Data Access Scheduling in Firm Real-Time Database Systems

Jayant R. Haritsa[†]

Systems Research Center

University of Maryland

College Park, MD 20742

Michael J. Carey

Miron Livny

Computer Sciences Department

University of Wisconsin

Madison, WI 53706

ABSTRACT

A major challenge addressed by conventional database systems has been to efficiently implement the transaction model, which provides the properties of atomicity, serializability, and permanence. Real-time applications have added a complex new dimension to this challenge by placing deadlines on the response time of the database system. In this paper, we examine the problem of real-time data access scheduling, that is, the problem of scheduling the data accesses of real-time transactions in order to meet their deadlines. In particular, we focus on "firm deadline" real-time database applications, where transactions that miss their deadlines are discarded and the objective of the real-time database system is to minimize the number of missed deadlines. Within this framework, we use a detailed simulation model to compare the performance of several real-time locking protocols and optimistic concurrency control algorithms under a variety of real-time transaction workloads. The results of our study show that in moving from the conventional database system domain to the real-time domain, there are new performance-related forces that come into effect. Our experiments demonstrate that these factors can cause performance recommendations that were valid in a conventional database setting to be significantly altered in the corresponding real-time setting.

[†] This work was done while the author was with the Computer Sciences Department, University of Wisconsin - Madison.

This research was partially supported by the National Science Foundation under grant IRI-8657323.

1. INTRODUCTION

A real-time database system (RTDBS) is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of the system is to meet these deadlines, that is, to complete processing transactions before their deadlines expire. Our interest in real-time database systems stems from the increasing number of data-intensive applications that are faced with timing requirements. These applications include aircraft control, stock trading, network management and factory automation [Abbo88, Stan88]. The real-time performance of an RTDBS depends on several factors such as the database system architecture and the underlying processor and disk speeds. For a given system configuration, however, the primary real-time performance determinants are the policies used for scheduling transactions at the system resources. These policies determine *when* service is provided to a transaction, and therefore directly impact whether or not a transaction completes before its deadline.

The resources that are typically scheduled in real-time computer systems are processors, memory, and disks; these hardware resources are cooperatively scheduled to meet the system's real-time performance objectives. A special feature of real-time *database* systems is that, in addition to the standard physical resources, the data objects stored in the database form a set of logical resources. Therefore, the data accesses of transactions also have to be scheduled in accordance with real-time performance objectives. In this paper, we investigate the problem of real-time data access scheduling, that is, how to schedule the data accesses of real-time transactions in order to meet their deadlines.

Before considering the real-time aspects of data access scheduling, it is important to note several factors that make data a unique resource and therefore require its scheduling to be considered separately from hardware resource scheduling. First, there is a notion of "correctness" associated with the order in which transactions access data objects; this is unlike hardware resources, where the order of service generally does not affect the validity of the received service. The standard notion of correctness in database systems is *serializability* [Eswa76], which requires the outcome of concurrent data accesses by transactions, as reflected in the database, to be the same as that produced by executing the concurrent set of transactions in some serial order. Due to the serializability requirement, a transaction that is scheduled to access a data item does not necessarily make progress towards its completion. This is because the access may later be determined to have violated the serializability requirement, in which case the transaction may have to start all over again. Second, a transaction that is preempted from utilizing a data object must usually be restarted. In the event of a restart, the transaction loses the service that it has already received from the system resources. A transaction that is preempted from service at the hardware resources, however, merely suffers a delay in its execution. Third, data is a logical resource, and the number of transactions that may be concurrently scheduled to utilize a

particular data object is a function of the scheduling policy, rather than of the data object itself. Hardware resources such as CPUs and disks, however, are physically constrained to serve only a single request at a time. Finally, data objects have "identity" in the sense that one data object cannot be substituted for another, unlike hardware resources where service received from any of a set of similar servers is usually equivalent. In summary, for all the afore-mentioned reasons, the policies used for hardware resource scheduling are not directly applicable to data access scheduling.

The data access scheduling policies used in database systems are commonly referred to as *concurrency control* protocols. Concurrency control protocols preserve database integrity by resolving non-serial concurrent transaction data accesses in a manner that induces a serialization order among the conflicting transactions. Several concurrency control mechanisms have been proposed [Bern87], such as *locking*, *validation* (or *optimistic methods*), and *timestamping*. Each of these mechanisms takes a different approach to achieving serializability. The performance tradeoffs of these various mechanisms have been investigated in depth for conventional database systems (e.g. [Fran85, Agra87]), where the performance objective is usually to minimize the mean response times of transactions. However, these tradeoffs have to be reevaluated for *real-time* database systems as the performance objectives of an RTDBS differ from those of a conventional DBMS. The goal of an RTDBS is to meet transaction deadlines, therefore how early a transaction completes relative to its deadline is not as important as whether or not it completes by its deadline. Due to this difference in objectives, the performance results obtained for data access scheduling in conventional DBMSs may not carry over to RTDBSs.

There are two major issues that need to be explored with regard to real-time data access scheduling: First, how do we adapt the various concurrency control mechanisms to the real-time domain? Second, how do these concurrency control mechanisms compare in their real-time performance? In this paper, we address these questions for a specific real-time application framework. Within this framework, we compare the performance of several real-time locking protocols and optimistic concurrency control algorithms, using a detailed RTDBS simulation model as the evaluation tool.

The results of our study show that, in moving from the conventional DBMS domain to the RTDBS domain, there are new performance-related forces that come into effect. These new factors are a consequence of the real-time environment and therefore have not arisen in the performance studies of conventional database systems. Our experiments demonstrate that these factors can cause performance recommendations that were valid in a conventional DBMS setting to be significantly altered in the corresponding RTDBS setting. In this paper, we highlight these real-time-specific performance factors and evaluate their impact on concurrency control performance under a variety of real-time transaction workloads.

The remainder of this paper is organized in the following fashion: In Section 2, we describe the real-time application framework that is considered in our study. Related work on real-time data access scheduling is reviewed in Section 3. Then, in Section 4, we describe the functioning of the concurrency control algorithms that are compared in this study and discuss, at an intuitive level, their real-time strengths and weaknesses. The RTDBS simulation model is presented in Section 5, and the results of the performance experiments are highlighted in Section 6. Finally, in Section 7, we summarize the main conclusions of the study and outline future research avenues.

2. REAL-TIME FRAMEWORK

Real-time applications can be grouped into three categories: **Hard Deadline**, **Firm Deadline**, and **Soft Deadline**. The grouping is based on how the application is impacted by the violation of task time constraints. For hard deadline applications, missing a deadline is equivalent to a catastrophe. For firm deadline or soft deadline applications, however, missing deadlines leads to degraded performance but does not entail catastrophic results. The distinction between firm deadline and soft deadline applications lies in their treatment of late tasks. For a firm deadline application, completing a task after its deadline has expired is of no utility and may even be harmful. Therefore, late tasks are permanently aborted (i.e., "killed") and discarded. For a soft deadline application, however, there is some (diminished) utility to completing tasks even after their deadlines have expired.

In this study, we restrict our attention to firm deadline database applications, and late transactions are therefore discarded by the real-time database system. We choose to selectively investigate firm deadline applications for the following reasons: First, database systems for efficiently supporting hard deadline applications, where *all* transaction deadlines have to be met, appear infeasible due to the large variance between the average case and worst case execution times of a typical database transaction [Stan88]. Second, we expect that understanding scheduling issues for firm deadline applications will provide a foundation for addressing the more complex framework of soft deadline applications, where transactions may retain some utility if completed after their deadlines.

For the purposes of this study, we assume that the objective of the RTDBS is to minimize the number of missed transaction deadlines. We also assume that the RTDBS has no a-priori knowledge about transaction processing requirements, as such knowledge is difficult to obtain in a general database context where queries and updates are embedded in application programs. This lack of knowledge means that transactions are detected to be late only when they *actually* miss their deadlines, as the system cannot estimate their remaining service requirements. It also means that, from the system's perspective, transactions are distinguished only by their arrival times and deadlines.

Since satisfaction of transaction timing constraints is the primary goal (rather than other considerations, such as fairness), the RTDBS scheduling policies can be reasonably expected to be priority-driven; the priority assignment scheme is tuned to minimize the number of discarded transactions. Our simulation model implements a database system architecture wherein a *priority mapper* unit assigns a priority to each transaction on its arrival. These priorities are then used by the various system schedulers in resolving contention among transactions for hardware resources and data objects. This priority architecture shields the internal database mechanisms from the details of the priority assignment process, and is modular since it separates *priority generation* from *priority usage*.

3. RELATED WORK

Real-time database systems are a recent concept, and have received attention only during the last few years. The problem of scheduling transactions with the objective of minimizing the number of late transactions was first addressed in [Abbo88, Abbo89]. In these studies, the performance of several different concurrency control protocols, all of which used locking as the underlying serialization mechanism, were evaluated through simulation. The experimental results showed real-time locking protocols based on a priority inheritance approach [Sha87] to perform better than those based on a priority abort approach [Abbo88]. (These priority schemes are described in detail in Section 4.)

A simulation-based study of the relative performance of optimistic techniques and locking protocols in a real-time environment was presented in [Hari90a]. Optimistic algorithms were found to perform significantly better than their locking counterparts in the experiments of that study, especially when data contention was the primary performance-limiting factor. That work was extended in [Hari90b], where several new real-time optimistic algorithms that delivered improved real-time performance were presented and evaluated.

Similar real-time concurrency control studies have also been conducted as part of the SPRING project [Rama89]. A feature of these studies [Huan89, Huan91a, Huan91b] is that they were conducted on a real-time database testbed, RT-CARAT, and their results therefore include the effect of implementation overheads. Testbed limitations constrained the studies to consider a closed system with a fixed amount of resources; in addition, disk scheduling was under the control of the operating system and therefore did not incorporate transaction priorities. Real-time concurrency control algorithms based on locking were compared in [Huan89, Huan91b]. In contrast to the results of [Abbo89], their experiments showed locking protocols based on priority abort to perform considerably better than those based on priority inheritance. A comparative study of real-time optimistic techniques and locking protocols was reported in [Huan91a]. The experiments of this study showed optimistic algorithms to outperform locking protocols under low data contention. At high data contention, however, it was the

locking protocols which outperformed the optimistic algorithms; these results differed from those seen in [Hari90a, Hari90b].

From the above summary, we note that there have been several recent studies of real-time concurrency control; some of the results of these studies have been contradictory in nature. There are significant workload, system, and implementation differences between the studies conducted by the various research groups, thus making it difficult to pinpoint the source of the performance discrepancies. We attempt to reconcile some of these contradictions in this paper.

Data access scheduling algorithms that *combine* different concurrency control mechanisms have also been proposed recently. In [Lin90], a real-time concurrency control algorithm that has the flavor of both locking and optimistic methods was presented, while an algorithm that combines timestamp and optimistic techniques was described in [Cook91]. Based on a qualitative analysis, it was conjectured that these "mixed" algorithms would perform better than algorithms based on a single serialization mechanism. To our knowledge, however, a quantitative performance study that supports these claims has not yet been published.

4. CONCURRENCY CONTROL ALGORITHMS

In this section, we discuss the various locking and optimistic concurrency control algorithms that are evaluated in our study. For each algorithm class, we first present the basic conventional protocol and then describe real-time adaptations of this basic protocol. The description of each algorithm is supplemented with a discussion, at an intuitive level, of its potential strengths and weaknesses in the real-time domain. (We consider here only concurrency control algorithms based exclusively on locking or optimistic concurrency control; hybrid algorithms (e.g. [Lin90]) are not included in the scope of our study.)

4.1. Conventional Locking (2PL)

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. Multiple transactions can simultaneously share a read lock on a data item, but write locks are exclusive. If a lock request is denied, the requesting transaction is blocked until the lock is released. Locks obtained by a transaction during the course of its execution are held until the transaction commits, at which time it simultaneously releases all of its locks. Deadlocks are possible with the 2PL protocol, and therefore a deadlock detection scheme is required to find deadlocks. When a deadlock is found, it is broken by restarting one of the transactions in the cycle of waiting transactions.

Most commercial conventional database systems use 2PL as their concurrency control mechanism. This is because 2PL's blocking-based conflict resolution policy results in conservation of resources, thus delivering good performance under resource-limited conditions, and because recovery methods for use with 2PL are well understood (e.g. [Gray81]). A potential drawback in the RTDBS environment, however, is that 2PL does not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* [Sha87]. Priority inversion can cause the affected high-priority transactions to miss their deadlines.

4.2. Real-Time Locking Algorithms

Two different schemes, *priority inheritance* [Sha88] and *priority abort* [Abbo88], have been proposed as basic mechanisms for incorporating priority in locking protocols. Real-time locking algorithms based on each of these schemes were studied in [Abbo89] and are described below.

4.2.1. 2PL Wait Promote (2PL-WP)

The 2PL Wait Promote algorithm [Abbo89] is identical to basic 2PL in its resolution of conflicts, that is, transactions always block whenever a lock request is denied. A difference, however, is that it includes a priority inheritance mechanism. With this mechanism, whenever a requester blocks behind a lower-priority lock holder, the lock holder's priority is promoted to that of the requester. In other words, the lock holder *inherits* the priority of the lock requester, and the holder retains this elevated priority until it either commits or is restarted (due to deadlock resolution). When a data item is locked by more than one transaction, only those lock holders that have a lower priority than the requester inherit the priority of the requester. This algorithm guarantees that every transaction holding a lock on a data item has a priority that is at least as high as that of the highest priority transaction waiting for the lock. An important point to note here is that a transaction's inherited priority becomes its priority at *all* the resources in the system.

The 2PL-WP algorithm retains the resource-conservation features of 2PL. In addition, it reduces the blocking time of high priority transactions by increasing the priority of conflicting low priority lock holders (these low priority transactions execute faster and therefore release their locks earlier). A drawback, however, is that the blocking times of high-priority transactions are still uncertain in their duration [Huan91b]. In fact, under high data contention, where data conflicts are frequent, priority inheritance could result in most or all of the transactions in the system executing at the same priority. In this situation, the behavior of the RTDBS would effectively reduce to that of a conventional DBMS.

4.2.2. 2PL High Priority (2PL-HP)

The 2PL High Priority algorithm modifies the basic 2PL protocol by incorporating a *High Priority* [Abbo88] conflict resolution scheme which ensures that high priority transactions are not delayed by low priority transactions. The High Priority scheme resolves all data conflicts immediately in favor of the transaction with the higher priority. In particular, when a transaction requests a lock on an object held by one or more lower priority transactions in a conflicting lock mode, the lock-holding transactions are restarted and the requester is granted the lock. If the requester's priority is lower than that of any of the lock holders, it waits for the object to be released (the wait queue for an object is managed in priority order). In addition, a new reader can join a group of lock-holding readers only if its priority is higher than that of all the writers waiting for the lock. A secondary benefit of the High Priority scheme is that it also serves as a deadlock prevention mechanism.[1]

The 2PL-HP algorithm ensures that high priority transactions do not "see" lower priority transactions, and thus helps these urgent transactions to meet their deadlines. A drawback, however, is that a transaction may be restarted by a higher priority transaction that later misses its deadline and is discarded. This means that the restart did not result in the higher priority transaction meeting its deadline. In addition, it may cause the lower priority transaction to miss its deadline as well, apart from the loss of system resources due to the restart. Therefore, such *wasted restarts* may result in performance degradation. Also, 2PL-HP loses some of basic 2PL's beneficial blocking factor due to the partially restart-based nature of the High Priority scheme.

4.3. Conventional Optimistic Concurrency Control (OPT)

In classical optimistic concurrency control (OPT) [Kung81], transactions read and update data items freely, storing their updates into a private workspace. These updates are made public at commit time. Before a transaction is allowed to commit, however, it has to pass a validation test. This test checks that there is no conflict of the validating transaction with transactions that committed since it began execution. The validating transaction is restarted if it fails this test.

A variant of the above algorithm incorporates the *Broadcast Commit* scheme suggested in [Mena82, Robi82].[2] Here, when a transaction commits, it identifies other currently executing transactions that it conflicts with 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, already restarted the validating transaction at its (the committed

---

[1] This is true only for priority assignment policies that assign unique priority values and do not dynamically change the relative priority ordering of concurrently executing transactions.

[2] The Broadcast Commit scheme is also sometimes referred to as "forward" optimistic concurrency control [Haer84].

transaction's) own earlier commit time. This also means that a validating transaction is always guaranteed to commit. The broadcast commit variant detects conflicts earlier than the classical OPT algorithm, resulting in fewer wasted resources and earlier restarts. In the rest of this paper, we will refer to this variant as the basic OPT algorithm.

Optimistic algorithms, due to their entirely restart-based conflict resolution policy, tend to waste resources since a restart necessitates previously performed work to be redone. This results in their performing poorly in conventional DBMSs operating under resource-limited conditions [Agra87]. In an RTDBS environment, however, OPT implicitly derives a blocking effect due to *resource contention*: low priority transactions wait when resources are captured by high priority transactions. This blocking effect can decrease data conflicts since low priority transactions that may conflict with high priority transactions are effectively prevented from making significant progress by the priority-based resource scheduling. Moreover, if a conflict does occur and a low priority transaction has to be restarted, the resource wastage is at least reduced.

A second drawback of OPT in conventional DBMSs is that data conflicts are detected and resolved only at transaction commit time, and this delayed conflict resolution causes additional resources to be wasted. In an RTDBS, however, delayed conflict resolution can actually aid in making *better decisions* since more information about the conflicting transactions is available at commit time when the conflict is resolved. For example, 2PL-HP's problem of "wasted restarts" cannot occur with OPT. This is because, with OPT, a transaction that reaches its validation stage is *guaranteed* to commit and to complete before its deadline. Since only validating transactions can cause restarts of other transactions, there is *no* possibility of having wasted restarts.

A potential drawback of OPT in the RTDBS environment is that it does not take transaction priorities into account – low-priority transactions that reach validation unilaterally commit and cause conflicting higher priority transactions to be restarted. This may result in the affected high priority transactions missing their deadlines. In addition, there is the question of whether the beneficial aspects of delayed conflict resolution (no wasted restarts) outweigh its negative aspects (increased resource wastage by transactions that are eventually restarted).

4.4. Real-Time Optimistic Algorithms

In developing real-time optimistic algorithms, the goal is to prevent low priority validating transactions from unilaterally committing when they conflict with higher priority transactions. Two different mechanisms, *priority sacrifice* and *priority wait*, were proposed in [Hari90b] to address this problem, and algorithms based on these mechanisms are described in this section.

In the subsequent discussion, we will use the term *conflict set* to denote the set of currently executing transactions that conflict with a validating transaction. The acronym *CHP (Conflicting Higher Priority)* is used to refer to transactions that are in the conflict set and have a higher priority than the validating transaction. Similarly, we use the acronym *CLP (Conflicting Lower Priority)* to refer to transactions that are in the conflict set and have a lower priority than the validating transaction.

4.4.1. OPT-SACRIFICE

The OPT-SACRIFICE algorithm modifies the basic OPT protocol by incorporating a priority sacrifice mechanism. In this algorithm, a transaction that reaches its validation stage checks for conflicts with currently executing transactions. If conflicts are detected and one or more of the transactions in the conflict set is a higher priority transaction, then the validating transaction is restarted – that is, it is *sacrificed* in an effort to help the CHP transactions make their deadlines. The validation algorithm of OPT-SACRIFICE can therefore be written as:

```
if CHP transactions in conflict set then
   restart the validating transaction;
else
   restart transactions in conflict set;
   commit the validating transaction;
```

OPT-SACRIFICE satisfies the goal of giving preferential treatment to high priority transactions. It suffers, however, from the potential problem of *wasted sacrifices*, where a transaction is sacrificed on behalf of another transaction that is later discarded. Such sacrifices are useless and cause performance degradation. This drawback of OPT-SACRIFICE is analogous to the "wasted restarts" problem of 2PL-HP.

4.4.2. OPT-WAIT

The OPT-WAIT algorithm modifies the basic OPT protocol by incorporating a priority wait mechanism. In this algorithm, a transaction that reaches validation and finds CHP transactions in its conflict set is "put on the shelf", that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. While a transaction is waiting, it is possible that it will be restarted due to the commit of one of the CHP transactions. The validation algorithm of OPT-WAIT can therefore be written as:

```
while CHP transactions in conflict set do
      wait;
restart transactions in conflict set;
commit the validating transaction;
```

There are several features of the priority wait mechanism that may have a positive impact on performance: First, precedence is given to high-priority transactions, thus helping them to meet their deadlines. Second, the problem of "wasted sacrifices" does not exist here because the waiting transaction cannot be restarted by CHP transactions that miss their deadlines and are discarded. In other words, all restarts are made "on demand" and at the commit time of a higher priority transaction. Also, if a waiter's CHP transactions are all discarded due to missing their deadlines, then the waiter is immediately "taken off the shelf" and committed. Third, since transactions wait instead of immediately restarting, a blocking effect is derived – this results in conservation of resources, which can be beneficial to performance [Agra87]. Finally, the fact that a CHP transaction commits does not necessarily imply that the waiting transaction will be restarted. This is because, although the waiter conflicts with the high-priority transaction, the converse may not be true, that is, data conflicts may be *uni-directional* [Robi82]. For uni-directional conflicts, therefore, the waiting transaction can commit immediately after the CHP transaction has committed (if no other CHP transactions remain). This means that the data conflict between a waiter and a higher priority transaction may possibly be resolved without a restart of either transaction. Therefore, the priority wait mechanism has the potential to actually *eliminate* some data conflicts [Hari90b].

While the waiting scheme appears to have many positive features, it has some drawbacks as well. One potential drawback is that if a transaction finally commits after waiting for some time, it causes all of its CLP transactions to be restarted at a later point in time. The delayed restart decreases the chances that these transactions will meet their deadlines, and also results in more wasted resources. A second drawback is that the validating transaction may develop new conflicts during its waiting period, thus causing an increase in conflict set sizes and leading to more restarts. Another way to understand this is to realize that waiting causes objects to be, in a sense, "locked" for longer periods of time. Therefore, while waiting has the capability to reduce the probability of a restart-causing conflict between a given pair of transactions, it simultaneously increases the probability of having a greater number of conflicts per transaction. This increase may be substantial when there are a large number of transactions in the system.

4.4.3. WAIT-50

The WAIT-50 algorithm is an extension of the OPT-WAIT algorithm – in addition to the priority wait mechanism, it incorporates a *wait control* mechanism. The wait control mechanism monitors transaction conflict states and dynamically decides when, and for how long, a validating transaction should be made to wait for the higher priority transactions in its conflict set. A transaction's conflict state is assumed to be characterized by the index *HPpercent*, which is the percentage of the transaction's total

conflict set size that is formed by CHP transactions. The operation of the wait mechanism is conditioned on the value of this index. In the WAIT-50 algorithm, a simple "50 percent" rule is used – a validating transaction is made to wait only while HPpercent is greater than or equal to 50, that is, while half or more of its conflict set is composed of higher priority transactions. (The rationale for the choice of 50 percent is provided in Section 6.) The validation algorithm of WAIT-50 can therefore be written as:

```
while CHP transactions in conflict set and
          HPpercent ≥ 50 do
      wait;
restart transactions in conflict set;
commit the validating transaction;
```

The aim of the wait control mechanism is to detect when the beneficial effects of waiting, in terms of giving preference to higher priority transactions and decreasing pairwise conflicts, are outweighed by its drawbacks, in terms of later restarts and increased conflict set sizes. Therefore, while OPT-WAIT and OPT represent the extremes with regard to waiting – OPT-WAIT always waits for a CHP transaction, and OPT never waits – WAIT-50 is a *hybrid* algorithm that dynamically controls the amount of waiting. In fact, we can view OPT, WAIT-50, and OPT-WAIT as all being special cases of the general algorithm **WAIT-X**, where X is the cutoff HPpercent level, with X taking on the values ∞, 50 and 0, respectively.

4.5. Summary

Conventional locking and conventional optimistic concurrency control represent two extremes in terms of data conflict detection and data 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. In this section, we described the conventional locking and (forward) optimistic concurrency control algorithms (2PL and OPT), and discussed several real-time variants of these algorithms (2PL-WP, 2PL-HP, OPT-SACRIFICE, OPT-WAIT, and WAIT-50). The real-time concurrency control algorithms all aim to meet more transaction deadlines by preferentially serving the urgent transactions. They incorporate different priority mechanisms such as priority inheritance and priority abort in the locking algorithms, and priority sacrifice and priority wait in the optimistic algorithms.

We conducted experiments to evaluate the real-time performance of the various locking and optimistic concurrency control algorithms described here, and the following sections describe the experimental framework and the results of the experiments.

5. REAL-TIME DBMS PERFORMANCE MODEL

A detailed model of a real-time database system was used to study the performance of the various concurrency control algorithms (see [Hari91] for a complete model description). In this model, the database system configuration consists of a shared-memory multiprocessor operating on disk resident data (for simplicity, we assume that all data is accessed from disk and buffer pool considerations are therefore ignored.)[3] The database itself is modeled as a collection of data pages.

Transactions arrive in a Poisson stream and each transaction has an associated deadline. A transaction consists of a sequence of read page and write page operations. A read page operation 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 page operations are handled similarly except for their disk I/O – their disk activity is deferred until the transaction has committed. Here we assume that the RTDBS has sufficient buffer space to allow the retention of updates until commit time, and we also assume the use of a log-based recovery scheme where only log pages are forced to disk prior to commit. A transaction that is restarted due to a data conflict follows the same data access pattern as its original incarnation. If a transaction has not completed by its deadline, it is immediately aborted and discarded. The basic structure of the model is shown in Figure 1.

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 completed transactions. The *priority mapper* unit (described in Section 2) is embedded in the transaction manager. The following two subsections describe the workload generation process and the hardware resource configuration.

5.1. Workload Model

The workload model characterizes transactions in terms of the data 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 mean rate of transaction arrivals. The number of pages accessed by a transaction varies uniformly between half and one-and-a-half times the value of *TransSize*. Page requests are generated from a uniform distribution (without replacement) spanning the entire database. A page that is read is updated with probability *WriteProb*. Therefore, a page write operation is always preceded by a read for the same page; this means that the write set of a

[3] While modeling buffering would certainly result in different absolute performance numbers, we do not expect that doing so would significantly alter the relative performance behavior of the concurrency control algorithms.

transaction is a subset of its read set and that there are no "blind writes" [Bern87].

In each of our experiments, a single formula is used to assign deadlines to all transactions, and the choice of formula is determined by the *DeadlineAssignment* parameter. We use two different deadline assignments in this study. The first assignment, DA1, is:

$$D_T = A_T + SF * E_T \qquad \text{(DA1)}$$

where D_T, A_T and E_T are the deadline, arrival time, and execution time of transaction T, respectively (the method for computing E_T is discussed in Section 5.3). SF is a *slack factor* that provides control over the tightness/slackness of deadlines. With this formula, all transactions have the same *slack ratio* – this is defined to be the ratio $\frac{D_T - A_T}{E_T}$ and is equal to the slack factor SF in DA1. The physical interpretation of this ratio is that it is the number of completion opportunities available to the transaction, given its deadline. (A slack ratio of less than 1 implies that it is impossible for the transaction to complete before its deadline.)

The second deadline assignment, DA2, is:

$$D_T = A_T + Uniform(LSF, HSF) * E_{\max} \qquad \text{(DA2)}$$

The transaction execution time used in this assignment, $E_{\max}$, is the execution time of the largest transaction in the workload (i.e., a transaction accessing 1.5 * *TransSize* pages). Each transaction's slack factor is uniformly chosen over the range set by LSF and HSF. With this deadline assignment, transaction slack ratios are spread over a range of values based on the ratio of $E_{\max}$ to the E_T's and the spread in slack factors. The DA2 assignment makes the deadline of a transaction *independent* of its execution time, unlike DA1 where transaction deadlines are linearly dependent on their execution times.

The LSF and HSF workload parameters in Table 1 set the slack factors to be used in the deadline formulas. (For DA1, both these parameters are set to the same number and SF takes on this

Parameter	**Meaning**
ArrivalRate	Transaction arrival rate
TransSize	Avg. transaction size (in pages)
WriteProb	Write probability per accessed page
DeadlineAssignment	DA1 or DA2
LSF	Low Slack Factor
HSF	High Slack Factor

Table 1: Workload Model Parameters

value.) While the simulation model's workload generator utilizes information about transaction execution times in assigning deadlines, it is important to note that the real-time database system itself is assumed to have no access to such information (as discussed in Section 2).

5.2. System Model

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 preemption being based on transaction priorities. Each of the disks has its own queue and is scheduled with a priority Head-Of-Line scheduling policy [Abbo89]. Table 2 summarizes the key parameters of the system model. The *DatabaseSize* parameter gives the number of pages in the database, and the data pages are modeled as being uniformly distributed across all of the disks. The *NumCPUs* and *NumDisks* parameters specify the hardware resource configuration, while the *PageCPU* and *PageDisk* parameters capture the CPU and disk processing times per data page.

5.3. Execution Time Computation

As mentioned earlier, the deadline assignments used in this study, DA1 and DA2, incorporate transaction execution times in their deadline computations. The execution time of a transaction, E_T, is computed with the following expression

$$E_T = NumReads_T * (PageCPU + PageDisk) + NumWrites_T * PageCPU;$$

where $NumReads_T$ and $NumWrites_T$ are the number of pages that are read and updated by the transaction, respectively. The disk time for writing updated pages is not included in the resource time computation since these writes occur *after* the transaction has committed.

5.4. Concurrency Control Overhead

There are no explicit concurrency cost control parameters included in our system model since we assume that the overhead of performing concurrency control is small compared to data processing

Parameter	Meaning
DatabaseSize	Number of pages in database
NumCPUs	Number of processors
NumDisks	Number of disks
PageCpu	CPU time for processing a data page
PageDisk	Disk service time for a page

Table 2: System Model Parameters

times. Moreover, a rough equivalence between the concurrency control costs for locking and optimistic concurrency control was established in [Care83]; we therefore do not expect the omission of these overheads to bias our results.

6. EXPERIMENTS and RESULTS

In this section, we present the performance results from our experiments comparing the various locking protocols and optimistic concurrency control algorithms in a real-time database system environment. The simulator used to obtain the results was written in DeNet [Livn88], a Modula-2 based simulation language. The transaction priority assignment in all of the experiments described here is *Earliest Deadline* – transactions with earlier deadlines have higher priority than transactions with later deadlines. Earliest Deadline is widely used in real-time systems, and does not require knowledge of transaction processing requirements, thus making it suitable for our operating constraints.

During the discussion of the concurrency control algorithms in Section 4, it was mentioned that deadlocks are possible with the 2PL and 2PL-WP locking algorithms. In our simulator, deadlock detection was initiated for these two algorithms whenever a transaction blocked, thereby detecting deadlocks as soon as they occurred. Deadlocks were resolved by aborting the transaction with the lowest priority among the transactions involved in the deadlock (for 2PL-WP, the *original* priorities of the deadlocked transactions were used in determining the victim).[4]

The validation test of algorithms based on forward optimistic concurrency control involves checking for conflict with the read sets of active transactions. This raises some implementation difficulties since the data sets of active transactions are dynamically changing [Haer84]. For simplicity, the problem was bypassed in the simulator by modeling the validation test as an instantaneous operation. However, we expect this idealization to have little effect on our results since a mechanism for implementing the validation test with minor overhead is described in [Hari91].

In the following presentation, we first describe the study's performance metric and then list the settings used for the system parameters. Subsequently, we discuss our results with regard to the impact of resource contention, data contention, and variations in workload characteristics.

6.1. Performance Metric

The performance metric of our experiments is *MissPercent*, which is computed as

[4] The original priorities, and not the inherited priorities, are used for deadlock resolution with 2PL-WP since, if transactions *have* to be restarted, we would like the restarted transactions to be the least urgent transactions.

$$MissPercent = \left(\frac{No.\ of\ Input\ Transactions - No.\ of\ InTime\ Transactions}{No.\ of\ Input\ Transactions} \right) * 100$$

Thus, MissPercent is the percentage of input transactions that the system is *unable* to complete before their deadlines. MissPercent values in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 20 to 100 percent represent system performance under "heavy" loads. A long-term operating region where the miss percentage 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. All MissPercent graphs in this paper show mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level, with each experiment having been run until at least 5000 transactions were processed by the system. Only statistically significant differences are discussed here.

The simulator was instrumented to generate a host of other statistical information, including CPU and disk utilizations, number of transaction restarts, mean system population, etc. These secondary measures help to explain the MissPercent behavior of the concurrency control algorithms under various workloads and system conditions.

6.2. Parameter Settings

The resource parameter settings are such that the CPU time to process a page is 10 milliseconds while disk access times are 20 milliseconds. For experiments that were intended to factor in the effect of resource contention on the performance of the algorithms, the number of processors and number of disks were set to 10 and 20, respectively. For experiments that were intended to isolate the effect of data contention, an "infinite" resources situation [Tay84, Fran85, Agra87], where there is no queueing for resources, was simulated. A point to note here is that 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 often the driving consideration.

We began our performance evaluation by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. To serve as a basis for comparison, the performance achievable in the *absence* of concurrency control is also shown on the graphs, under the title NO-CC, for experiments in which the RTDBS has limited resources.

6.3. Experiment 1: Baseline Experiment

The settings of the workload parameters and system parameters for the baseline experiment are listed in Table 3. These settings were chosen with the objective of having significant data and resource contention in the system, thus helping to bring out the performance differences between the concurrency control algorithms. Transaction deadlines are assigned using deadline assignment DA1, which assigns the *same* slack ratio to all transactions. For this experiment, Figures 2a and 2b show the MissPercent behavior under normal load and heavy load conditions, respectively.

Focusing our attention on the *locking* algorithms (2PL, 2PL-HP, 2PL-WP) in these graphs, we observe that the priority abort-based algorithm, 2PL-HP, performs the best under both normal loads and heavy loads. 2PL-HP performs better than 2PL because it ensures that urgent transactions are not delayed by transactions with later deadlines, unlike 2PL whose priority-indifferent blocking policy results in long waiting times under high contention, thus causing urgent transactions to miss their deadlines. Interestingly, the performance of the priority inheritance-based algorithm, 2PL-WP, is only slightly better that of 2PL. This is because, under high data contention, urgent transactions are repeatedly blocked by low priority transactions and the priority inheritance mechanism results in many transactions executing at the same priority. Consequently, high-priority transactions effectively receive little or no preferential treatment and the objective of providing service to transactions based on the urgency of their deadlines is not realized.

Comparing the above results for locking algorithms to those of the *optimistic* concurrency control algorithms, we observe that OPT, the conventional optimistic algorithm, performs better than all the locking algorithms, including 2PL-HP. This is a surprising result since OPT is potentially wasteful of resources and is indifferent to transaction priorities, and could therefore be expected to perform worse than 2PL-HP. If we compare the average number of restarts suffered by a transaction with OPT and with 2PL-HP, however, we find that 2PL-HP has many more restarts than OPT, as shown in Figure

Workload Parameter	**Value**	**System Parameter**	**Value**
TransSize	16 pages	*DatabaseSize*	1000 pages
WriteProb	0.25	*NumCPUs*	10
DeadlineAssignment	DA1	*NumDisks*	20
LSF	4.0	*PageCPU*	10 ms
HSF	4.0	*PageDisk*	20 ms

Table 3: Baseline Parameter Settings#

2c.[5] This phenomenon is due to the "wasted restarts" problem of 2PL-HP (described in Section 4.2.2), where even a transaction that is later discarded may restart other transactions. In contrast, with OPT, only a *committing* transaction can restart other transactions. At higher loads, when many transactions miss their deadlines and are discarded, 2PL-HP has significantly more restarts than OPT. This is brought out clearly in Figure 2c, 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. (For all algorithms, the number of restarts decrease after a certain load because resource contention, rather than data contention, becomes the more dominant reason for transactions missing their deadlines.)

The average progress made by transactions before they were restarted due to data conflicts is shown in Figure 2d (a transaction's progress is measured in terms of the fraction of its total execution that is completed). We observe in this figure that 2PL-HP consistently detects conflicts earlier than OPT, as should be expected based on the discussion in Section 4. One might therefore also expect 2PL-HP to waste less resources than OPT. However, since OPT has far fewer restarts, it actually makes better *overall* use of resources than 2PL-HP. This concept is quantified in Figure 2e, where the total utilization and the useful utilization of the processors (the bottleneck resource) are shown. Useful utilization is computed as the processor usage made by those transactions that eventually met their deadlines. From the utilization curves, it is clear that OPT is more resource-efficient than 2PL-HP. Therefore, for the workload of this experiment, the delayed conflict resolution policy of OPT proves to be beneficial in its overall effect.

Moving on to the real-time optimistic algorithms, we observe that OPT-SACRIFICE, although performing better than the locking algorithms, performs worse than the priority-indifferent OPT algorithm. The degraded performance of OPT-SACRIFICE is due to its "wasted sacrifices" problem (described in Section 4.4.1), where validating transactions restart themselves for higher priority transactions that are later discarded. In Figure 2d, we observe that the average restart point of transactions is noticeably greater with OPT-SACRIFICE than with any of the remaining optimistic algorithms. This is because the sacrifice-induced restarts of low priority transactions occur when they have completed their entire processing. The combination of later restarts and greater number of restarts results in OPT-SACRIFICE wasting more resources than OPT. This is quantified in Figure 2e, where the useful utilization of resources by OPT-SACRIFICE is noticeably smaller than that of OPT.

---

[5] The "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 input transactions.

Turning our attention to the priority-wait-based algorithms, OPT-WAIT and WAIT-50, we observe in Figure 2a that under normal loads, their performance is superior to that of OPT. This is due to the beneficial effects of their priority cognizance. Under heavy loads (Figure 2b), however, WAIT-50 and OPT-WAIT behave identically to OPT. This latter result is because, with high resource contention, it is uncommon for a low priority transaction to gain access to the resources. Consequently, transactions usually reach their validation stage only when their deadline is quite close, which means that the priority wait mechanism has very limited impact, and WAIT-50, OPT-WAIT, and OPT become essentially the same algorithm.

An important point to note here is that the transaction workload of this experiment could be expected, in the absence of deadlines, to generate exactly the *opposite* results in a resource-limited conventional DBMS [Agra87]: A locking algorithm would perform better (w.r.t. mean response time) than an optimistic algorithm because (a) it has no wasted restarts since all transactions are eventually executed to completion, and (b) it is better at conserving resources.

6.4. Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the impact of data contention on the performance of the concurrency control algorithms. For this experiment, therefore, the resources were made "infinite"[6], keeping all the other parameter values the same as those used in the baseline experiment. The MissPercent performance results for this system configuration are presented in Figures 3a and 3b. We observe in these figures that 2PL and 2PL-WP perform very poorly, and that 2PL-HP is superior to both of them. We also observe that conventional OPT performs better than 2PL-HP over virtually the entire loading range. There are two reasons for OPT outperforming 2PL-HP here: First, the wasted restarts problem of 2PL-HP, as outlined earlier for the baseline experiment, occurs here too. This effect is shown in Figure 3c. Second, the blocking component of 2PL-HP reduces the number of transactions that are executing and making progress. This blocking causes an increase in the average number of transactions in the system, thus generating more conflicts and a greater number of restarts. With OPT, however, transactions are always executing and are never blocked. This effect is quantified in Figure 3d, which shows the mean number of transactions in the system for the different algorithms.

Moving on to the real-time optimistic algorithms, we observe that OPT-SACRIFICE performs significantly worse relative to the priority-wait-based algorithms than it did under finite resources. For the most part, OPT-SACRIFICE also performs worse than OPT. The performance of OPT-SACRIFICE is further degraded here since the high data contention levels lead to a steep increase in

[6] As mentioned in Section 6.2, infinite resources means that there is no queueing for resources.

the number of conflicts and, consequently, in the number of "wasted sacrifices".

Turning our attention to OPT-WAIT, we observe that it performs the best at low levels of data contention due to the beneficial effects of its priority wait mechanism. As data contention increases, however, its performance steadily degrades. Finally, at high data contention levels, it performs worse than OPT. The reason for OPT-WAIT's degraded performance in this region is that the priority wait mechanism causes a significant increase in the average number of transactions in the system, as shown in Figure 3d. This increase in transaction population leads to an increased number of data conflicts and to delayed restarts of low priority transactions, thus having an adverse effect on performance.

Finally, we observe that the WAIT-50 algorithm provides the best *overall* performance. It behaves like OPT-WAIT under low data contention, and behaves like OPT under high data contention. The explanation for this behavior of WAIT-50 is provided in the next experiment.

An important observation here is that while resource contention can be reduced by purchasing more and/or faster resources, there exists no equally simple mechanism to reduce data contention. It should also be noted that optimistic algorithms perform better than locking protocols under infinite resource conditions in a conventional DBMS setting as well [Fran85, Agra87].

6.5. Experiment 3: Variable Slack Ratio

Our next experiment investigated the case where transactions may have different slack ratios. For this experiment, we used deadline assignment DA2, which makes transaction deadlines independent of their execution times, and results in transactions having a range of slack ratios. The deadline-related workload parameters, *LSF* and *HSF*, were set at 1.33 and 4.0, respectively, with the remaining workload and system parameters being the same as those of the baseline experiment.

For this experiment, Figures 4a and 4b show the MissPercent behavior of the algorithms under normal load and heavy load conditions, respectively. We observe that the qualitative behavior of the various concurrency control algorithms is similar to that of the baseline experiment, with the optimistic algorithms generally performing better than the locking algorithms. A difference, however, is that the wait-based algorithms, WAIT-50 and OPT-WAIT, now perform noticeably better than OPT under normal loads. In fact, 2PL-HP and OPT-SACRIFICE also perform better than OPT at low loads. The reason for the degraded performance of OPT is that transactions with small slack ratios (relative to the slack ratios of other transactions) are present in the workload. Such transactions have limited completion opportunities and OPT's policy of permitting low priority validating transactions to restart high priority transactions therefore results in an increased number of missed deadlines among the transactions with small slack ratios. In the previous experiments, the detrimental effects of OPT's indifference to transaction priorities were reduced because all transactions had the same slack ratio.

When the same experiment is carried out under infinite resources, Figures 5a and 5b are obtained. We observe that the performance improvement of the wait-based algorithms over OPT at normal loads is greater in these figures relative to the corresponding performance under finite resources. The reason for this behavior is the following: In the presence of resource contention, the priority indifference of OPT is masked to some extent by the priority scheduling at the resources. Under pure data contention, however, the negative effects of OPT's priority-indifference show up in their entirety.

Considering performance at high loads, we observe that OPT-WAIT performs worse than OPT. This is due to the beneficial aspects of waiting being more than countered by its negative aspects in terms of later restarts and increased conflicts. In contrast, WAIT-50, which had been behaving like OPT-WAIT at normal loads, now changes character and behaves like OPT. Once again, as in the earlier experiments, we find that WAIT-50 turns in the best overall performance by behaving like OPT-WAIT when data contention is low and like OPT when data contention is high.

The results of the experiments discussed so far have shown that WAIT-50 provides performance close to that of either OPT or OPT-WAIT in operating regions where they behave well, and provides the same or slightly better performance at intermediate points. Therefore, in an overall sense, WAIT-50 integrates priority and waiting into the optimistic concurrency control framework. Its control mechanism is fairly effective at deciding when the benefits of priority waiting are outweighed by its drawbacks. In Figure 5c, the "wait factor" of WAIT-50 is plotted with respect to that of OPT-WAIT. The wait factor measures the total time spent in priority-waiting using WAIT-50, normalized by the waiting time of OPT-WAIT.[7] From this figure, it is clear that WAIT-50's wait factor is close to that of OPT-WAIT at low contention levels but decreases steadily as the data contention level is increased. Therefore, while OPT-WAIT and OPT represent the extremes with regard to priority-waiting, WAIT-50 controls the degree of waiting to match the level of data contention in the system.

6.6. Experiment 4: Wait Control Mechanism

Our next experiment examined the effect of the choice of 50 percent as the cutoff value for the HPpercent control index in the WAIT-50 algorithm. Keeping the workload and system parameters the same as those of Experiment 3, we measured the MissPercent performance of WAIT-25 and WAIT-75 under conditions of both finite and infinite resources. Figures 6a and 6b show the results of the finite resources experiment under normal load and heavy load, respectively, while Figure 7 gives the corresponding results under infinite resources. (For clarity, we show only the curves for the priority-wait based algorithms in these figures.)

[7] The wait factor of OPT is trivially zero as the algorithm has no wait component.

From these graphs, we observe that lowering the high-priority cutoff value to 25 percent results in slightly improved normal load performance but worsened heavy load performance. This behavior is due to the increased wait factor that is delivered by the decreased cutoff setting. On the other hand, raising the cutoff value to 75 percent has the opposite effect: the normal load performance becomes worse while the heavy load performance improves slightly. This behavior is due to the decreased priority cognizance that is delivered by the increased cutoff setting.

Based on these results, a 50 percent cutoff setting appears to establish a balanced tradeoff between the opposing forces of priority cognizance and increased data contention, thus providing good performance across the entire range of loading. The basic philosophy is that priority-based waiting is quite beneficial under light loads, when data contention levels are low. Under heavy loads, however, when data contention levels are high, waiting becomes detrimental to performance. WAIT-50 is effective in dynamically changing its behavior to match the level of data contention in the system.

In all of the experiments discussed so far, the real-time optimistic algorithms have outperformed the real-time locking algorithms over the entire range of loading, under conditions of both limited resources and infinite resources. In particular, WAIT-50, the best performing optimistic algorithm, has always been superior to 2PL-HP, the best among the locking algorithms, especially when data contention is the primary performance limiting factor. These results (and additional experimental results presented in [Hari90a, Hari90b]) might appear to suggest that WAIT-50 is always the preferred concurrency control algorithm for transaction workloads that have firm deadlines. However, as we will show in the following experiment, there are certain regions of operation where the disadvantages of delayed conflict resolution, in terms of later restarts and more wasted resources, can outweigh its benefits and result in WAIT-50 performing worse than 2PL-HP.

6.7. Experiment 5: Performance Crossover

In this experiment, we model a situation where transactions have a high write probability and the database size is large. For this experiment, the transaction write probability was set to 1.0 and the database size was increased to 10,000 pages, while keeping all the other parameters the same as those of the baseline experiment. The corresponding MissPercent behavior is shown in Figures 8a and 8b for normal load and heavy load conditions, respectively. (For graph clarity, we show only the best optimistic and locking algorithms, WAIT-50 and 2PL-HP, respectively.) In these figures we see that, unlike the previous experiments, it is now 2PL-HP which performs better than WAIT-50. The reason for this change in their relative performance behavior is explained in the restart curves shown in Figure 8c, where we see that the number of restarts of 2PL-HP is now significantly lower than that of WAIT-50 at normal loads. This is because the number of conflicts developed by each transaction is small due

to the large database size (compare the restart ratio magnitudes in Figure 8c to those in Figure 2c). In addition, relatively few transactions miss their deadlines in this loading region. Therefore, the degrading effect of restarts caused by discarded transactions is minimized, resulting in behavior similar to that in conventional DBMSs where locking-based algorithms have fewer restarts than optimistic algorithms. At heavy loads, resource contention is the dominant factor in causing transactions to miss their deadlines. Therefore, although 2PL-HP has higher number of restarts than WAIT-50 in this region, these excess restarts have little performance impact since they are restarts of transactions that will, in all likelihood, eventually miss their deadlines due to resource contention itself. If we consider the utilization curves in Figure 8d, it is clear that WAIT-50, due to its delayed conflict resolution and solely restart-based conflict resolution policy, wastes more resources and therefore performs worse than 2PL-HP.

We conducted another experiment where the arrival rate was fixed at 20 transactions/second, the write probability was set to 1.0 and the number of pages in the database was varied from 100 to 10,000. The results of this experiment are shown in Figure 9a and they clearly indicate that there is a crossover point, in terms of the database size, between the relative performance of 2PL-HP and WAIT-50. For database sizes smaller than the crossover point, the data contention generated is high enough to cause WAIT-50 to perform better than 2PL-HP due to 2PL-HP's problem of wasted restarts. In contrast, for database sizes larger than the crossover point, 2PL-HP's immediate detection of conflicts and its blocking factor result in its performing better than WAIT-50. This concept is quantified in the restart curves of Figure 9b.

In Experiments 1 through 4, the database size was smaller than the crossover size, and therefore these experiments showed WAIT-50 outperforming 2PL-HP. However, as demonstrated by the experiments just discussed, 2PL-HP performs better than WAIT-50 for database sizes greater than the crossover size. These experiments also help explain the apparent contradiction in results between the studies of [Hari90a, Hari90b] and [Huan91a] that was mentioned in Section 3. The [Hari90a, Hari90b] studies considered workloads in the small database region and therefore found WAIT-50 to outperform 2PL-HP. In contrast, the experiments of [Huan91a] considered workloads in the large database region and therefore observed 2PL-HP to outperform WAIT-50.

The database size in the [Huan91a] study, which implemented a closed system, was (MPL * 100 blocks) with each block containing 6 records. The average transaction length was 6 "steps", with 4 records accessed in each step. If we define the "database access ratio" to be the maximum number of objects that could be simultaneously accessed by all the transactions in the system relative to the size of the database, then the database access ratio of [Huan91a] study was (MPL * 6 * 4)/(MPL * 100 * 6) = 0.04. In the [Hari90a, Hari90b] RTDBS experiments, the database size was 1000 pages and the

average transaction length was 16 pages, with the mean population, depending on the arrival rate, varying from a few transactions to several tens of transactions. Therefore, with as few as three transactions in the system, the database access ratio = (3 * 16)/1000 = 0.05 was greater than that of the [Huan91a] study. At higher loads, the access ratios were substantially higher, resulting in high contention levels.

It was conjectured in [Huan91a] that a major cause for the differences in performance results was that their testbed-based study took algorithm implementation overheads into account, unlike the simulation-based studies of [Hari90a, Hari90b]. However, as the above experiment shows, the contradictions are primarily due to basic differences in workloads rather than differences in the systems or their implementations. This was confirmed in additional experiments that are not described here due to space considerations.

6.8. CONCLUSIONS

In this paper, we have addressed the problem of data access scheduling in a real-time database system supporting transactions with firm deadlines. In particular, we have made a quantitative study of the relative performance of locking and optimistic concurrency control techniques in the firm real-time domain. The performance metric of interest is the percentage of deadlines that are met, unlike a conventional DBMS where mean response time or throughput is usually the performance criterion. Using a detailed simulation model of an RTDBS, we studied the performance of the conventional locking algorithm (2PL), the conventional optimistic concurrency control algorithm (OPT), and several real-time variants of these algorithms under a range of workloads and system operating conditions.

Our experiments demonstrated that under sufficiently high data contention, optimistic algorithms outperform locking algorithms over a wide range of system loading and resource availability. This is a surprising result since optimistic algorithms perform worse than locking protocols in resource-limited conventional DBMSs. The improved performance of the optimistic algorithms seen here stems primarily from the firm-deadline feature of discarding late transactions. In this context, the delayed conflict resolution policy of optimistic algorithms aids them in making better conflict decisions than locking algorithms (which resolve conflicts immediately). By delaying conflict resolution to transaction commit time, the optimistic algorithms ensure that transactions which are destined to miss their deadlines do not impede the progress (w.r.t. data access) of other transactions in the system. The benefits of this feature were strong enough that even the conventional OPT algorithm outperformed the best real-time locking algorithm, 2PL-HP, over most of the loading range. This occurred because 2PL-HP was adversely affected by transactions that were eventually discarded.

Among the optimistic algorithms, WAIT-50 was observed to provide the best overall performance over a fairly wide range of workloads and operating conditions. The WAIT-50 algorithm

monitors transaction conflict states and gives precedence to urgent transactions in a controlled manner. It features a priority wait mechanism that provides preferential treatment to high priority transactions by forcing low priority validating transactions to wait for conflicting high priority transactions to complete first. While the priority waiting mechanism works well at low data contention levels, it can cause significant performance degradation at high contention levels by generating a steep increase in the number of data conflicts. A simple wait control mechanism consisting of a "50 percent" rule is used in the WAIT-50 algorithm to address this problem: If half or more of the transactions conflicting with a validating transaction are of higher priority, then the validating transaction is made to wait; otherwise, it is allowed to commit. This simple rule was found to be quite effective in detecting when the benefits of priority-waiting were outweighed by its drawbacks, and resulted in WAIT-50 performing well over the entire range of loading.

While the optimistic algorithms outperformed the locking algorithms under high data contention, the situation was reversed under limited data contention. This was because the performance degradation caused by eventually discarded transactions was minimized under these conditions. In this situation, the disadvantages of delayed conflict resolution, in terms of late restarts and wastage of resources, caused the optimistic algorithms to perform worse than the locking algorithms. Identifying these different workload regions helped us to resolve the apparent contradictions between the results described in [Hari90a, Hari90b] and those reported in [Huan91a].

Among the locking algorithms, 2PL-HP, which incorporates a priority abort mechanism, delivered the best performance. In contrast, the 2PL-WP algorithm, which incorporates a priority inheritance mechanism, performed almost the same as conventional 2PL for the workloads considered in our experiments. Similar behavior was observed for 2PL-WP in [Huan91b]. In [Abbo89], however, 2PL-WP was found to perform significantly better than 2PL-HP. We conjecture that the reason for the differences in results is that the [Abbo89] study considered a soft deadline transaction processing environment, where all transactions have to be executed to completion, while our present study and that of [Huan91b] have considered firm deadline transactions. Based on some preliminary results in [Hari90a], we expect the relative performance of locking and optimistic concurrency control algorithms in the soft-deadline framework to be similar to those observed in conventional DBMSs since firm deadline-induced problems such as wasted restarts do not arise in this environment. We plan to investigate this issue in greater detail in our future research.

Another interesting future research challenge is to develop an analytical model of the WAIT-50 algorithm and to theoretically account for the effectiveness of the 50 percent control rule. Also, WAIT-50's control mechanism, which monitors transaction conflict states, operates on a per-transaction basis. It would be instructive to compare its performance with that of a control mechanism that

monitors global data contention levels (e.g. [Care90]).

In summary, we have shown that the firm-deadline RTDBS feature of discarding late transactions can have a profound impact on the performance of concurrency control algorithms, resulting in performance recommendations quite different from those for the corresponding conventional DBMS.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions: A Performance Evaluation," *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions With Disk Resident Data," *Proc. of 15th Intl. Conf. on Very Large Data Bases*, August 1989.

[Agra87] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Systems*, 12(4), December 1987.

[Bern87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Care83] Carey, M., "An Abstract Model of Database Concurrency Control Algorithms," *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1983.

[Care90] Carey, M., Krishnamurthi, S., and Livny, M., "Load Control for Locking: The Half-and-Half Algorithm," *Proc. of 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, April 1990.

[Cook91] Cook, R., et al, "New Paradigms for Real-Time Database Systems," *Proc. of 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[Eswa76] Eswaran, K., et al, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of ACM*, 19(11), November 1976.

[Fran85] Franaszek, P., and Robinson, J., "Limitations of Concurrency in Transaction Processing," *ACM Trans. on Database Systems*, 12(1), March 1985.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Gray81] Gray, J., McJones, P., and Blasgen, M., "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, 13(2), June 1981.

[Haer84] Haerder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, 9(2), 1984.

[Hari90a] Haritsa, J., Carey, M., Livny, M., "On Being Optimistic about Real-Time Constraints," *Proc. of 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, April 1990.

[Hari90b] Haritsa, J., Carey, M., Livny, M., "Dynamic Real-Time Optimistic Concurrency Control," *Proc. of 11th IEEE Real-Time Systems Symposium*, December 1990.

[Hari91] Haritsa, J., "Transaction Scheduling in Firm Real-Time Database Systems," *Ph.D. Thesis*, Computer Sciences Dept., Univ. of Wisconsin, Madison, August 1991.

[Huan89] Huang, J., Stankovic, J., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *Proc. of 10th IEEE Real-Time Systems Symposium*, December 1989.

[Huan91a] Huang, J., Stankovic, J., Ramamritham, K. and Towsley, D., "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proc. of 17th Intl. Conf. on Very Large Data Bases*, September 1991.

[Huan91b] Huang, J., Stankovic, J., Ramamritham, K., and Towsley, D., "On Using Priority Inheritance in Real-Time Databases," *Proc. of 12th IEEE Real-Time Systems Symposium*, December 1991.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, 6(2), June 1981.

[Lin90] Lin, Y., and Son, S., "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *Proc. of 11th IEEE Real-Time Systems Symposium*, December 1990.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Computer Sciences Department, Univ. of Wisconsin, Madison, 1988.

[Mena82] Menasce, D., and Nakanishi, T., "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," *Information Systems*, 7(1), 1982.

[Rama89] Ramamritham, K., and Stankovic, J., "Overview of the SPRING Project," *IEEE Real-Time Systems Newsletter*, Winter 1989.

[Robi82] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems," *Ph.D. Thesis*, Computer Sciences Dept., Carnegie Mellon University, 1982.

[Sha87] Sha, L., Rajkumar, R., Lehoczky, J., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *Technical Report CMU-CS-87-181*, Depts. of CS, ECE, and Statistics, Carnegie Mellon University, 1987.

[Stan88] Stankovic, J., Zhao, W., "On Real-Time Transactions," *ACM SIGMOD Record*, 17(1), March 1988.

[Tay84] Tay, Y., "A Mean Value Performance Model for Locking in Databases," *Ph.D. Thesis*, Computer Sciences Dept., Harvard University, February 1984.

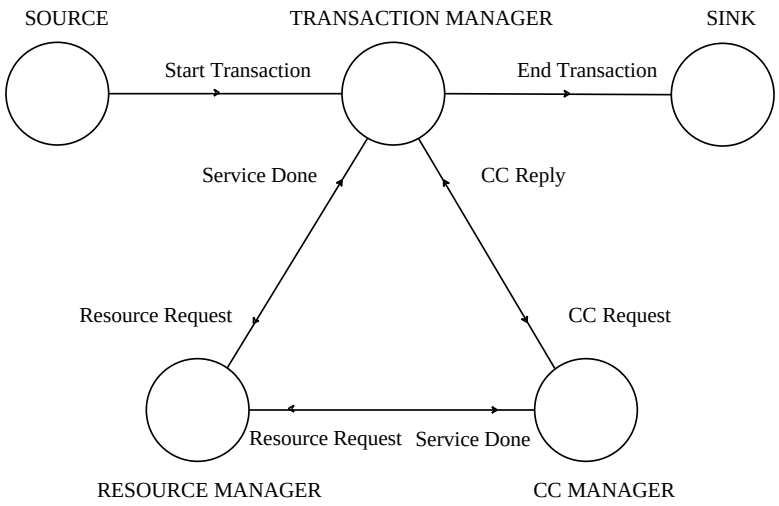

Figure 1: RTDBS Model Structure

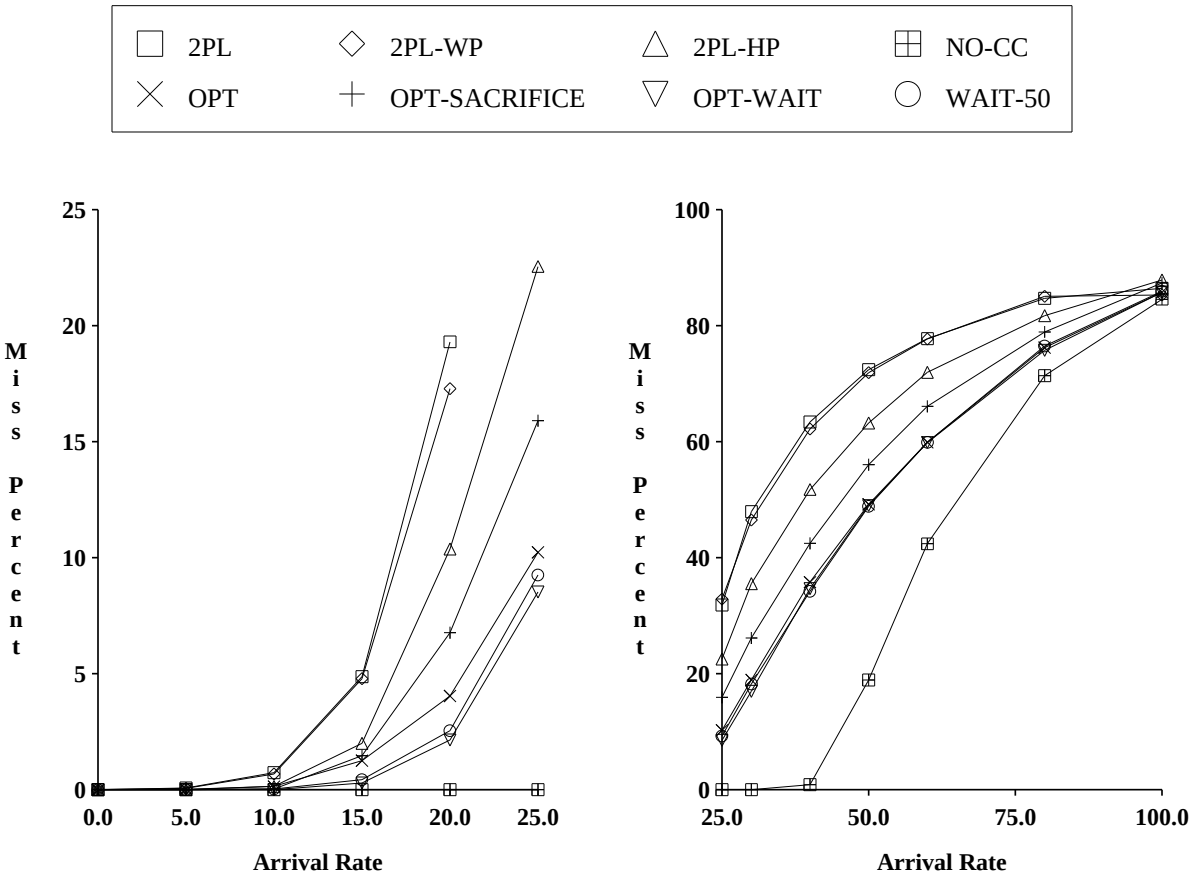

Figure 2a: Baseline (Normal Load)

Figure 2b: Baseline (Heavy Load)

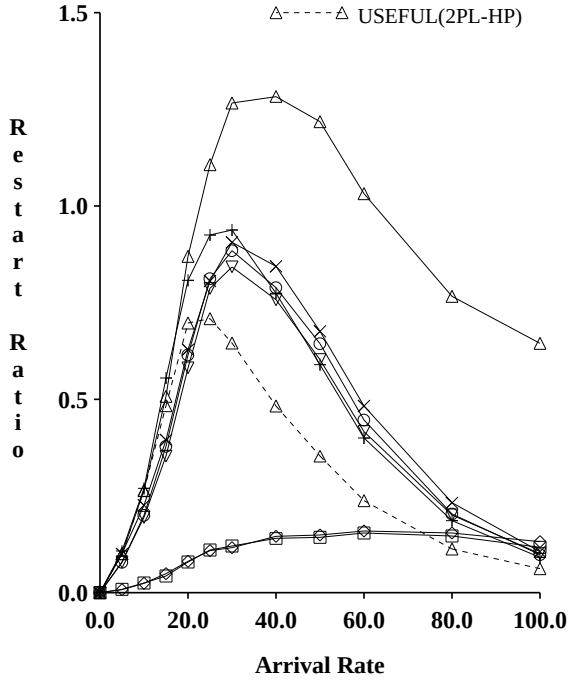

Figure 2c: Restarts (Baseline)

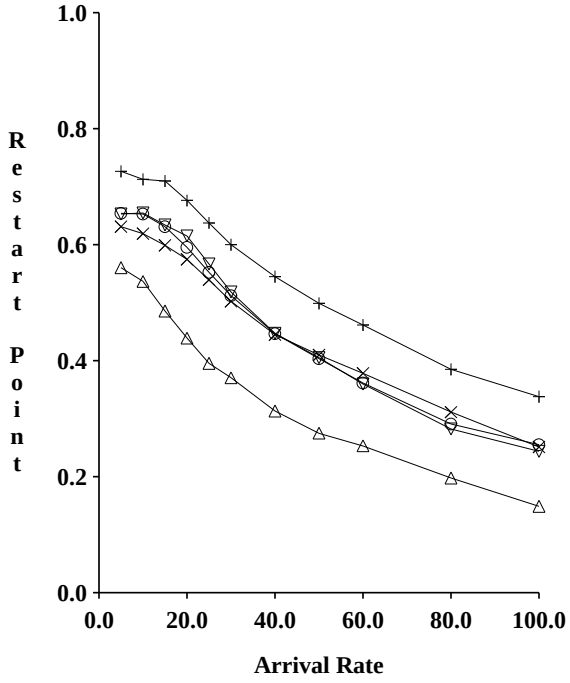

Figure 2d: Restart Point (Baseline)

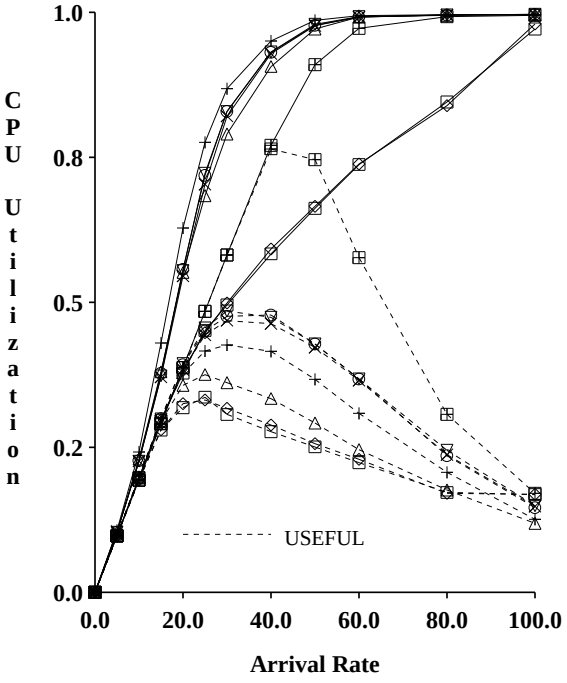

Figure 2e: CPU Utilization (Baseline)

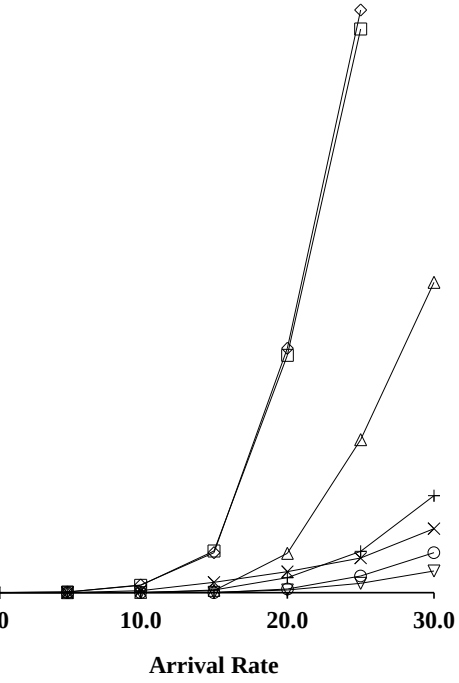

Figure 3a: Data Contention (Normal Load)

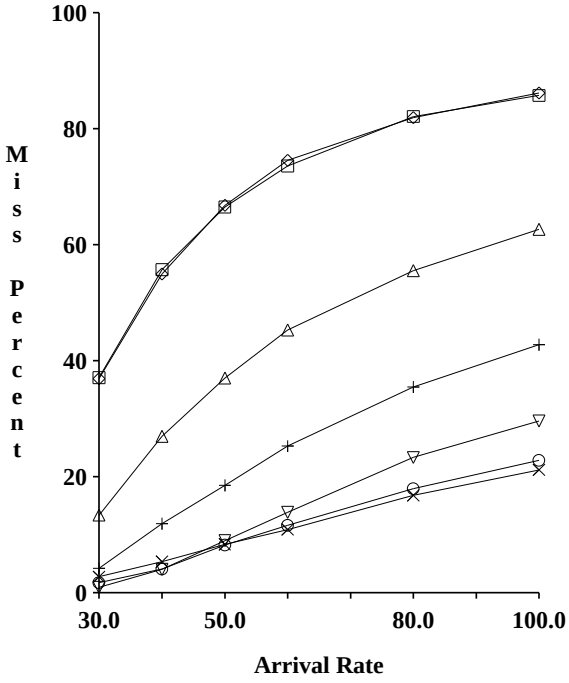

Figure 3b: Data Contention (Heavy Load)

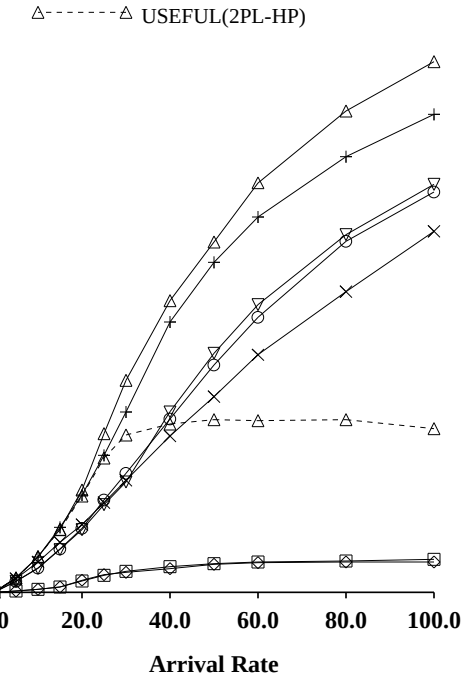

Figure 3c: Restarts (Data Contention)

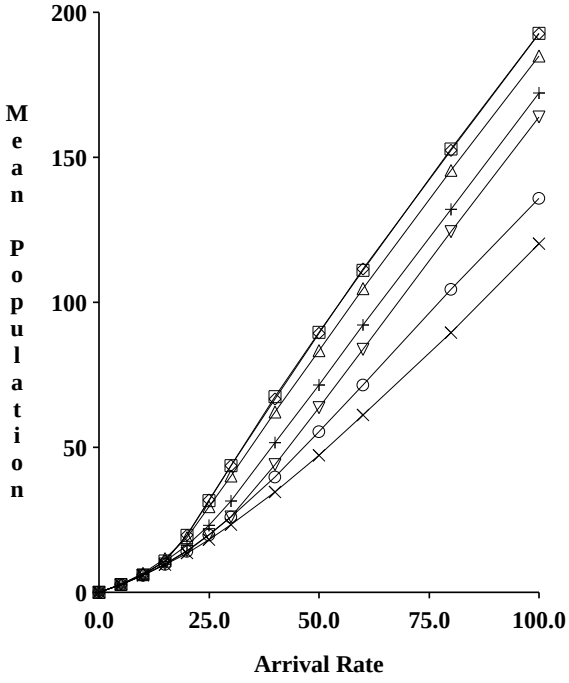

Figure 3d: Population (Data Contention)

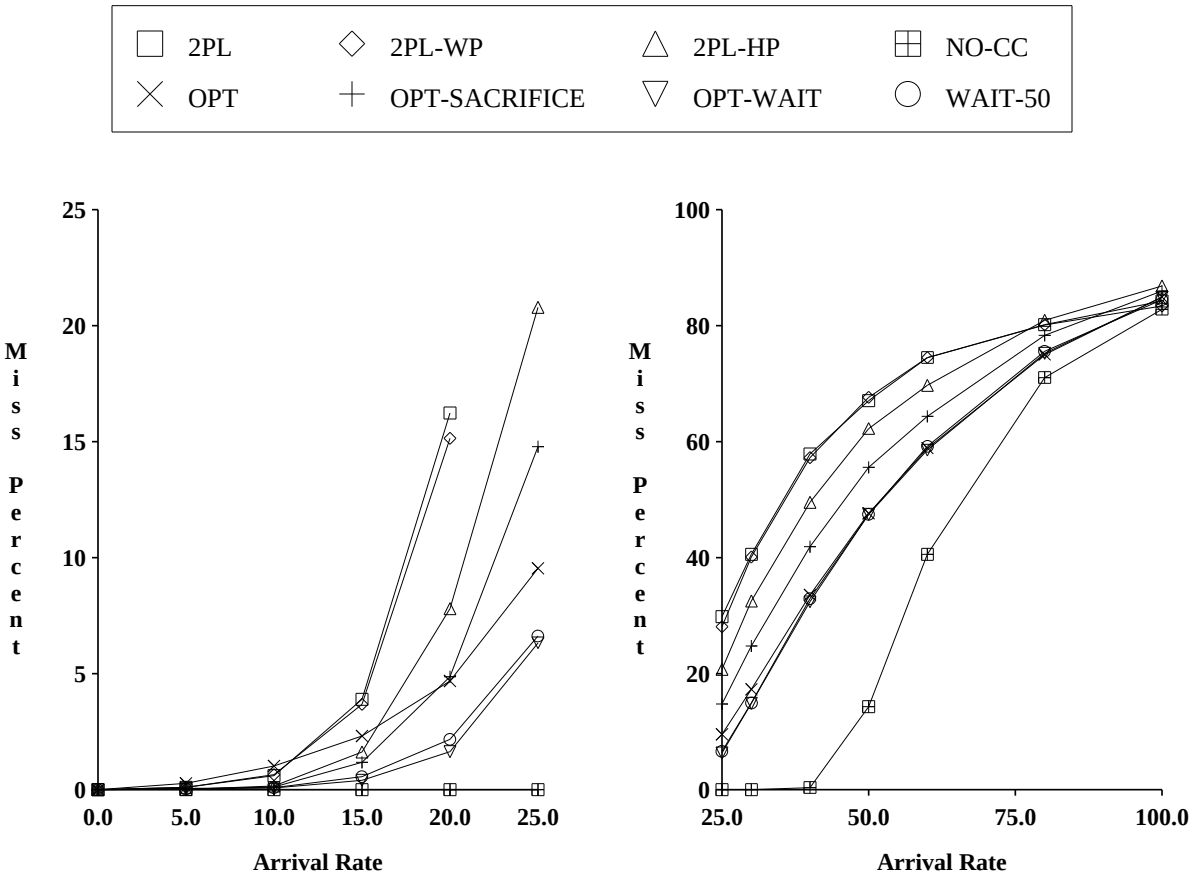

Figure 4a: Variable Slack Ratio (Normal Load)

Figure 4b: Variable Slack Ratio (Heavy Load)

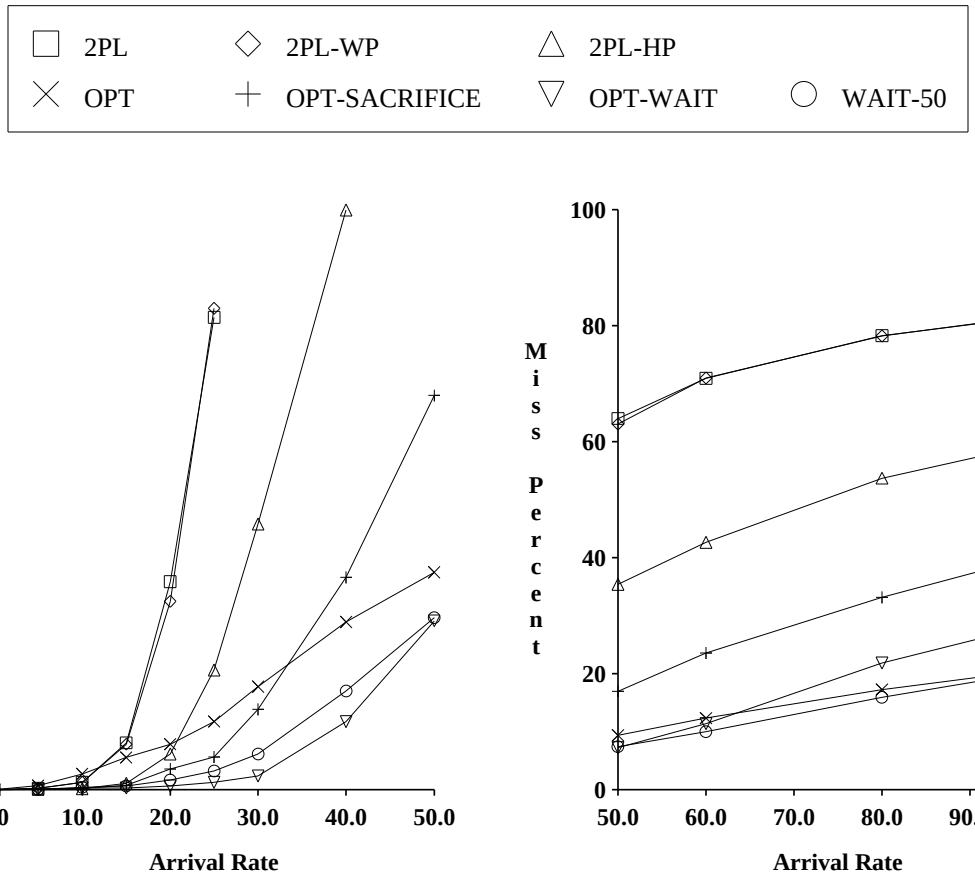

Figure 5a: Data Contention (Normal Load)

Figure 5b: Data Contention (Heavy Load)

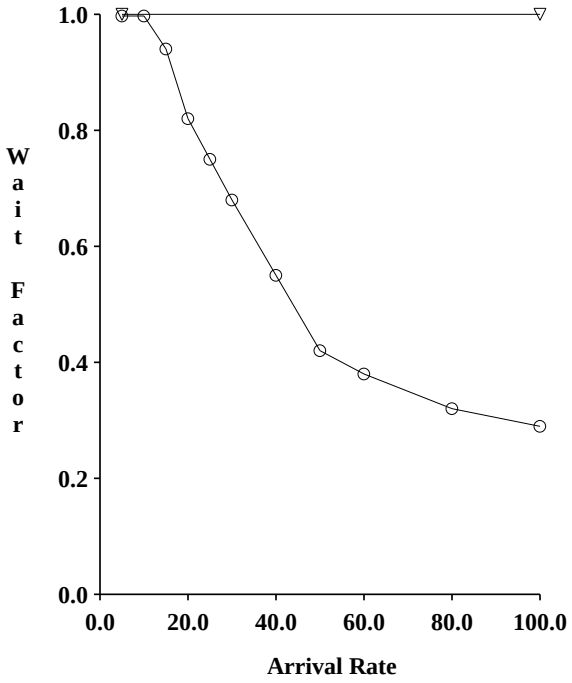

Figure 5c: Wait Factor (Data Contention)

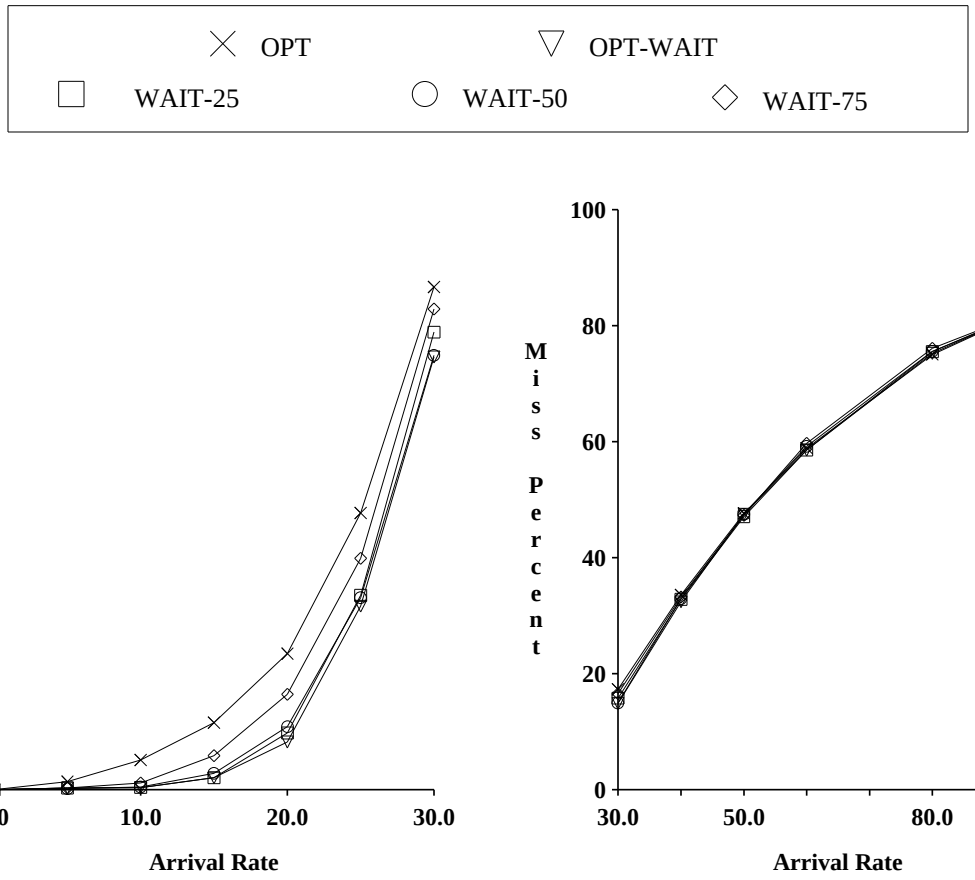

Figure 6a: Wait Control (Normal Load)

Figure 6b: Wait Control (Heavy Load)

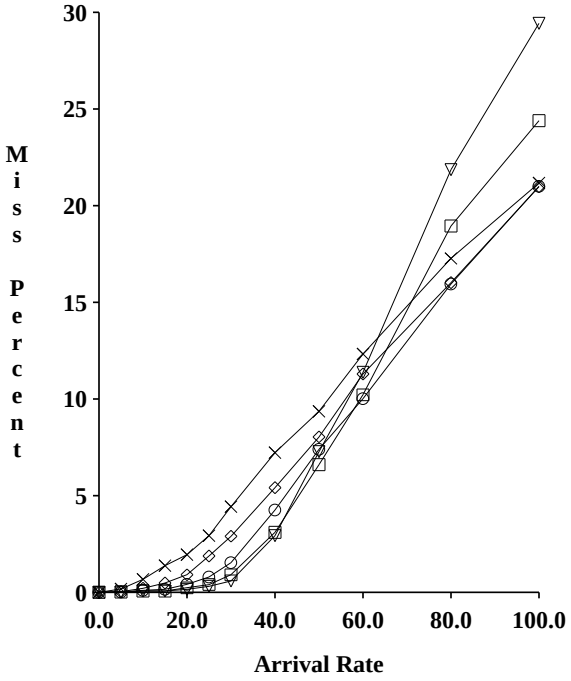

Figure 7: Data Contention

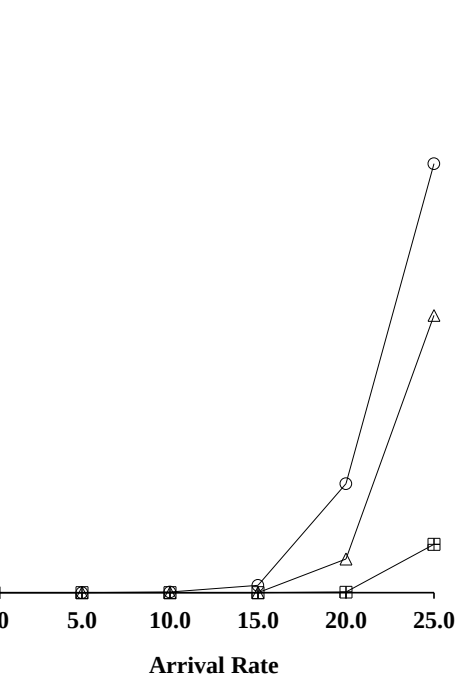

Figure 8a: Large Database (Normal Load)

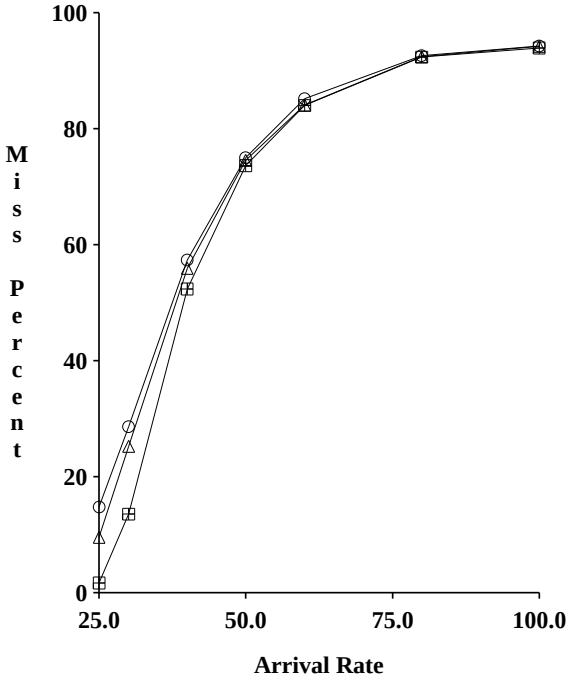

Figure 8b: Large Database (Heavy Load)

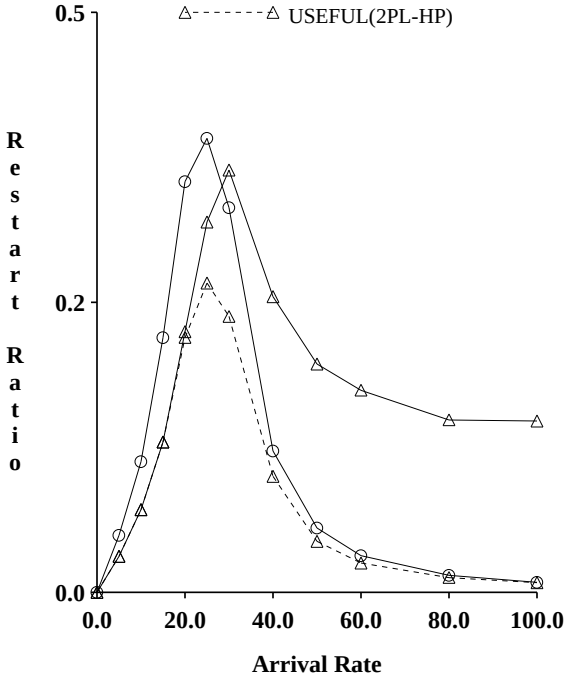

Figure 8c: Restarts

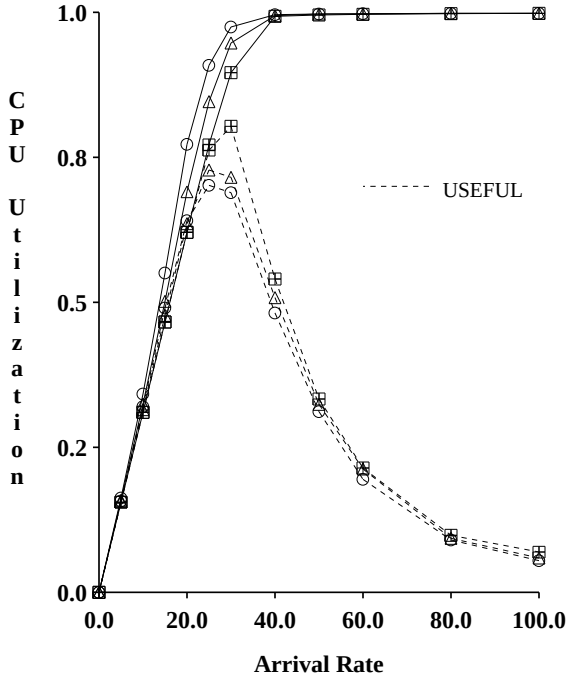

Figure 8d: CPU Utilization

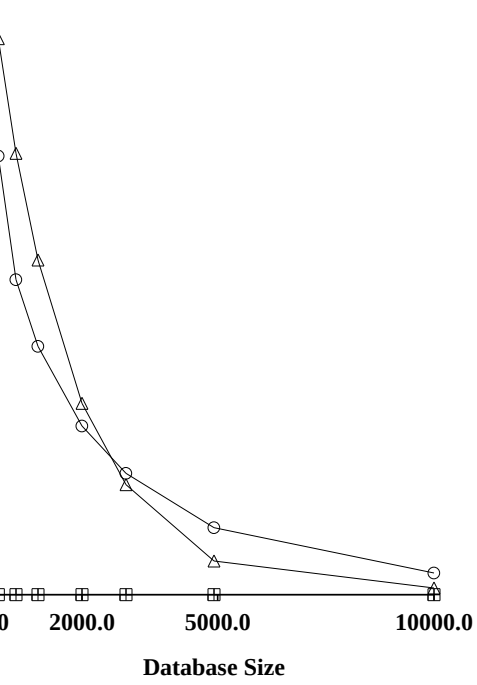

Figure 9a: Crossover (Arr. Rate = 20)

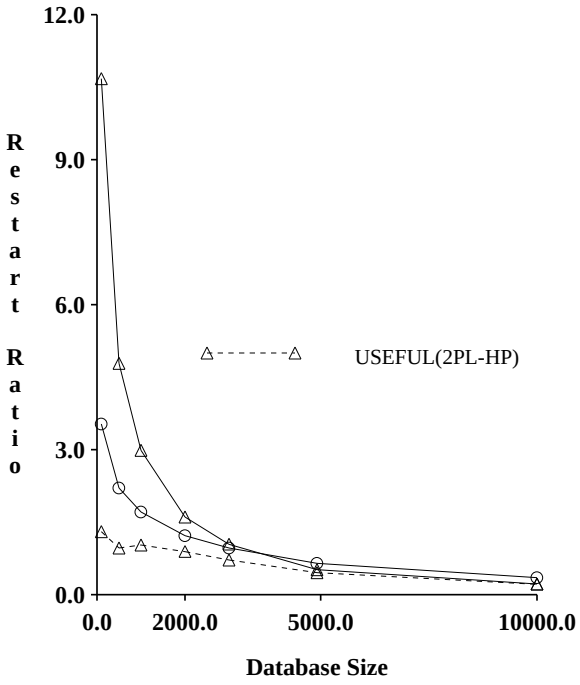

Figure 9b: Restarts (Crossover)