminology of queueing networks, we can, in a very loose and abstract fashion, compare a firm deadline system to a M/M/1/K system, while the soft deadline system can be compared to a M/M/1 system. The M/M/1/K queueing model characterizes a system with Poisson customer arrivals, exponential customer service times, a single server, and a maximum of K customers in the system. A new customer that arrives when there are already K customers in the system is thrown away. If we take the percentage of customers thrown away to be analogous to our MissPercent metric, and denote it by α, we then have the result (using Jackson's Theorem [Jack63], and assuming a mean customer service requirement of 1 time unit),

$$\alpha = 100 * \left[1 - \frac{\lambda^K - 1}{\lambda^{K+1} - 1}\right]$$

where λ is the customer arrival rate. A sample graph of α versus λ for K = 10 is shown in Figure 27 and, as we can see, the behavior is very similar to that seen in the performance graphs for a RTDBS with a firm deadline policy.

The formula for α can be split up in the following fashion :

For $\lambda \ll 1$,

$$\alpha \approx 100 * \lambda^K.$$

For $\lambda \gg 1$,

$$\alpha \approx 100 * \left[1 - \frac{1}{\lambda}\right].$$

These two formulas give good approximations for the basic shape of the curves seen at normal and high loadings, respectively.

The M/M/1 system is identical to the M/M/1/K system except that customers are never thrown away (i.e. K = ∞). For this system, if we take the percentage of customers that have a response time greater than some constant D, to be analogous to our MissPercent metric, and denote it by β, we then have the result (assuming a FCFS service discipline and a mean customer service requirement of 1 time unit),

$$\beta = 100 * e^{-(1-\lambda) * D}$$

where λ is the customer arrival rate. This result directly derives from the fact that the response time distribution for an M/M/1 system [Klei75] has an exponential distribution with parameter $(\mu - \lambda)$, where μ is the mean customer service requirement. A sample graph of β versus λ for D = 10 is shown in Figure 28 and, as we can see, the behavior is very similar to that seen in the performance graphs for a RTDBS with a soft deadline policy.

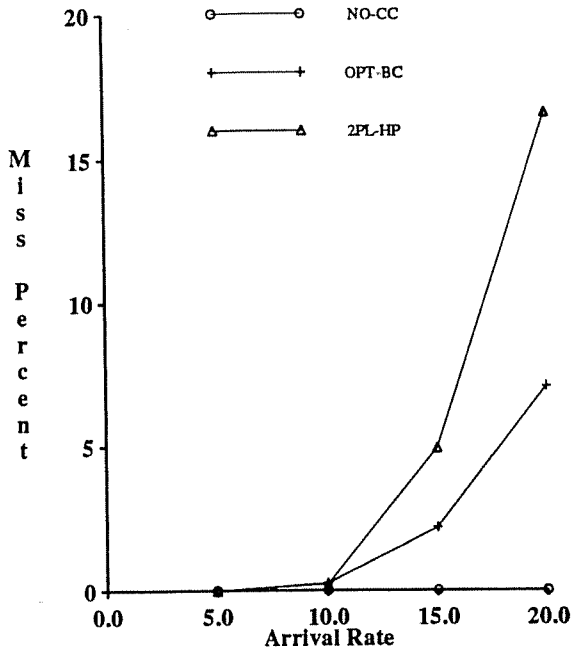

Figure 3: Baseline Model (Normal Load).

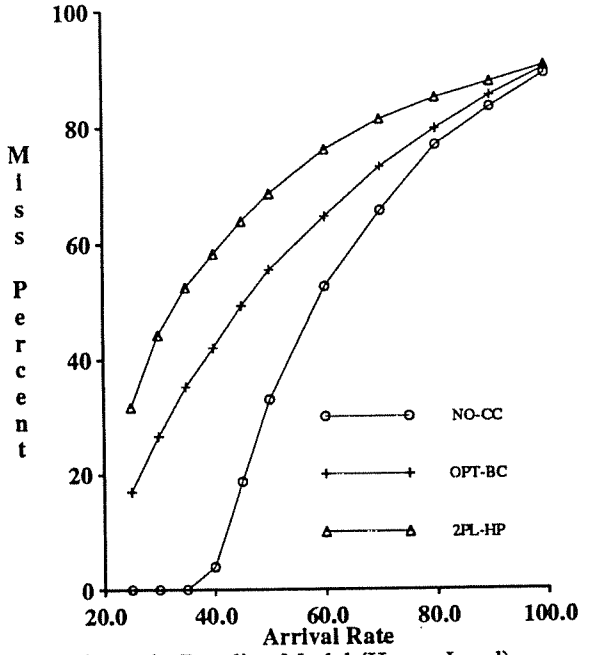

Figure 4: Baseline Model (Heavy Load)

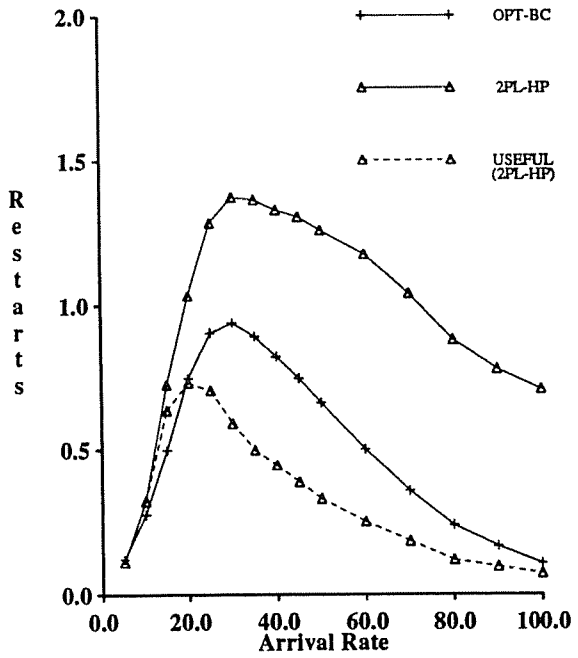

Figure 5: Restarts (Baseline Model).

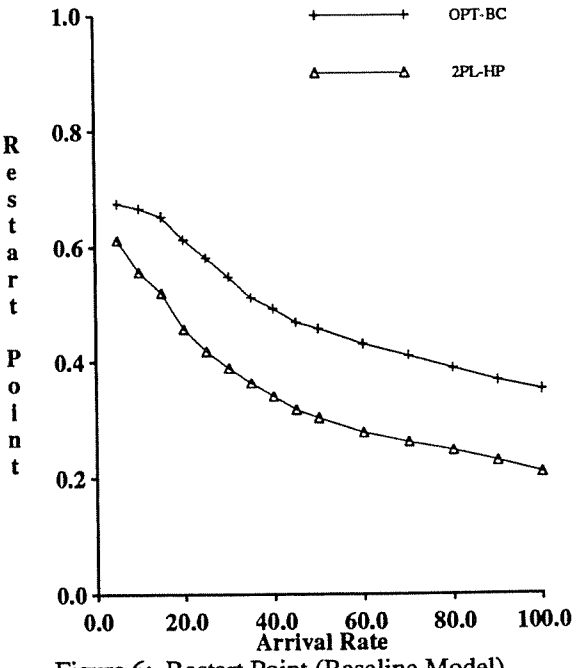

Figure 6: Restart Point (Baseline Model).

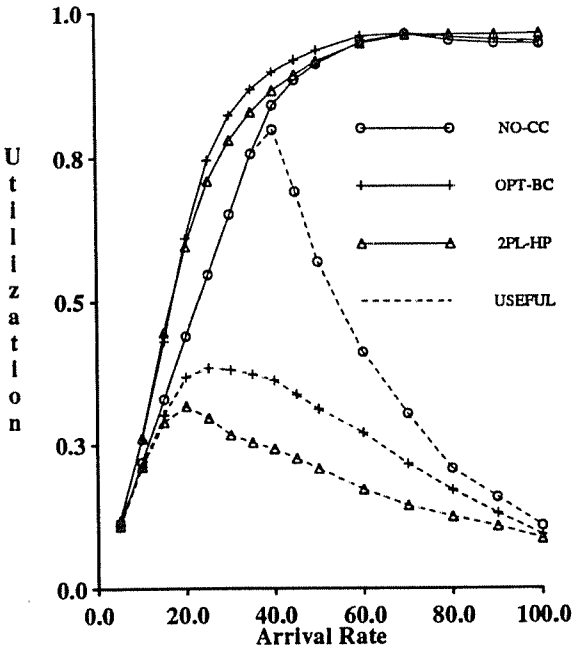

Figure 7: Utilization (Baseline Model).

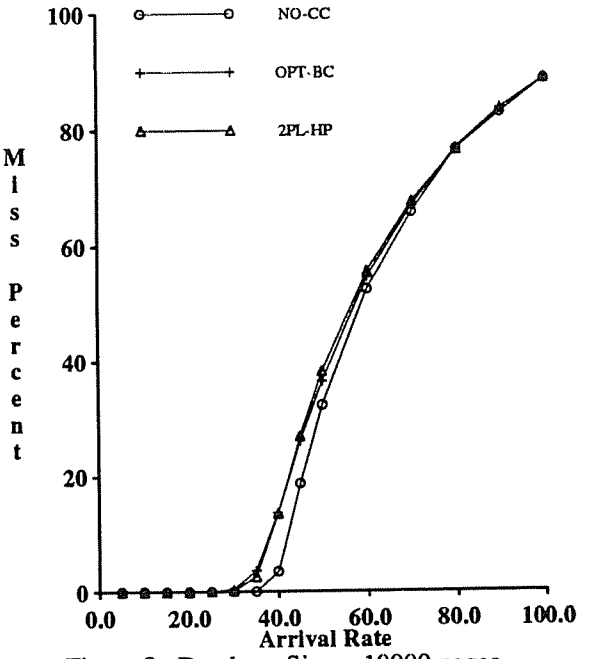

Figure 8: Database Size = 10000 pages.

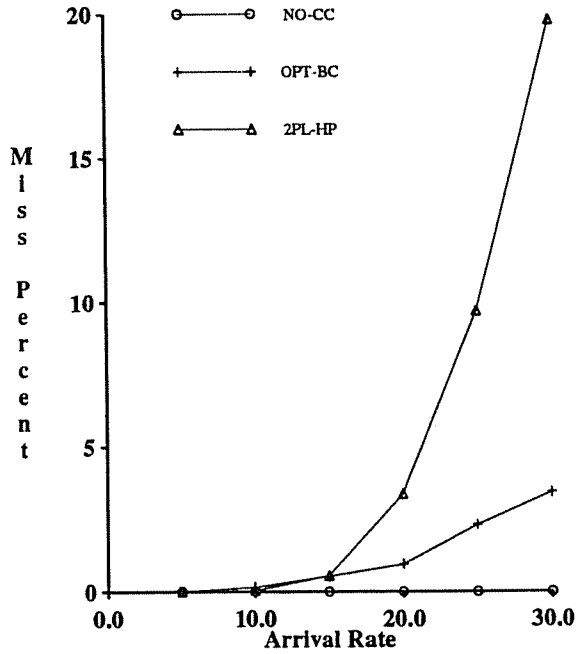

Figure 9: Infinite Resources (Normal Load).

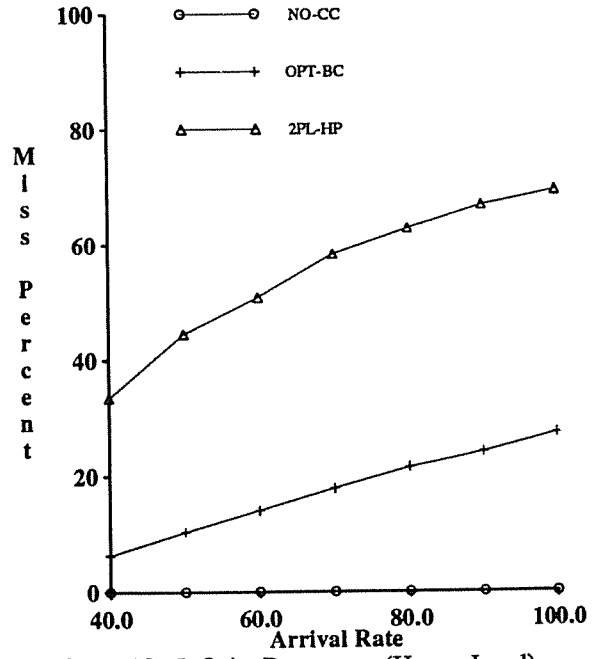

Figure 10: Infinite Resources (Heavy Load).

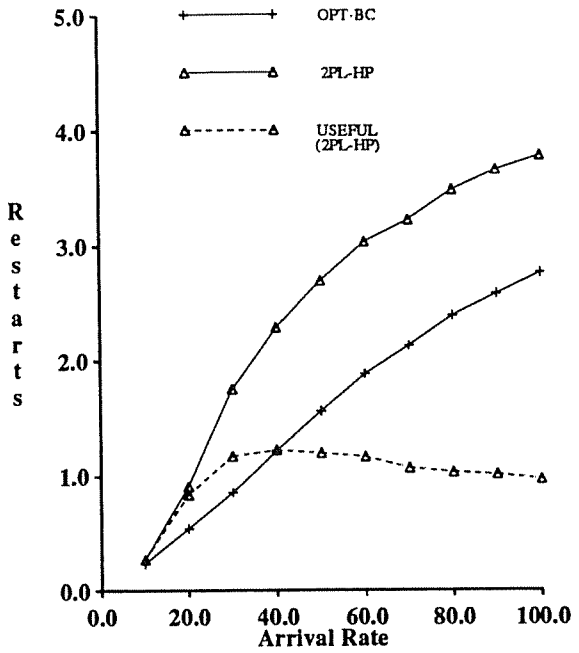

Figure 11: Restarts (Inf. Resources)

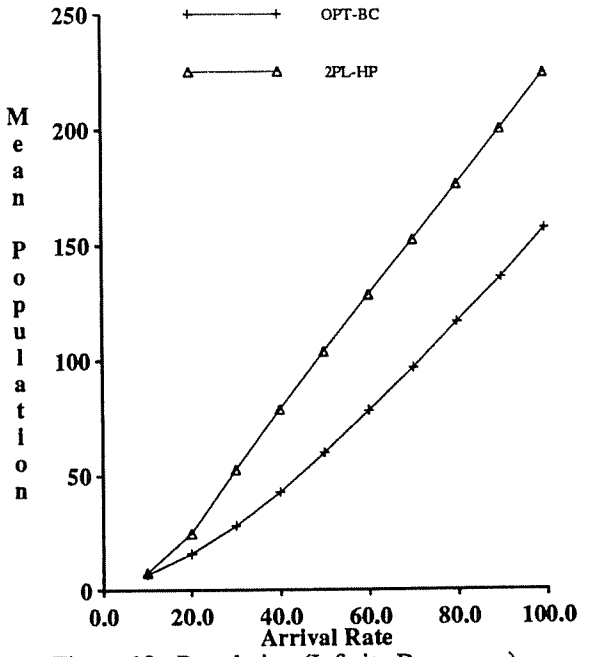

Figure 12: Population (Infinite Resources).

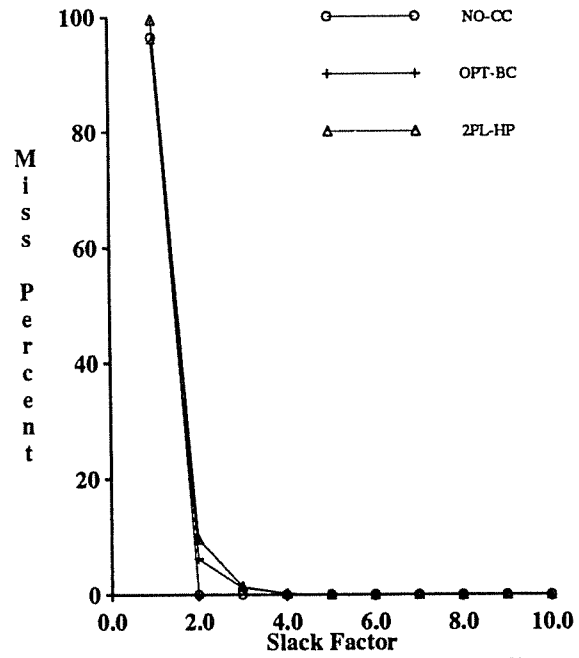

Figure 13: Slack Factor (Arrival Rate = 10).

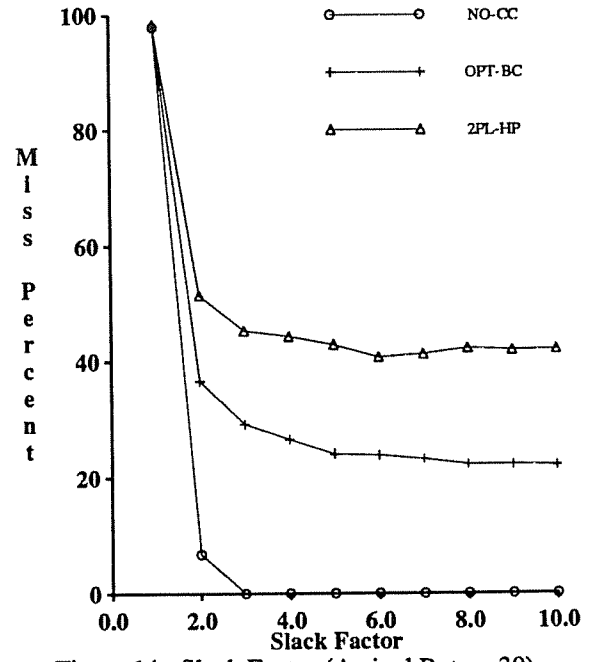

Figure 14: Slack Factor (Arrival Rate = 30).

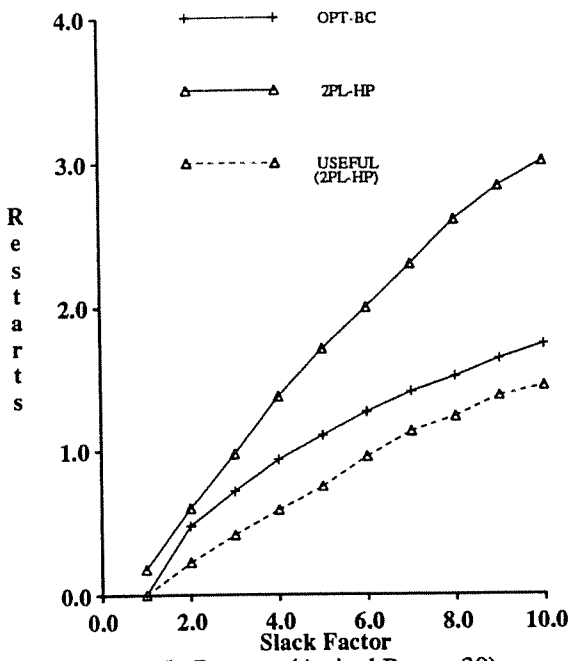

Figure 15: Restarts (Arrival Rate = 30).

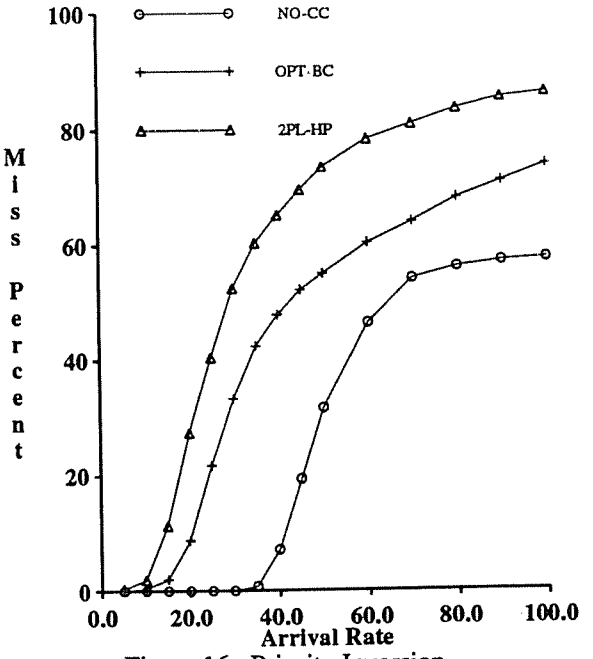

Figure 16: Priority Inversion.

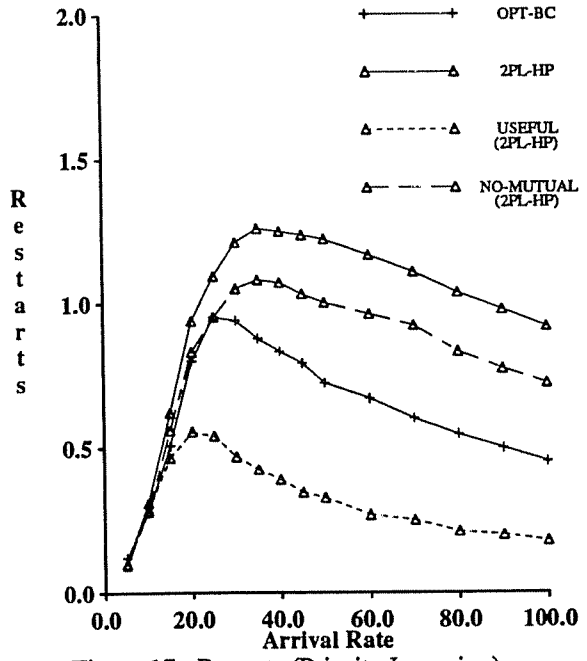

Figure 17: Restarts (Priority Inversion).

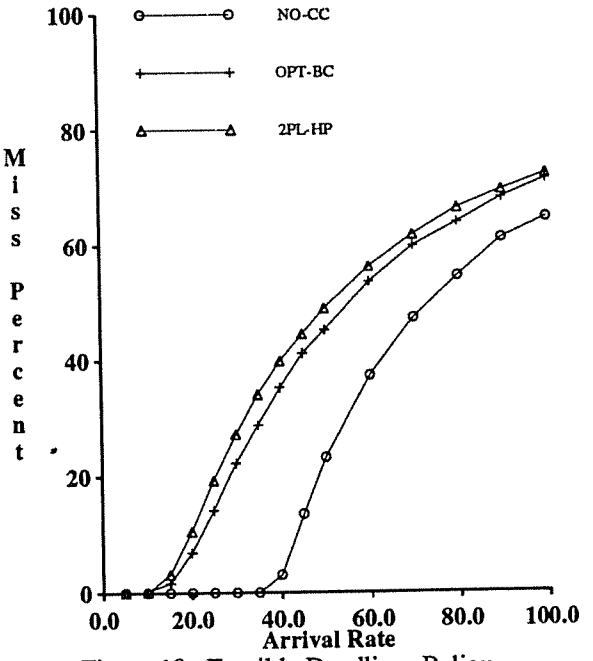

Figure 18: Feasible Deadlines Policy.

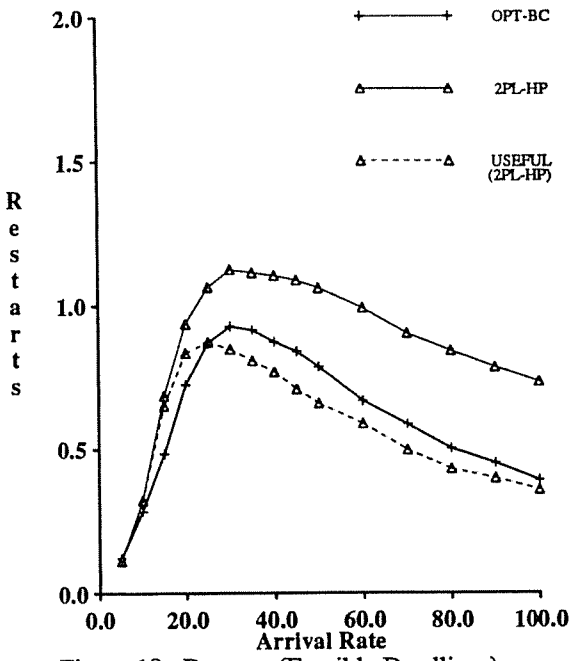

Figure 19: Restarts (Feasible Deadlines).

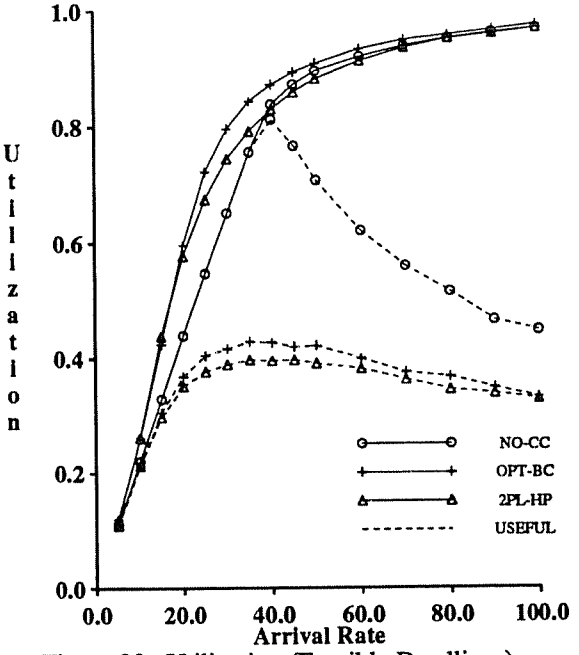

Figure 20: Utilization (Feasible Deadlines).

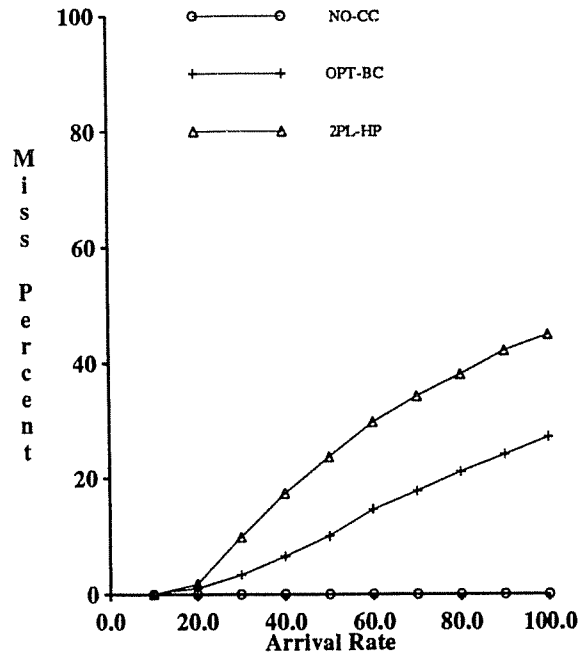

Figure 21: Infinite Resources (Feasible Deadlines).

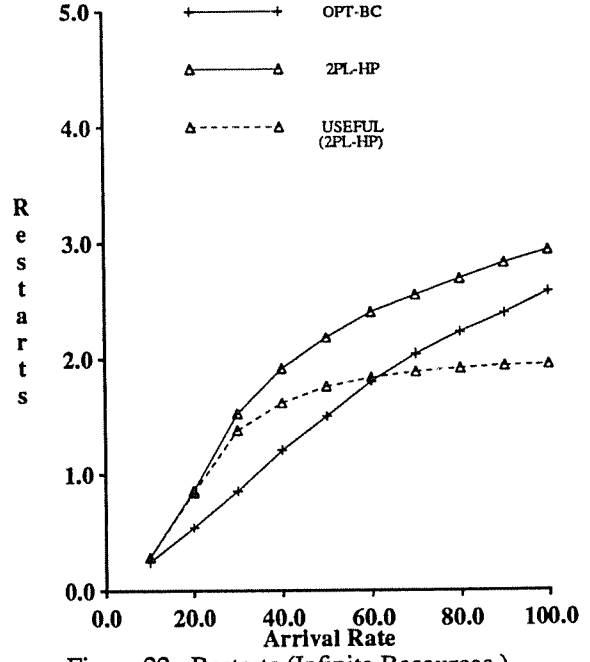

Figure 22: Restarts (Infinite Resources).

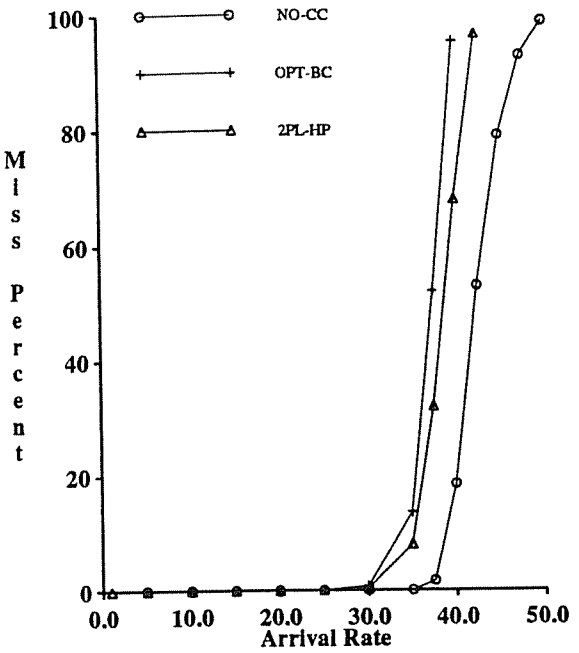

Figure 23: Soft Deadline Policy

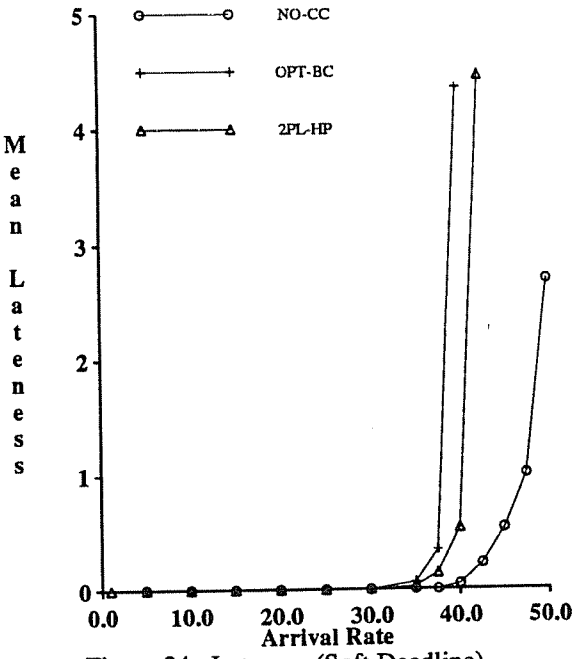

Figure 24: Lateness (Soft Deadline).

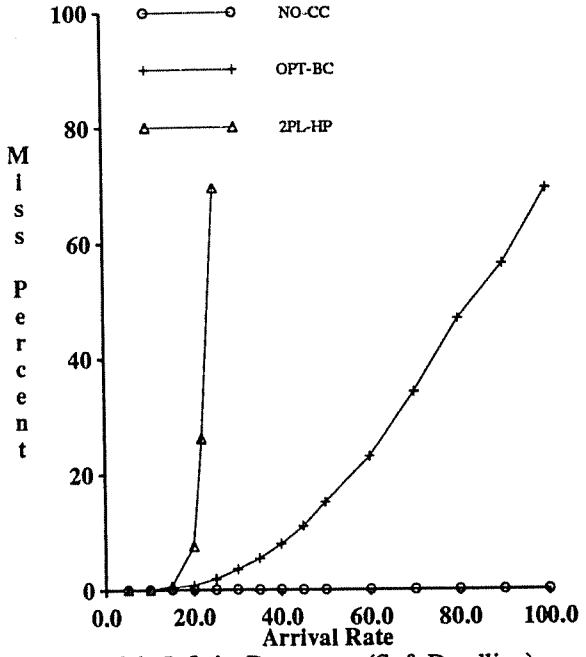

Figure 25: Infinite Resources (Soft Deadline).

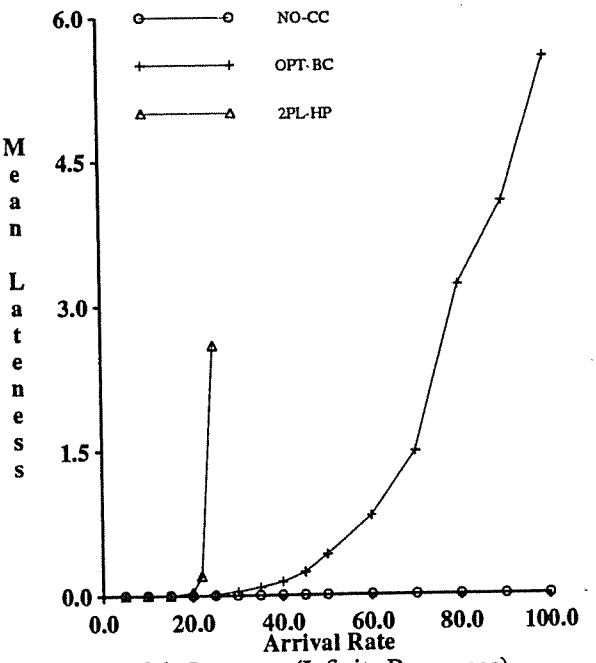

Figure 26: Lateness (Infinite Resources).

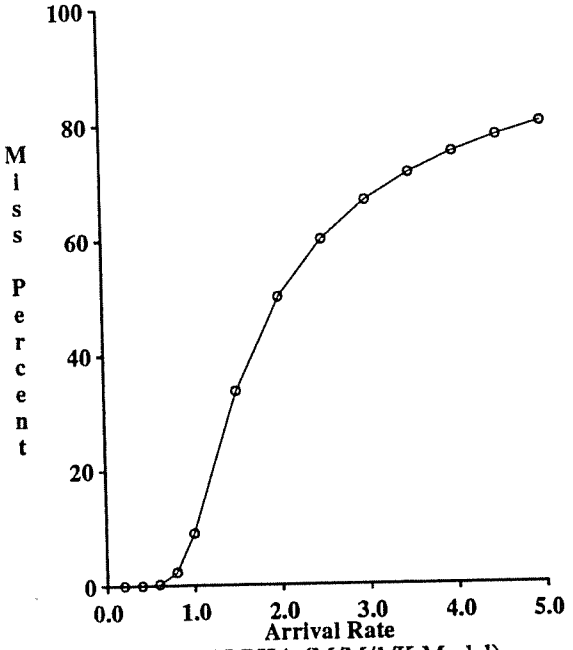

Figure 27: ALPHA (M/M/1/K Model)

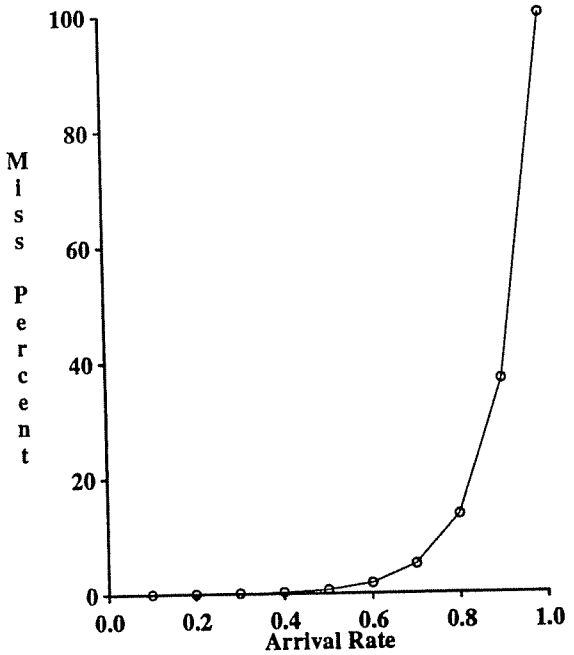

Figure 28: BETA (M/M/1 Model).