Dynamic Real-Time Optimistic Concurrency Control

Jayant R. Haritsa   Michael J. Carey   Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

In a recent study, we have shown that in real-time database systems that discard late transactions, optimistic concurrency control outperforms locking. Although the optimistic algorithm used in that study, OPT-BC, did not factor in transaction deadlines in making data conflict resolution decisions, it still outperformed a deadline-cognizant locking algorithm. In this paper, we discuss why adding deadline information to optimistic algorithms is a non-trivial problem, and describe some alternative methods of doing so. We present a new real-time optimistic concurrency control algorithm, WAIT-50, that monitors transaction conflict states and gives precedence to urgent transactions in a controlled manner. WAIT-50 is shown to provide significant performance gains over OPT-BC under a variety of operating conditions and workloads.

1. INTRODUCTION

A Real-Time Database System (RTDBS) is a transaction processing system that 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. Accordingly, greater value is associated with processing transactions before their deadlines as compared to completing them late. Therefore, in contrast to a conventional DBMS where the goal usually is to minimize response times, the emphasis here is on satisfying the timing constraints of transactions.

The problem of scheduling transactions in an RTDBS with the objective of minimizing the percentage of late transactions was first addressed in [Abbo88, Abbo89]. Their work focused on evaluating the performance of various real-time scheduling policies. All these policies enforced data consistency by using a two-phase locking protocol as the underlying concurrency control mechanism. Performance studies of concurrency control methods for conventional DBMSs (e.g.[Agra87]) have concluded that locking protocols, due to their conservation of resources, perform better than optimistic techniques when resources are limited. In a recent study [Hari90a], we investigated the behavior of these concurrency control schemes in a real-time environment. The study showed that for *firm deadline* real-time database systems, where late transactions are immediately discarded, optimistic concurrency control outperforms locking over a wide range of system loading and resource availability. The key reason for this surprising result is that the optimistic approach, due to its validation stage conflict resolution, ensures that eventually discarded transactions do not restart other transactions. The locking approach, on the other hand, allows these soon-to-be-discarded transactions to cause other transactions to be either blocked or restarted due to lock conflicts, thereby increasing the number of late transactions.

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

An important difference between the locking algorithm and the optimistic algorithm that were compared in the above study lies in their use of transaction deadline information. The locking algorithm used this information, which was encoded in the form of transaction priorities, to provide preferential treatment to urgent transactions. The optimistic algorithm, however, was just the conventional broadcast commit optimistic scheme [Mena82, Robi82], and ignored transaction priorities in resolving data contention. The study therefore concluded that, in the firm real-time domain, a "vanilla" optimistic algorithm can perform better than a locking algorithm that is "tuned" to the real-time environment. The following question then naturally arises: How can we use priority information to improve the performance of the optimistic algorithm and thus further decrease the number of late transactions?

A simple answer to this question would be to use priority information in the resolution of data conflicts, that is, to resolve data conflicts always in favor of the higher priority transaction. This solution, however, has two problems: First, giving preferential treatment to high priority transactions may result in an increase in the number of missed deadlines. This can happen, for example, if helping a high priority transaction to make its deadline causes several lesser priority transactions to miss their deadlines. Second, if fluctuations can occur in transaction priorities, repeated conflicts between a pair of transactions may be resolved in some cases in favor of one transaction and in other cases in favor of the other transaction. This would hinder the progress of both transactions and hence degrade performance. Therefore, a priority-cognizant optimistic algorithm must address these two problems in order to perform better than a simple optimistic scheme.

In this paper, we report on our efforts to develop such an algorithm, and present a new optimistic concurrency control algorithm, called **WAIT-50**. The algorithm incorporates a *priority wait* mechanism that makes low priority transactions wait for conflicting high priority transactions to complete, thus enforcing preferential treatment for high priority transactions.

To address the first problem raised above, WAIT-50 features a *wait control* mechanism. This mechanism monitors transaction conflict states and, with a simple "50 percent" rule, dynamically controls when and for how long a transaction is made to wait. The second problem is handled by having the priority wait mechanism resolve conflicts in a manner that results in the commit of at least one of the conflicting transactions. Simulation results show that WAIT-50 performs significantly better than **OPT-BC**, the optimistic algorithm used in our earlier study.

The remainder of this paper is organized in the following fashion: Section 2 reviews our earlier study. In Section 3, we discuss deficiencies of OPT-BC. The new optimistic algorithm, WAIT-50, is presented in Section 4. Then, in Section 5, we describe our RTDBS model and its parameters, while Section 6 highlights the results of the simulation experiments. Finally, Section 7 summarizes the main conclusions of the study.

2. BACKGROUND

Our earlier study [Hari90a] investigated the relative performance of locking protocols and optimistic techniques in an RTDBS environment. In particular, the performance of a locking protocol, 2PL-HP, was compared with that of an optimistic technique, OPT-BC. These particular instances were chosen because they are of comparable complexity and are general in their applicability since they make no assumptions about knowledge of transaction semantics or resource demands. The details of these algorithms are explained below.

In 2PL-HP, classical two phase locking [Eswa76] is augmented with a *High Priority* [Abbo88] conflict resolution scheme to ensure that high priority transactions are not delayed by low priority transactions. This scheme resolves all data conflicts 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 holders, the holders are restarted and the requester is granted the lock; otherwise, the requester waits for the lock holders to release the object. The High Priority scheme also serves as a deadlock prevention mechanism.[1]

In OPT-BC, classical optimistic concurrency control [Kung81] is modified to implement the notion of a *Broadcast Commit* [Mena82, Robi82]. Here, when a transaction commits, it notifies other running transactions that conflict with it and these transactions are immediately restarted. Since there is no need to check for conflicts with already committed transactions, a transaction which has reached the validation stage is guaranteed to commit. The broadcast commit method detects conflicts earlier than the basic optimistic algorithm, resulting in less wasted resources and earlier restarts; this increases the chances of meeting transaction deadlines. An important point to note is that transaction priorities are *not* used in resolving data conflicts.

[1] This is true only for priority assignment schemes that do not change a transaction's priority during the course of its execution.

The results of our study showed that both the policy for dealing with late transactions and the availability of resources have a significant impact on the relative behavior of the algorithms. In particular, for a *firm deadline* system, where late transactions are discarded without being run to completion, OPT-BC outperformed 2PL-HP over a wide range of system loading and resource availability. Figures 1 and 2 present sample graphs of how the percentage of late transactions varies as a function of the transaction arrival rate. These graphs were derived for the baseline model of the study, which characterized an RTDBS system with high data contention, under conditions of limited resources and plentiful resources, respectively.

In the above scenario, 2PL-HP suffered from two major problems: *wasted restarts* and *mutual restarts*. A "wasted restart" occurs when an executing transaction is restarted by another transaction that later misses its deadline. Such restarts are useless and cause performance degradation. In OPT-BC, however, we are guaranteed the commit of any transaction that

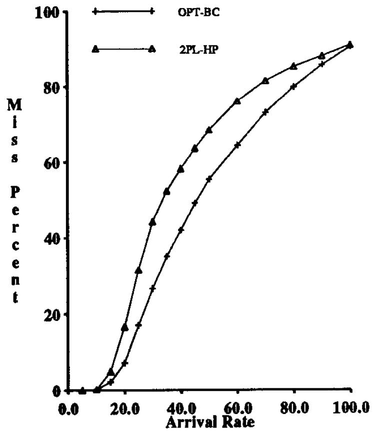

Figure 1: Baseline Model (Limited Resources)

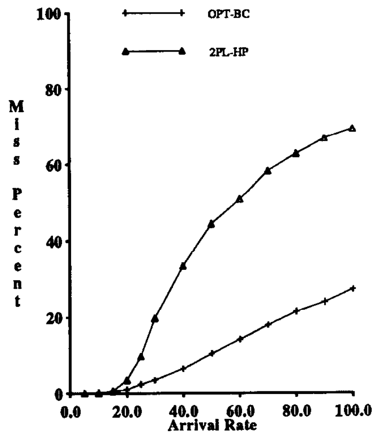

Figure 2: Baseline Model (Plentiful Resources)

reaches the validation stage. Since only validating transactions can cause restarts of other transactions, *all* restarts generated by the OPT-BC algorithm are useful.

The problem of "mutual restarts" arises when fluctuations occur in transaction priority profiles. For certain types of dynamic transaction priority assignment schemes (e.g. Least Slack [Jens86]), 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".[2] For algorithms like 2PL-HP, which use transaction priorities to resolve data conflicts, priority reversals may lead to "mutual restarts" – a pair of transactions restart each other, thus hindering the progress of both transactions. Since OPT-BC does not use transaction priorities in resolving data contention, such problems simply *do not arise*.

3. PROBLEMS WITH OPT-BC

In this section, we will motivate why there is room for improvement on the OPT-BC algorithm. The validation algorithm of OPT-BC can be succinctly written as:

```
restart all conflicting transactions;
commit the validating transaction;
```

Although this algorithm provides immunity from priority dynamics due to its unilateral commit, it does not allow for using transaction priorities to further decrease the number of missed deadlines. To illustrate this problem, consider the scenario in Figure 3, where the execution profile of two concurrently executing transactions, X and Y, is shown. X has an arrival time A_X and deadline D_X, and Y has an arrival time A_Y and deadline D_Y. Also, assume that transaction X, by virtue of its earlier deadline, has a higher priority than transaction Y. Now, consider the situation where at time $t = V_Y$, when transaction X is close to completion, transaction Y reaches its validation point and detects a conflict with X. Under the OPT-BC algorithm, Y would immediately commit and in the process restart X. Restarting X at this late stage guarantees that it has no chance of meeting its deadline.

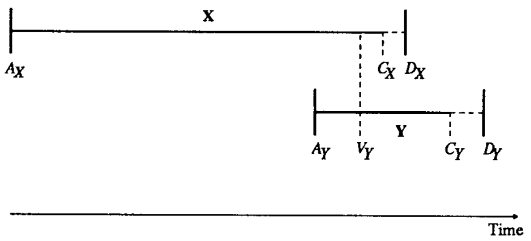

Figure 3: Poor OPT-BC data conflict decision

[2] 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.

If a priority-cognizant algorithm had been used instead, it would have recognized that X's priority was higher than that of Y. Then, in some fashion, it would have *prevented Y from committing* until X had completed. With this decision, we could possibly gain the completion of both transactions X and Y before their deadlines, as shown in Figure 3 where X completes at time $t = C_X$ and Y completes later at time $t = C_Y$.

The above example shows how OPT-BC's indifference to transaction priorities can degrade performance. Another drawback of OPT-BC is that it has an inherent bias against long transactions, just like the classical optimistic algorithm. The use of priority information in resolving conflicts can help counter this bias.

4. PRIORITY-COGNIZANT ALGORITHMS

As explained in the previous sections, although the OPT-BC algorithm highlights some major strengths of optimistic concurrency control in real-time database systems, there remains potential for improving its performance. We therefore tried to develop new optimistic algorithms that address the problems of OPT-BC without sacrificing the performance-beneficial aspects of the broadcast commit scheme. These algorithms are described in this section. In the subsequent discussion, we will use the term *conflict set* to denote the set of currently running transactions that conflict with a validating transaction. The acronym *CHP (Conflicting Higher Priority)* will be used to refer to transactions that are in the conflict set and have a higher priority than the validating transaction. Similarly, the acronym *CLP (Conflicting Lower Priority)* will be used to refer to transactions that are in the conflict set and have a lower priority than the validating transaction. In this section, our aim is to motivate the development of the algorithms and discuss, at an intuitive level, their potential strengths and weaknesses.

The example in Section 3, illustrating poor conflict decisions by OPT-BC, showed that we need a scheme to prevent low priority transactions that conflict with higher priority transactions from unilaterally committing. The following two options are available:

(1) *Restart:* The low priority transaction is restarted.

(2) *Block:* The low priority transaction is blocked.

Two algorithms, OPT-SACRIFICE and OPT-WAIT, were developed based on these options. WAIT-50 was then developed as an extension of the OPT-WAIT algorithm. These three algorithms are presented below.

4.1. OPT-SACRIFICE

In this algorithm, when a transaction reaches its validation stage, it checks for conflicts with currently executing transactions. If conflicts are detected and at least one of the transactions in the conflict set is a CHP transaction, then the validating transaction is restarted – that is, it is *sacrificed* in an effort to help the higher priority 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;
```

Referring back to Figure 3, if we were using OPT-SACRIFICE, then at time $t = V_Y$, transaction Y would restart itself due to the conflict with the higher priority transaction X.

OPT-SACRIFICE is priority-cognizant and satisfies the goal of giving preferential treatment to high priority transactions. It suffers, however, from two potential problems. First, there is the 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. Second, the algorithm does not have immunity to priority dynamics. For example, the situation may arise where transaction A is sacrificed for transaction B because B's priority is currently greater than that of A, and transaction B at a later time is sacrificed for transaction A because A's priority is now greater than that of B. Therefore, priority reversals may lead to *mutual sacrifices*. These two drawbacks are analogous to the "wasted restarts" and "mutual restarts" problems of 2PL-HP.

4.2. OPT-WAIT

This algorithm incorporates a *priority wait* mechanism: 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;
```

Referring back to Figure 3, if we were using OPT-WAIT, then at time $t = V_Y$, transaction Y would wait, without committing, for transaction X to complete first. Of course, X's completion may cause Y to be restarted.

There are several reasons that suggest that the priority wait mechanism may have a positive impact on performance, and these are outlined below:

(1) In keeping with the original goal, precedence is given to high-priority transactions.

(2) The problem of "wasted sacrifices" does not exist because if a CHP transaction is discarded due to missing its deadline, or is restarted by some other transaction, then the waiter is immediately "taken off the shelf" and committed if no other CHP transactions remain.

(3) Priority reversals are not a problem because, if a CHP transaction being waited for were to become a CLP transaction, the waiting transaction will no longer wait for it, and will immediately commit if no other CHP transactions remain.

(4) 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].

(5) The fact that a CHP transaction commits does not necessarily imply that the waiting transaction has to be restarted (!).

The last point requires further explanation: The key observation here is that if transaction A conflicts with transaction B, it does not necessarily mean that the converse is true [Robi82]. This is explained as follows: Under the broadcast commit scheme, a validating transaction A is said to conflict with another transaction B if and only if

$$WriteSet_A \cap ReadSet_B \neq \phi \qquad (1)$$

We will denote such a conflict from transaction A to B by $A \rightarrow B$. For transaction B to also conflict with transaction A, i.e. for $B \rightarrow A$, it is necessary that

$$WriteSet_B \cap ReadSet_A \neq \phi \qquad (2)$$

As is obvious from Equations (1) and (2), $A \rightarrow B$ does not imply $B \rightarrow A$. Therefore, if in fact $B \rightarrow A$ is not true, then by committing the transactions in the order (B,A) instead of the order (A,B), both transactions can be committed without restarting either one.

As per the explanation given above, it is possible with our waiting scheme for the CHP transaction and the waiting transaction to commit in that order without either transaction being restarted. Therefore, the priority wait mechanism has a potential to actually *eliminate* some data conflicts. (A simple probabilistic analysis of the extent to which waiting can reduce data conflicts is presented in [Hari90b]).

Although the waiting scheme has many positive features, it is not an unmixed blessing. 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. This decreases the chances of these transactions meeting their deadlines, and also wastes 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 view 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 can simultaneously increase the probability of having a larger *number* of conflicts per transaction. This increase may be substantial when there are many concurrently executing transactions in the system.

4.3. WAIT-50

The WAIT-50 algorithm is an extension of OPT-WAIT – in addition to the priority wait mechanism, it incorporates a *wait control* mechanism. This mechanism monitors transaction conflict states and dynamically decides when, and for how long, a low priority transaction should be made to wait for its CHP transactions. 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 WAIT-50, a simple "50 percent" rule is used – a validating transaction is made to wait only while HPpercent ≥ 50, that is, while half or more of its conflict set is composed of higher priority transactions. 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 high priority transactions and decreasing pairwise conflicts, are outweighed by its drawbacks, in terms of later restarts and an increased number of conflicts. Therefore, while OPT-WAIT and OPT-BC represent the extremes with regard to waiting – OPT-WAIT always waits for a CHP transaction, and OPT-BC never waits – WAIT-50 is a ***hybrid*** algorithm that controls the amount of waiting based on transaction conflict states. In fact, we can view OPT-WAIT, WAIT-50, and OPT-BC as all being special cases of a general algorithm **WAIT-X**, where X is the cutoff HPpercent level, with X taking on the values 0, 50, and ∞, respectively, for these algorithms.

We conducted experiments to evaluate the performance of the various optimistic algorithms, and the following sections describe our experimental framework and results.

5. REAL-TIME DBMS MODEL

The real-time database system model employed here is the same as that of our earlier study – in this model, the system consists of a shared-memory multiprocessor DBMS operating on disk resident data.[3] The database itself is modeled as a collection of pages. Transactions arrive in a Poisson stream and each transaction has an associated deadline. Each transaction consists of a sequence of page 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 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 pages that they access and the number of pages that they update. Table 1 summarizes the key workload parameters. *ArrivalRate* specifies the rate of transaction arrivals. *DatabaseSize* gives the number of pages in the database. The number of pages accessed by a transaction varies uniformly between half and one-and-a-half times the value of *PageCount*. Page requests are generated from a uniform distribution spanning the entire database. *WriteProb* gives the probability that a page that is read will also be updated.

We use two transaction deadline assignment formulas in this study. The first formula, which is the same as the one used in our previous study, is:

$$D_T = A_T + SF * R_T \qquad \text{(DF1)}$$

where D_T, A_T, and R_T are the deadline, arrival time and resource time, respectively, of transaction *T*, while *SF* is a slack factor. The ***resource time*** is the total service time at the resources that the transaction requires for its data processing. The *slack factor* is a constant that provides control over the tightness/slackness of deadlines. The formula ensures that all transactions, independent of their service requirement, have the same *slack ratio* – this is defined to be the ratio $(D_T - A_T) / R_T$. Therefore, all transactions have *SF* as their slack ratio.

In order to evaluate the effects of variability in transaction slack ratios, a second deadline assignment formula is used in the present study. This formula is:

$$D_T = \begin{cases} A_T + LSF * R_T \\ A_T + HSF * R_T \end{cases} \qquad \text{(DF2)}$$

With this formula, transactions will have a slack factor of either *LSF* or *HSF*, with both choices being equally likely. Therefore, the slack ratio for a transaction will be either *LSF* or *HSF*. The *LSF* and *HSF* workload parameters set the slack factors to be used in the deadline formulas. (For DF1, these two parameters have the same value).

The transaction priority assignment scheme used in all the experiments reported here is *Earliest Deadline* – transactions with earlier deadlines have higher priority than transactions with later deadlines. The system operates under firm deadlines, and therefore discards late transactions. It is important to note that while the workload generator uses transaction resource requirements in assigning deadlines, we assume that the system itself lacks any knowledge of these requirements. This implies that a transaction is detected as being late only when it actually misses its deadline.

Table 1: Workload Model Parameters

Parameter	Meaning
ArrivalRate	Transaction arrival rate
DatabaseSize	Number of pages in database
PageCount	Avg. # pages accessed/transaction
WriteProb	Write probability/accessed page
DeadlineFormula	DF1 or DF2
LSF	Low Slack Factor
HSF	High Slack Factor

[3] It is assumed, for simplicity, that all data is accessed from disk and buffer pool considerations are therefore ignored.

5.2. Resource 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 according to a priority-based variant of the elevator disk scheduling algorithm [Care89]. Requests at each disk are grouped into priority levels and the elevator algorithm is applied within each priority level; requests at a priority level are served only when there are no pending requests at higher priority levels. The details of our implementation of this algorithm are described in [Hari90b]. The data pages are modeled as being uniformly distributed across all the disks and across all tracks within a disk.

6. EXPERIMENTS and RESULTS

In this section, we present performance results for our simulation experiments comparing the various optimistic algorithms in a real-time database system environment. Our experiments evaluated the algorithms under a variety of operating conditions, workloads, and data access patterns [Hari90b]. We present only a subset of the results here due to space limitations. The performance metric 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.[4] The simulations also generated a host of other statistical information, such as the number of data conflicts, the time spent in priority waiting, etc. These secondary measures help explain the behavior of the algorithms under various loading levels. The resource parameter settings are such that the CPU time to process a page is 10 milliseconds while disk access times are between 15 and 30 milliseconds, depending on the level of disk utilization. Disk access times depend on disk utilization due to the elevator scheduling policy.

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 intended to isolate the effect of data 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. 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 usually the driving consideration.

[4] 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.

We began our experiments by evaluating the various optimistic algorithms for the baseline model of our earlier study. This was done to provide continuity from that study to the present work. Subsequently, for reasons explained in the following discussion, we moved to a new baseline model. After initial experiments with this model, further experiments were constructed around it by varying a few parameters at a time. These experiments evaluated the impact of data contention, resource contention, deadline slack variation, transaction write probabilities, and the wait control mechanism parameter. We will hereafter refer to the old baseline model as FIX-SR (Fixed Slack Ratio), and the new baseline model as VAR-SR (Variable Slack Ratio).

6.1. FIX-SR Baseline Model

The settings of the workload parameters and resource parameters for the FIX-SR baseline model are listed in Tables 2 and 3. These settings generate an appreciable level of both data contention and resource contention. For this model, Figures 4a and 4b show MissPercent behavior under normal load and heavy load, respectively. When the same experiment is carried out under infinite resources, Figures 5a and 5b are obtained. From this set of graphs, we can make the following observations:

(1) OPT-SACRIFICE performs significantly worse than the wait-based algorithms over the entire operating region, and for the most part, also performs worse than OPT-BC. The poor performance of this algorithm is primarily due to the problem of "wasted sacrifices", discussed in Section 4. Also, in the infinite resource case, the sacrifice policy generates a steep rise in the number of data conflicts by causing a significant increase in the average number of transactions in the system. This is brought out quantitatively in Figure 5c, which plots the average number of conflicts per input transaction.

(2) OPT-WAIT, due to its priority cognizance, performs very well at low levels of data contention (Figs.4a, 5a). As data contention increases, however, its performance

Table 2: FIX-SR Baseline Model Workload Settings

Parameter	Value
DatabaseSize	1000 pages
PageCount	16 pages
WriteProb	0.25
DeadlineFormula	DF1
$SlackFactor_l$	4.0
$SlackFactor_h$	4.0

Table 3: FIX-SR Baseline Model Resource Settings

Parameter	Value
NumCPUs	10
NumDisks	20

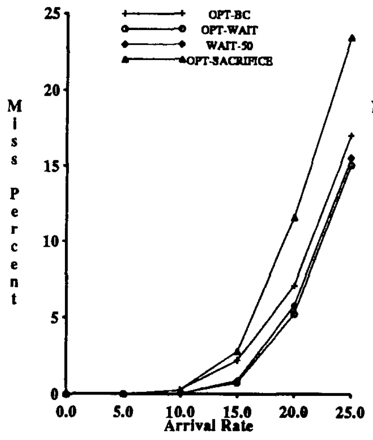

Fig. 4a: FIX-SR (Normal Load)

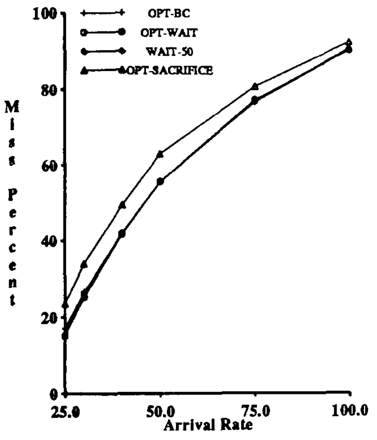

Fig. 4b: FIX-SR (Heavy Load)

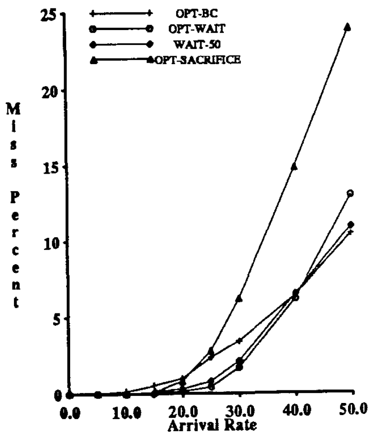

Fig. 5a: Inf. Res. (Normal Load)

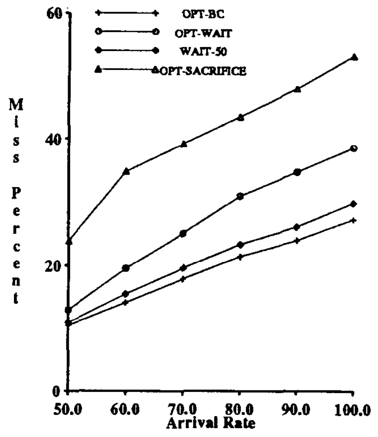

Fig. 5b: Inf. Res. (Heavy Load)

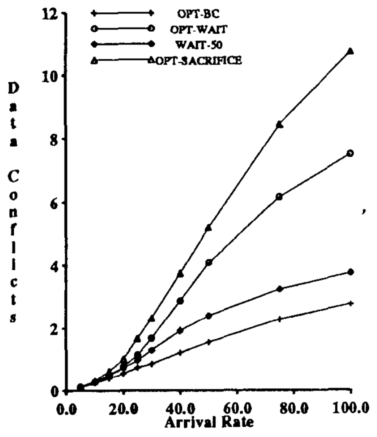

Fig. 5c: Conflicts (Inf. Res.)

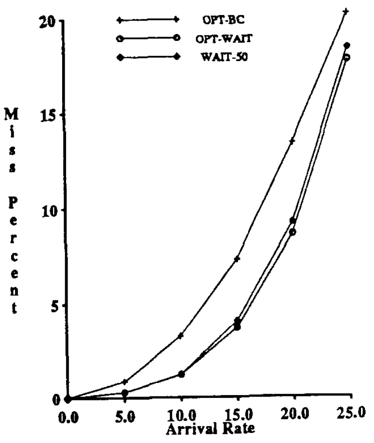

Fig. 6a: VAR-SR (Normal Load)

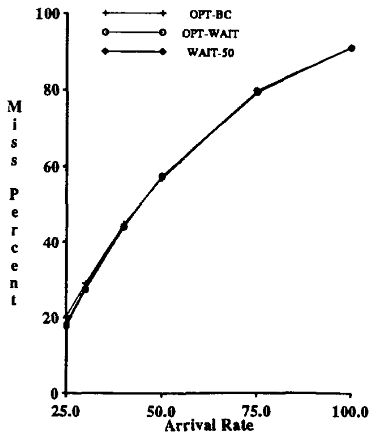

Fig. 6b: VAR-SR (Heavy Load)

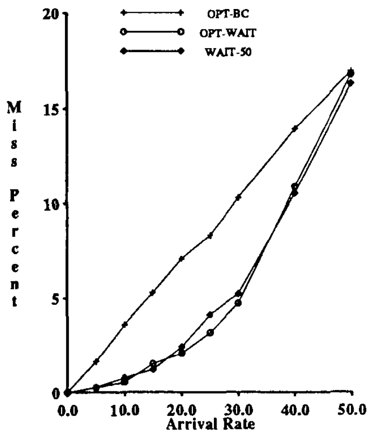

Fig. 7a: Inf. Res. (Normal Load)

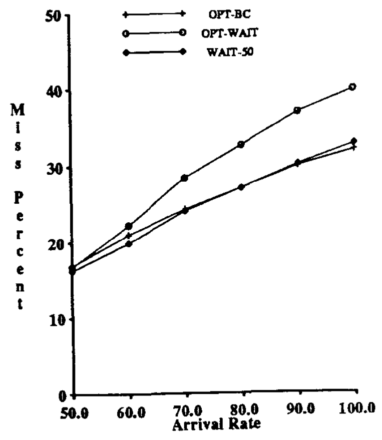

Fig. 7b: Inf. Res. (Heavy Load)

steadily degrades. Finally, at high contention levels under infinite resources (Fig.5b), it performs significantly worse than OPT-BC. The reason for OPT-WAIT's poor performance in this region is that its priority wait mechanism, just like the sacrifice policy, causes an increase in the average number of transactions in the system. This population increase generates a corresponding rise in the number of data conflicts (see Fig. 5c), resulting in higher miss percentages.

(3) WAIT-50 provides the *best overall* performance. At low data contention levels, it behaves like OPT-WAIT, and at high contention levels it behaves like OPT-BC. The explanation for this behavior is given in the next section.

(4) Under high resource contention (Fig.4b), WAIT-50 and OPT-WAIT behave identically to OPT-BC. This is because, with heavy resource contention, it is uncommon for a low priority transaction to reach its validation stage much before its deadline, and therefore the wait-times of transactions are mostly small. Accordingly, the priority wait mechanism has very limited impact, and WAIT-50, OPT-WAIT, and OPT-BC become essentially the same algorithm.

The above results are encouraging because they show that there are performance benefits to be gained by using priority-cognizant algorithms. It is all the more encouraging that these performance improvements are obtained despite all transactions having the same slack ratio (from using deadline formula DF1). A fixed transaction slack ratio reduces the likelihood of a validating transaction finding a higher priority transaction in its set of conflicting transactions. This creates favorable circumstances for OPT-BC since the detrimental effects of its priority insensitivity are reduced.

6.2. VAR-SR Baseline Model

In order to generate a workload with variation in transaction slack ratios, the VAR-SR baseline model was developed for the current study. This model uses deadline assignment formula DF2 to generate variation in transaction slack ratios. The workload parameters *LSF* and *HSF* are set at 2.0 and 6.0, respectively.[5] The remaining workload parameter settings and resource parameter settings are the same as those for the FIX-SR baseline model (see Tables 2 and 3). In the subsequent discussions, we will compare the performance of only the OPT-BC, OPT-WAIT and WAIT-50 algorithms since OPT-SACRIFICE invariably performed worse than the wait-based algorithms.

For the VAR-SR baseline model, Figures 6a and 6b show the behavior of the algorithms under normal load and heavy load, respectively. When the same experiment was carried out under infinite resources, Figures 7a and 7b were obtained. From this set of graphs we can make the following observations:

[5] These parameter selections ensure that the *mean* slack ratio is the same as that of the FIX-SR baseline model, namely 4.0.

(1) The priority-cognizant algorithms, WAIT-50 and OPT-WAIT, now perform *significantly better* than OPT-BC under normal loads.

(2) WAIT-50 again turns in the best overall performance by behaving like OPT-WAIT at low data contention levels and like OPT-BC at high data contention levels.

As can be seen from this experiment, and will be further confirmed in subsequent experiments, WAIT-50 provides performance close to either OPT-BC 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 effectively integrates priority and waiting* into the optimistic concurrency control framework. The control mechanism is clearly quite competent at deciding when the benefits of waiting, in terms of helping high priority transactions to make their deadlines, are outweighed by the drawbacks of causing an increased number of conflicts. In Figure 7c, we plot the "wait factor" of OPT-WAIT and WAIT-50, which measures the total time spent in priority-waiting due to each algorithm, normalized by the waiting time of OPT-WAIT. As can be seen from this figure, 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-BC represent the extremes with regard to waiting, WAIT-50 gracefully controls the waiting to match the data contention level in the system.

6.3. Write Probability

All the previously described experiments were carried out for a write probability of 0.25. The next set of experiments look into the performance effects of varying transaction write probabilities. In the first experiment, the write probability was increased to 1.0, keeping the other parameters the same as those of the baseline model. This experiment was conducted for both finite resource and infinite resource scenarios, and the results are shown in Figures 8 and 9a. From this set of figures, we can make the following observations:

(1) OPT-WAIT suffers a substantial performance degradation and does worse than OPT-BC over almost the entire operating region. There are two reasons for this: First, the increased write probability generates higher levels of data contention which, in combination with the population increase effect of the priority wait mechanism, results in a steep increase in the number of conflicts. Second, the conflict-elimination capability of OPT-WAIT vanishes since *all* conflicts are now *bi-directional*. These effects are captured dramatically in Figure 9b, which profiles the average number of conflicts per input transaction under infinite resources.

(2) Although WAIT-50 also employs the priority wait mechanism, it does not suffer OPT-WAIT's performance degradation. This is due to its control mechanism, which ensures OPT-BC-like behavior when high data contention levels are reached by sharply reducing its wait factor. Figure 9c, which plots the wait factor of the algorithms for the infinite resources case, shows this

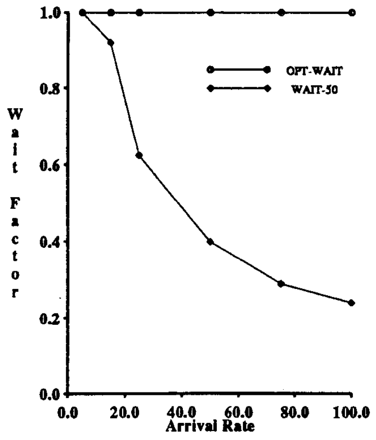

Fig. 7c: Wait Factor (Inf. Res.)

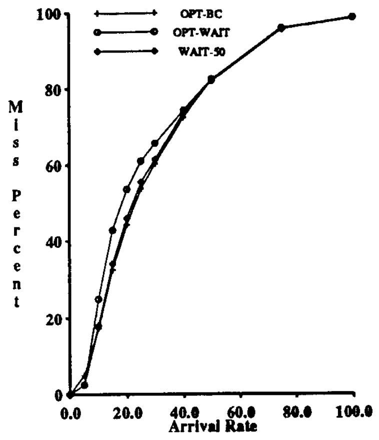

Fig. 8: Write Pr. = 1.0 (Finite Res.)

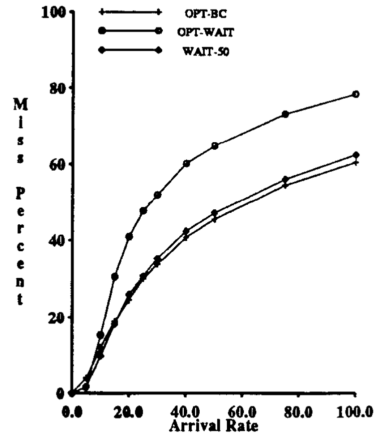

Fig. 9a: Write Pr. = 1.0 (Inf. Res.)

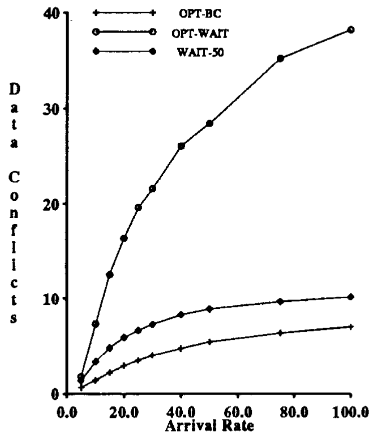

Fig. 9b: Conflicts (Inf. Res.)

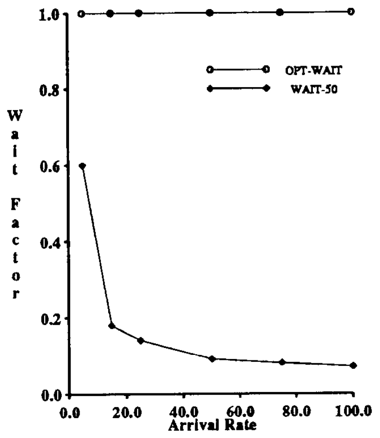

Fig. 9c: Wait Factor (Inf. Res.)

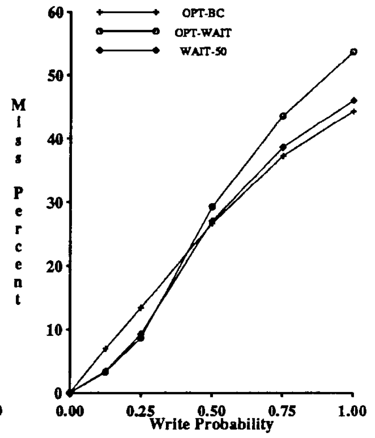

Fig. 10: Finite Res. (Arr. Rate = 20)

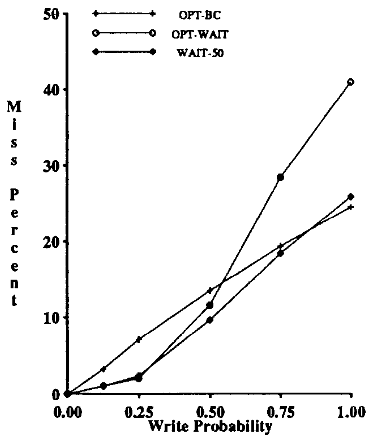

Fig. 11: Inf. Res. (Arr. Rate = 20)

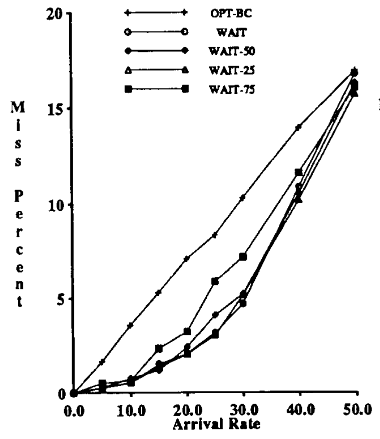

Fig. 12a: Control (Normal Load)

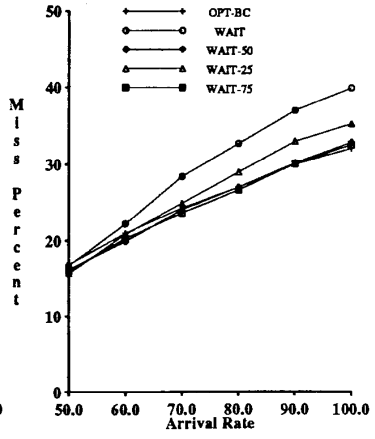

Fig. 12b: Control (Heavy Load)

effect quantitatively.

In the second experiment, the write probability was varied from 0.0 to 1.0, keeping the arrival rate constant at 20 transactions/sec. Figures 10 and 11 show how the algorithms behave under conditions of finite and infinite resources, respectively. These graphs clearly show that while OPT-WAIT performs well at low conflict levels, OPT-BC does much better at high conflict levels. We also observe that WAIT-50 again provides good performance over the entire range.

6.4. Wait Control Mechanism

The final experiment presented here examines the effect of the choice of 50 as the cutoff value for the HPpercent control index. Keeping all parameters the same as those of the baseline model, we measured the performance of WAIT-25 and WAIT-75 under conditions of infinite resources. Figures 12a and 12b give the results of this experiment under normal load and heavy load, respectively. From these graphs, we can make the following observations:

(1) Lowering the cutoff value to 25 percent results in a slight improvement of normal load performance, but worsens the heavy load performance. This behavior is due to the increased wait factor that is delivered by the lowered cutoff value.

(2) Raising the cutoff value to 75 percent has the opposite effect: the normal load performance becomes worse, while there is a slight improvement in heavy load performance. This behavior is due to the decreased priority cognizance that is delivered by the increased cutoff value.

A 50 percent cutoff, therefore, appears to establish a reasonable tradeoff between these opposing forces, providing good performance across the entire range of loading. The basic philosophy is that at light loads, when data contention levels are low, waiting is always beneficial. At heavy loads, however, when data contention levels are high, waiting is the wrong thing to do. WAIT-50 is effective in dynamically making this transition.

7. CONCLUSIONS

In this paper, we have addressed the problem of incorporating transaction deadline information into optimistic concurrency control algorithms. We presented a new real-time optimistic concurrency control algorithm, called WAIT-50, that uses transaction deadline information to improve data conflict resolution decisions. The algorithm features a *priority wait* mechanism that gives precedence to urgent transactions. This mechanism forces low priority transactions to wait for conflicting high priority transactions to complete, thus enforcing preferential treatment for high priority transactions. We showed that the mechanism has a capacity to eliminate some data conflicts due to its wait component, which causes changes to be made to the commit order of transactions. The priority-wait mechanism provides immunity to priority fluctuations by resolving conflicts in a manner that results in the commit of at least one of the conflicting transactions.

While the priority wait mechanism works well at low system 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. The "50 percent" rule is the following: If half or more of the transactions conflicting with a transaction are of higher priority, the transaction is made to wait; otherwise, it is allowed to commit.

Using a simulation model of a RTDBS, we studied the performance of the WAIT-50 algorithm over a range of workloads and operating conditions. WAIT-50 was shown to provide significant performance gains over OPT-BC, a priority-insensitive optimistic algorithm. The wait control mechanism of WAIT-50 was found to be effective in maintaining good performance, even at high data contention levels. In summary, we conclude that the WAIT-50 algorithm utilizes transaction priority information to stably provide improved performance.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions: a Performance Evaluation," *Proc. of the 14th VLDB Conference*, Aug. 1988.

[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data," *Proc. of the 15th VLDB Conference*, Aug. 1989.

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

[Care89] Carey, M., Jauhari, R., and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. of the 15th VLDB Conference*, 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.

[Hari90a] Haritsa, J., Carey, M., and Livny, M., "On Being Optimistic about Real-Time Constraints," *Proc. of the 1990 ACM PODS Symposium*, April 1990.

[Hari90b] Haritsa, J., Carey, M., and Livny, M., "Dynamic Real-Time Optimistic Concurrency Control," *Tech. Report*, University of Wisconsin-Madison, October 1990.

[Jens86] Jensen, E., Locke, C., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proc. 7th IEEE Real-Time System Symposium*, IEEE 1986.

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

[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* Carnegie Mellon University, Dec. 1987.