SCHEDULING REAL-TIME TRANSACTIONS WITH DISK RESIDENT DATA

Robert Abbott
Hector Garcia-Molina

CS-TR-207-89

February 1989

Scheduling Real-Time Transactions with Disk Resident Data

Robert Abbott
Hector Garcia-Molina

Dept. of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Managing transactions with real-time requirements and disk resident data presents many new problems. In this paper we address several: How can we schedule transactions with deadlines? How do the real-time constraints affect concurrency control? How does the scheduling of IO requests affect the timeliness of transactions? How should exclusive and shared locking be handled? We describe a new group of algorithms for scheduling real-time transactions which produce serializable schedules. We present a model for scheduling transactions with deadlines on a single processor disk resident database system, and evaluate the scheduling algorithms through detailed simulation.

1. Introduction

A *real-time database system* (RTDBS) processes transactions with timing constraints such as deadlines. The system guarantees serializable executions while at the same time minimizing the number of transactions that miss their deadlines. Conventional database systems differ from RTDB ones in that the former do not take into account individual transaction timing constraints in making scheduling decisions. Conventional real-time systems, on the other hand, differ from RTDB systems in that they assume advance knowledge of the data requirements of programs and their goal is to guarantee *no* missed deadlines [9]. However, they do not guarantee data consistency. Such systems are called hard real-time.

RTDB systems can be useful in many applications, one of which is air traffic control. The system that directly controls aircraft (e.g., to avoid collisions) is a hard real-time system. However, there are a lot of additional data that must be handled by a RTDB, including weather reports, flight schedules, traffic patterns, and so on.

Transactions on this data have deadlines. For example, a pilot may want to compute fuel requirements taking into account current winds and flight routes. The deadline of such a transaction would be the scheduled flight departure time. If on route the plane approaches a storm, the pilot may request a revised flight plan, taking into account current weather, remaining fuel, and the status of the destination airport. The deadline would reflect how close the plane is to the storm. Missing these deadlines in very undesirable but not an immediate disaster (the plane can leave late or can circle in the air waiting for information). More important than missing a few deadlines is guaranteeing that the database is consistent. For instance, getting the wrong flight plan is worse than getting the right one a bit late. Other applications for RTDB include threat analysis in military systems and program trading in financial systems.

There are many new and challenging problems in designing a RTDB. Two of these problems were studied in [1] :transaction scheduling and concurrency control. In particular, that paper presented several algorithms for resolving lock conflicts and for determining in what order to execute available transactions. The algorithms were studied via detailed simulations. Two major assumptions were made in that work: (a) the database was memory resident, and (b) only exclusive locks were available.

In this paper we continue our investigations of real-time scheduling and concurrency control. Assumptions (a) and (b) have been dropped, a new set of algorithms has been developed, and some additional issues and measures have been considered. The new results, we believe, provide substantial additional insights into the operation of RTDB systems.

Allowing the database to reside on disk, with a portion residing in a main memory buffer pool, introduces more interesting questions that one might initially imagine. For instance, the disk is now a resource that transactions must compete for. How are the disk requests to be scheduled? Do the same real-time priorities that worked for CPU scheduling work for disk scheduling? Some disk controllers do scheduling on their own (trying to minimize head movement). Does this interfere with the real-time scheduling? Since transactions now are suspended more frequently (lock waits and IO waits), there are more opportunities for CPU scheduling. How do the CPU scheduling algorithms respond? Finally, transaction commit must be considered. That is, transactions must flush their dirty pages to disk and write log records. What priorities should these operations receive? Should the log be placed on a separate disk?

Shared locks also introduce a new set of challenging questions. With exclusive locks only, conflicts always involve a pair of transactions, the holder and the requester. The conflict can be resolved by comparing the priorities (e.g., earliest deadline) of each.

With shared locks, the holder can actually be a set of concurrently reading transactions, each with different deadlines. If the requester needs an exclusive lock, what is to be done? What priority does the group have? If the requester needs a shared lock, it could be granted immediately, but there may be other transactions already waiting for exclusive locks. How are the priorities of the waiting transactions compared against that of the new requester? Should the new requester be granted the shared lock or not?

We have extended the algorithms of [1] to cope with disk data and shared locks. In addition, we have studied concurrency control algorithms not considered initially, including one that *promotes* transactions that are blocking higher priority transactions. Finally, we have also considered two supplementary measures (in addition to mean number of missed deadlines). One is the mean tardiness of transactions, i.e., average time by which transactions miss their deadlines. The second is the response of the system to a batch of transactions that arrive at once. Such an "input step function" emulates a severe overload situation. Such overloads may not be frequent, but having algorithms that can cope with them gracefully is important.

2. Model and Assumptions

In this section we describe our basic assumptions and real-time transaction model. The system consists of a *single processor*, a disk-based database, and a main memory buffer pool. (The multiple processor case is also of interest, but we have not addressed it yet.)

The unit of database granularity we consider is the *page*. Transactions access a sequence of pages. If a page is not found in the buffer pool, a disk read is initiated to transfer the page to the pool. Modified pages are held in the pool until the transaction completes. At that time, the log is flushed and the transaction commits. Finally, the modified pages are written back to disk to free space in the buffer pool. We assume that the buffer pool is large enough so that a transaction never has to write modified pages to disk until after commit. Thus, aborting a transaction involves no disk writes. We assume that the log is kept on a disk (or tape) separate from the database disks (the most common scenario in practice).

Each arriving transaction has a release time r, a deadline d, and a run time estimate E. The release time is the earliest time the transaction can be started and is usually the arrival time. The deadline is the desired maximum commit time. Estimate E approximates the duration of the transaction on an unloaded system. It takes into account both the CPU and disk read time involved. Parameters r, d, and E are known to the system as soon as the transaction arrives. However, the access pattern of the transaction is not known in advance. As the transaction is executed, it asks to read or write one page at a time. Our decision to assume knowledge of a run time estimate but no knowledge of data requirements is justified because it is easier to estimate the execution time of a transaction than to predict its data access pattern. And in any case, *E* is simply an estimate that could be wrong or not given at all.

The RTDB system schedules transactions with the objective of minimizing the number of missed deadlines. As the system runs it may observe that some transactions have already missed their deadlines before committing. We assume that all transactions must be executed eventually, regardless of whether they are tardy or not. In this case, we will assume that the priority of tardy transactions increases, in order to limit the tardiness. For instance, in our air control example, the flight plans should be delivered, even if late. Other options for handling tardy transactions and their performance are discussed in [1].

As discussed in the introduction, we assume that transaction executions must be serializable [4]. (If transaction semantics were known, other strategies could be used [5].) We assume that serializability is enforced via a locking protocol that allows for shared and exclusive locks. Deadlock detection is used to find and break deadlocks. We have selected a locking protocol because locking is widely used in practice; other strategies may be possible [2] but are not considered here.

3. Scheduling Algorithms

Our scheduling algorithms have three components: a policy for assigning priorities to tasks, a concurrency control mechanism, and a policy for scheduling IO requests. The priority policy controls how a priority is assigned to a real-time transaction. The concurrency control mechanism can be thought of as a policy for resolving conflicts between two (or more) transactions that want to lock the same data object. Some concurrency control mechanisms permit deadlocks to occur. For these a deadlock detection and resolution mechanism is needed. The third component controls how scheduling of the IO queue is done, i.e., whether a transactions's real-time constraints are used to decide which IO request is serviced next.

Each component may use only some of the available information about a transaction. In particular we distinguish between policies which do not make use of *E*, the runtime estimate, and those that do. A goal of our research is to understand how the accuracy of the runtime estimate affects the algorithms that use it.

3.1. Assigning Priorities

There are many ways to assign priorities to real-time tasks [6, 7]. We have studied three.

First Come First Serve. This policy assigns the highest priority to the transaction with the earliest release time. If release times equal arrival times then we have the traditional version of FCFS. The primary weakness of FCFS is that it does not make use of deadline information. FCFS will discriminate against a newly arrived task with an urgent deadline in favor of an older task which may not have such an urgent deadline. This is not desirable for real-time systems.

Earliest Deadline. The transaction with the earliest deadline has the highest priority. A major weakness of this policy is that it can assign the highest priority to a task that has already missed or is about to miss its deadline. One way to solve this problem is to use an overload management policy to screen out transactions that have missed or are about to miss their deadlines [1].

Least Slack. For a transaction T we define a slack time $S = d - (t + E - P)$, where t is the current time. The slack time is an estimate of how long we can delay the execution of T and still meet its deadline. If $S \geq 0$ then we expect that if T is executed without interruption, it will finish at or before its deadline. A negative slack time is an estimate that it is impossible to make the deadline. A negative slack time results either when a transaction has already missed its deadline or when we estimate that it cannot meet its deadline.

Note that Least Slack is very different from Earliest Deadline in that the priority of a task depends on how much service time it has received. The slack of a transaction which is executing does not change. (Its service time and the clock time increase equally.) The slack time of a transaction which is not executing *decreases*. Hence the priority of that transaction *increases*.

A natural question to consider is how often to evaluate a transaction's slack. We consider two methods. With the first, *static evaluation*, the slack of a transaction is evaluated once when the transaction arrives. This value is the transaction's priority for as long as the transaction is in the system. (If a transaction is rolled back and restarted, the slack is recalculated. In effect, the transaction is re-entering the system as a new arrival.) Under the second method, *continuous evaluation*, the slack is recalculated whenever we wish to know the transaction's priority. This method yields more up-to-date information but also incurs more overhead.

Our performance studies have shown that sometimes it is better to use static evaluation and sometimes it is better to use continuous evaluation. (See Section 4.) The majority of our experimental results use static evaluation. We chose this because static evaluation performed better than continuous at higher load settings, which is where we performed many of our experiments.

3.2. Concurrency Control

If transactions are executed concurrently then we need a mechanism to order the updates to the database so that the final schedule is serializable. Our mechanisms allow shared and exclusive locks. Shared locks permit multiple concurrent readers. Before presenting the algorithms we introduce some terminology and explain the conventions we use to implement two-phase locking with shared and exclusive locks.

The priority of a data object O is defined to be the maximum priority of all transactions which hold a lock on object O. If O is not locked then its priority is undefined.

Let T be a transaction requesting a shared lock on object O which is already locked in shared mode by one or more transactions. Transaction T is allowed to join the read group only if the priority of T is greater than the maximum priority of all transactions, if any, which are waiting to lock O in exclusive mode. In other words, a reader can join a read group only if it has a higher priority then all waiting writers. Otherwise the reader must wait.

Conflicts arise from incompatability of locking modes in the usual way. That is, an exclusive lock request conflicts with both shared and exclusive lock modes, and a shared lock request conflicts with an exclusive lock mode.

We are particularly interested in conflicts that can lead to priority inversions. A *priority inversion* occurs when a transaction T of high priority requests and blocks on a lock for object O which has a lesser priority than T. This means that T has a higher priority then the transaction(s) which holds the lock on O. T must wait until the lock holder(s) releases its lock on O, either voluntarily or involuntarily. Conflicts which cannot lead to priority inversion, i.e., the priority of the requester is less than the priority of the object, are handled by having the requester wait. Of course a deadlock detection method must be employed to detect cycles of waiting transactions. We assume that transactions do not self-deadlock. We now discuss four techniques that we use to resolve conflicts.

In the following discussion let T_R be a transaction that is requesting a lock on a data object O that is already locked by transaction T_H. Furthermore, the lock modes are incompatible and T_R has a higher priority than the priority of O. Thus the priority of T_R is greater than T_H. Namely we have a priority inversion.

Wait. Under this policy priority inverting conflicts are handled exactly as non-priority inverting conflicts. That is, the requesting transaction always blocks and waits for the data object to become free.

This is the standard method for most DBMS which do not execute real-time transactions. All conflicts are handled indentically and the concurrency control mechanism makes no effective use of transaction priorities.

Wait Promote. Wait Promote handles conflicts as Wait does except when a priority inversion occurs. The high priority transaction T_R will block and wait but now we promote the priority of the lock holder T_H so it is as high as the priority of T_R. In other words, T_H inherits the priority of T_R. (Since locks are retained until commit time, T_H will keep its inherited priority until it commits or is restarted. In the event that T_H is restarted, e.g., because of deadlock, it assumes its normal priority. A pure implementation of priority inheritance would demote the priority of T_H if T_R is aborted before T_H finishes. We chose not to implement demotion. Our tests showed that it occurs so seldom that any difference in overall performance is not measurable.) This method for handling priority inversions was proposed in [8].

The reason for promoting T_H is that it is blocking the execution of T_R a higher priority transaction. Thus T_H should execute at an elevated priority in order to get it done and removed so that T_R can execute. Priority inheritance ensures that only a transaction with priority greater than T_R will be able to preempt T_H from the CPU. A transaction T_I of intermediate priority, a priority greater than T_H and less than T_R, would normally be able to preempt T_H. But with priority inheritance, T_I has a lesser priority than T_H which is now executing on behalf of T_R.

What if the object is locked by more than one transaction? In this event all transactions in the read group will inherit the priority of T_R. Note that a priority inversion can affect only some of the transactions in a read group. For example, the requesting transaction may have a priority that is greater than only some of the transactions in the read group. These transactions will inherit the greater priority of the requester. The priority of the other transactions in the read group remains unchanged. Thus every transaction holding a lock on object O has a priority that is at least as high as the highest priority transaction which is waiting for the lock.

Finally, the property of priority inheritance is transitive. If, for example, T_H is blocked by transaction T_{HH}, and the priority of T_{HH} is less than T_R then T_{HH} will inherit the priority of T_R.

Note that when priority inheritance is combined with Least Slack, continuously evaluated, T_H inherits not a static priority but a priority function which evaluates the slack of T_R.

High Priority. The idea of this policy is to resolve a conflict in favor of the transaction with the higher priority. The favored transaction, the winner of the conflict, gets the lock on the contested object. We implement this policy by comparing transaction priorities at the time of the conflict. If the priority of T_R is greater than the priority of object O, and thus greater than every transaction holding a lock on O, then we abort the lock holders thereby freeing the lock for T_R. T_R can resume processing; the lock holders are rolled back and scheduled for restart. If the priority of T_R is less than or equal to the priority of O then T_R blocks to wait for O to become free.

An interesting problem arises when we use Least Slack to prioritize transactions. Recall that under this policy, a transaction's priority depends on the amount of service time that it has received. Rolling back a transaction to its beginning reduces its effective service time to 0 and raises its priority under the Least Slack policy. Thus a transaction T_H, which loses a conflict and is aborted to allow a higher priority transaction T_R to proceed, can have a higher priority than T_R immediately after the abort. The next time the scheduler is invoked, T_R will be preempted by T_H. T_H may again conflict with T_R initiating another abort and rollback.

Our solution to this problem is to compare the priority of T_R against that of each lock holder assuming that the lock holder were aborted. Using the notation $p(T_H)$ to denote the priority of T_H and $p(T_H^A)$ to denote the priority of T_H were it to be aborted, we can write this algorithm as follows:

High Priority Conflict Resolution Policy

```
if for any $T_H$ holding a lock on $O$
  $p(T_H) < p(T_R)$ and $p(T_H^A) < p(T_R)$
then
    Abort each lock holder
else
    $T_R$ blocks
```

Conditional Restart Sometimes High Priority may be too conservative. Let us assume that we have chosen the first branch of the algorithm, i.e., T_R has a greater priority than T_H and T_H^A. We would like to avoid aborting T_H because we lose all the service time that it has already consumed. We can be a little cleverer by using a conditional restart policy to resolve conflicts. The idea here is to estimate if T_H, the transaction holding the lock, can be finished within the amount of time that T_R, the lock requester, can afford to wait.

Let S_R be the slack of T_R and let $E_H - P_H$ be the estimated remaining time of T_H. If $S_R \geq E_H - P_H$ then we estimate that T_H can finish within the slack of T_R. If so, we let T_H proceed to completion, release its locks and then let T_R execute. This saves us from restarting T_H. If T_H cannot be finished in the slack time of T_R then we restart T_H (as in the previous algorithm). This modification yields the following algorithm:

Conditional Restart Conflict Resolution Policy

```
if $p(T_H) < p(T_R)$ and $p(T_H^A) < p(T_R)$
then
        if $E_H - P_H \leq S_R$
        then
                $T_R$ blocks
                $T_H$ inherits the priority of $T_R$
        else
                Abort $T_H$
else
        $T_R$ blocks
```

Note that if T_R blocks in the inner branch, then T_H inherits the priority of T_R. This inheritance is exactly the same as described in the Wait Promote algorithm.

We only implement Conditional Restart if the conflict is one-on-one, i.e., there is no read group involved. We chose this because it is NP-complete to choose a maximal subset of readers all of which can finish within the slack of the requester. Furthermore we do not consider chained blockings as we did in our earlier study [1]. That is, we only make the special Conditional Restart decision if the requester conflicts with exactly one lock holder and the lock holder is not blocked waiting for some other lock. Experience with our earlier simulations indicated that chained blockings were rare, so that the payoff for handling them in a clever way was limited.

3.3. IO Scheduling

In a non memory resident database system, the disk is an important resource which can be managed to optimize various performance criteria. In conventional systems the usual goal is to maximize the throughput of the IO system. One way that this is accomplished is by using a disk scheduling algorithm (e.g., SCAN) to order the sequence of IO requests so that the mean seek time is minimized. While this may be good for maximizing throughput, it may be bad for a real-time system which is trying to meet transaction deadlines. For example, SCAN may order a batch of requests so that an IO request from a transaction with an early deadline is serviced last.

In this paper we study the consequences of using algorithms to schedule IO requests based on transaction priorities as opposed to minimization of disk head seek times. We have looked at two ways to schedule the IO queue.

FIFO. When FIFO is used to schedule the IO queue, requests are serviced in the order in which they are generated. This service order is somewhat related to transaction priorities because IO requests are generated by the CPU, which is scheduled by priority. The ordering is essentially random with respect to cylinder position on the disk.

Priority. Under this policy each IO request has a priority which is equal to the priority of the transaction which issued the request. The next IO request to service is the one with the highest priority. Thus a newly arrived request from a transaction with a high priority can leapfrog over other requests which have been waiting longer in the IO queue. We also expect this ordering to be random with respect to cylinder positions on the disk.

In our model there are two types of IO requests: reads, that are issued by unfinished transactions, and writes that are generated by committed transactions that are flushing their updates back to disk. (The log resides on a separate device, so it receives only log writes which are serviced FCFS.) Giving higher priority to reads over writes is desirable because it will speed the completion of transactions which are trying to meet their deadlines. Giving high priority to writes does not enhance performance directly because the transactions which issued the writes have already committed. In fact, as our studies have shown, giving high priority to writes can decrease performance if it excessively delays the servicing of read requests. The priority of writes cannot be too low however as writes must be completed in order to free space in the memory buffer pool.

In our experiments writes have the same priority as the transaction that issued them. For the priorities FCFS and Earliest Deadline, this means that writes are given a relatively high priority. (The arrival times and deadlines of committed transactions are usually earlier than those of uncommitted transactions.) Because we use static evaluation to implement Least Slack, the slack times of committed transactions are not necessarily larger or smaller than those of uncommitted transactions.

4. Simulation Results

To study the algorithms we have built a program to simulate a RTDB. The names and meanings of the four parameters that control the configuration of the system resources are given in Table 1. The database log is maintained on a separate device which is of equal speed as the database disks. Each disk has its own queue of service requests.

Parameter	Meaning	Base Value
DBsize	Number of pages in database	400
MemSize	Number of pages in memory buffer pool	200
NumDisks	Number of disks	2
IOtime	Time to perform a disk access (read or write)	25 ms.

Table 1. System Resource Parameters.

The database buffer pool is modeled as a set of pages each of which can contain a single database object. We do not model each buffer page individually, that is, we do not maintain a free list of pages, nor do we keep track of which pages have been modified. Instead we model the buffer pool as a collective set. When a transaction attempts to read an object, the system generates a random boolean variable which has the value true with probability $\frac{MemSize}{DBsize}$. If the value is true then the page is in memory and the transaction can continue processing. If the value is false then an IO service request is created and placed in the input queue of the appropriate disk. The database is partitioned equally over the disks and we use the function $D = \left\lceil \frac{i * NumDisks}{DBsize} \right\rceil$ to map an object i to the disk where it is stored.

Transactions characteristics are controlled by the parameters listed in Table 2. Transactions enter the system with exponentially distributed inter-arrival times and they are ready to execute when they enter the system (i.e., release time equals arrival time). The number of objects accessed by a transaction is chosen from a normal distribution with mean *Pages* and the actual database items are chosen uniformly from the database. Each page is updated with with probability *Update*. Pages which are updated are locked exclusively, other pages are locked in shared mode. Updated pages are stored in the buffer pool until a transaction commits and then they are flushed out to disk.

A transaction has an execution profile which alternates lock requests with equal size chunks of computation, one for each page accessed. Thus the total computation time is directly related to the number of items accessed. Let C denote the CPU requirement for a transaction; then $C = Pages' * CompFactor$. (We use *Pages'* to denote the actual number of pages for a specific transaction rather than the mean.) The IO service requirement for a transaction has two components: the first is the number of IO requests to read pages from the disk into memory; the second is the IO requests needed to write the modified pages back to the disk. Thus the total amount of IO service time needed is $I = IOtime * Pages' * \left(1 - \frac{MemSize}{DBsize}\right) + IOtime * Pages' * Update$. Thus the total runtime service needed by a transaction executing in an unloaded system is $R = C + I$. The accuracy of a transaction's runtime estimate E with respect to R is controlled by the parameter *EstErr*, $E = R * (1 + EstErr)$. How we choose the value of *EstErr* is explained later when we discuss the experimental results.

Parameter	Meaning	Base Value
ArrRate	Mean exponential arrival rate of jobs	7 jobs/sec
Pages	Mean number of pages accessed per job	8
CompFactor	CPU computation per page accessed	15 ms.
Update	Probability that an page is updated	0.5
MinSlack	Minimum slack as fraction of total runtime	2
MaxSlack	Maximum slack as fraction of total runtime	8
EstErr	Error in runtime estimate as fraction of total runtime	0
Rebort	Time needed to abort a transaction	5 ms.
MaxActive	Limit of number of active jobs in system	25

Table 2. Transaction Parameters.

The assignment of a deadline is controlled by two parameters *MinSlack* and *MaxSlack* which set a lower and upper bound respectively on a transaction's slack time. A deadline is assigned by choosing a slack time uniformly from the range specified by the bounds. *Rebort*, controls the amount of CPU time needed to abort or restart a transaction. Aborting a transaction consists of rolling it back and removing it from the system. The transaction is not executed. When a transaction is restarted it is rolled back and placed again in the ready queue. (Recall that updates are flushed after transaction commit, so an abort does not generate an IO service request.) Note that aborts are generated by the overload management policy, restarts result from lock conflicts. The program does not explicitly account for time needed to execute the lock manager, conflict manager, and deadlock detection manager. These routines are executed on a per data object basis and we assume that the costs of these calls are included in the variable that states how much CPU time is needed per object that a transaction accesses. Context switching and the time to execute the scheduler is also ignored.

Deadlocks are detected by maintaining a wait-for graph and searching for cycles whenever a new arc is added to the graph. When a deadlock is detected a victim is selected by choosing the transaction with lesser priority of the two transactions which correspond to the arc which completed the cycle in the graph. The victim is rolled back and placed in a special queue until the transaction with which it deadlocked is flushed from the system, either because it finishes or because it is aborted. When this happens the victim is placed in the ready queue and allowed again to enter the system. This system of enforced waiting is necessary to prevent the excessive formation of deadlocks.

In the following sections we discuss some of the results of eight different experiments that we performed. Due to space considerations we cannot present all our results but have selected the graphs which best illustrate the differences and performance of the algorithms. For each experiment we ran the simulation with the same parameters for 20 different random number seeds. Each run, except for the input step function experiment, continued until at least 700 transactions were executed. For each algorithm tested, numerous performance statistics were collected and averaged over the 20 runs. In particular we measured the percentage of transactions which missed their deadlines, the average amount of time by which transactions missed their deadlines, and the number of restarts caused by lock conflicts. The percentage of missed deadlines is calculated with the following equation: $\%missed = 100 * \frac{tardy\ jobs + aborts}{jobs\ processed}$. A job is processed if either it executes completely or it is aborted.

Another metric that we use is the average tardy time of all committed transactions. A transaction that commits before or on its deadline has a tardy time of zero. Aborted transactions do not contribute to this metric. It is these averages and 95% confidence intervals that are plotted in the following figures.

In this study we have included tardy jobs and aborted jobs together in the *%missed* metric. For some applications it may be useful to describe a separate metric for aborted jobs as they represent tasks which were never completed and as such may be more serious than simply tardy jobs. For reasons of space we do not study these other metrics here. Note that we are not particularly interested in transaction response times as conventional performance evaluations of concurrency control mechanisms are. The reason is that response time is not critical as long as a transaction meets its deadline. We are interested in learning how the various strategies are affected by load, the percentage of updates per transaction, the size of the memory buffer, and error of the runtime estimate.

For many of the experiments the base values for parameters determining system configuration and transaction characteristics were as shown in tables 1 and 2. These values are not meant to model a specific real-time application but were chosen as reasonable values within a wide range of possible values. We chose the arrival rate so that the corresponding CPU utilization (an average 0.84 seconds of computation arrive per second) is high enough to test the algorithms. It is more interesting to test the algorithms in a heavily loaded rather than lightly loaded system. (We return to this issue in the conclusions section.)

Section 3 proposed three different methods for assigning priority and four methods for managing concurrency. Also IO scheduling can be done in two different ways. Taking the cross product yields 24 different algorithms. Table 3 summarizes the methods of Section 3 and provides the abbreviations that we will use when referring to them.

Priority	FCFS - First Come First Serve ED - Earliest Deadline LS - Least Slack
Concurrency	W - Wait WP - Wait Promote HP - High Priority CR - Conditional Restart
IO Scheduling	FIFO Priority

Table 3. Summary of Scheduling Policies.

Effect of increasing load.

In this experiment we varied the arrival rate from 6 jobs/sec to 8 jobs/sec in increments of 0.5. The other parameters had the base values given in Tables 1 and 2. This set of parameters is designed to load the CPU more heavily than the IO system. The CPU utilization ranges from 0.72 to 0.96 seconds of computation arriving per second. The IO system experiences a range in utilization of 0.60 to 0.80 seconds of I/O service requests arriving per second.

Figures 1-4 graph the three priority policies for each of the four different concurrency control strategies. Our first observation is that FCFS consistently performs worse than ED and LS. Also, where the other priority algorithms perform better with the concurrency control strategies that use promotion, FCFS does not. This is understandable as FCFS is a poor notion of priority for real-time transactions.

The relative performance of ED and LS depends on the type of concurrency control that is used. Fortunately we can state some general rules as to when ED or LS may be better. The policy ED operates best under low levels of load and when the concurrency control strategy produces few restarts due to lock conflict resolution. At high load levels ED tends to assign high priorities to transactions which will miss their deadlines anyway. This can prevent transactions with later, but feasible, deadlines from finishing on time. Conflict resolution policies which can produce a lot of restarts (e.g., HP and CR) work against ED because the restarts effectively increase the arrival rate of new transactions into the system, thus increasing the load. LS also performs well under low load conditions and it performs better than ED under higher loads and with HP and CR. Under HP and CR, LS produces fewer restarts than ED because LS makes use of an extra test that can prevent transactions from being restarted (see Section 3).

Examining the graphs, we see that ED is better than LS when when the simple Wait strategy is used for concurrency control (Figure 1). When promotion is added to the Wait strategy, Figure 2, ED and LS perform comparably although ED is slightly better at lower loads and LS slightly better at higher loads. Both policies are much better than FCFS.

Under the two policies which restart transactions when conflicts occur, HP and CR, we note that LS is clearly the better policy when the load is high, Figures 3 and 4. Strategy ED performs well when the load is low but its performance degrades rapidly once the CPU utilization is greater than 0.84. As we stated before, restarts work against ED because they effectively increase the arrival rate of new transactions into the system.

Having discussed the priority policies we now turn our attention to the concurrency control policies. The performance of a concurrency control strategy depends on the priority policy used, so it is hard to compare concurrency without including priority. We will compare combinations of priority (ED and LS) and the four concurrency control strategies. We ignore FCFS because of its poor performance. Figure 5 graphs the eight combinations.

The reader can see for himself that no one combination is always better. ED/WP is best when the load is light but LS/WP is better when the load is high. Thus we see that WP works well with both LS and ED, but CR works better with LS; HP is not very good with either priority policy.

Ideally we want our algorithms to schedule all transactions such that all deadlines are met. However, if this is not possible, then we would want to minimize the amount by which tardy transactions miss their deadlines. Figures 6 and 7 graph the average overdue time in seconds against arrival rate. With the Wait strategy, Figure 6, we observe that ED has the least average overdue time, then FCFS and finally LS. These results are not surprising for it is known that ED minimizes the maximum task tardiness and LS maximizes the minimum task tardiness [3]. Under Wait-Promote, Figure 7, ED still has the minimum average task tardiness but now LS is better than FCFS because promotion works best with a good priority assignment. The performance of the different concurrency strategies with respect to task tardiness parallels what we observed with missed deadlines, namely, W and WP are best with ED, and WP and CR are best with LS.

In the experiment described above, the priority policy LS was implemented using static evaluation. We are also interested in learning how LS performs when continuous evaluation is used.

Figure 8 compares LS/WP and LS/CR for both static and continuous evaluation. The reader can see that no algorithm is best under all load settings. Under low arrival rates, the continuously evaluated versions of LS work better than the versions that use static evaluation. However, under higher loads, the opposite is true. This is an interesting and counter-intuitive result. The explanation is that transactions executing under LS, continuously evaluated, can experience priority reversals. We illustrate with an example.

At time t let T_1 and T_2 be transactions with slack times of 5 and 6 seconds respectively. Since T_1 has the lesser slack and thus the greater priority, it gains the processor and begins to execute. Two seconds later a new transaction arrives. The scheduler is invoked and slack times are recalculated. The slack of T_1 is unchanged, the slack of T_2 has decreased by 2 to 4. So now T_2 has the greater priority and gains the processor. A similar event may occur 2 seconds later which would cause the priorities of T_1 and T_2 to reverse again. If T_1 and T_2 are the highest priority transactions in the system, this continuous method of priority evaluation virtually guarantees that the two transactions are executed concurrently in the manner just described. If T_1 and T_2 have overlapping read and write sets then this scheduling method increases the likelihood that a conflict will occur. It also increases the likelihood that a deadlock will occur. Both of these events are undesirable.

Biasing the Run Time Estimate

The concurrency control strategy Conditional Restart uses the runtime estimate to decide if a low priority transaction holding a lock can finish within the slack time of the higher priority transaction requesting the lock. We can easily describe the behavior of CR for the extreme values of E. When E = 0, CR will always judge (assuming that the lock requester has a positive slack, this is true if the transaction has not missed its deadline) that the lock holder can finish within the slack of the lock requester. Thus the lock requester will always wait for the lock holder. Since promotion is used by CR, this behavior is exactly the same as WP.

At the other extreme, when E is much larger than R, the algorithm will nearly always judge that the lock holder cannot finish within the slack time of the lock requester. Thus the lock holder will be rolled back and restarted. This behavior is the same as the concurrency control strategy HP. When the value of E is not so extreme CR will behave somewhere between WP and HP.

The runtime estimate is also used by the priority policy Least Slack. We performed a number of experiments to study biasing the runtime estimate affects the performance of LS. Our results showed that LS, under static evaluation, is surprisingly robust to inaccuracy in the estimate. To illustrate, we summarize a "worst case" experiment that yielded the largest performance degradation.

Half of the transactions were made to overestimate their runtime estimate $E = R * (1 + EstErr)$; the other half underestimated $E = R * (1 - EstErr)$. (Recall that R is the service time needed in an unloaded system. If *all* transactions over or underestimate, LS is not affected greatly.) The value of *EstErr* was varied from 0 to 5 in increments of 0.5. (Negative runtime estimates were converted to $E = 0$.) Figure 9 shows that even with this worst case bias, LS is not very sensitive to errors in the runtime estimate.

When continuous evaluation is used, the performance of LS, is more sensitive to error in the runtime estimate, Figure 10. This is understandable since continuous evaluation means that the inaccurate runtime estimate will be used many more times to make scheduling decisions.

Effect of Increasing Memory

An important difference between this paper and our earlier study on real-time transaction scheduling is that the database is now disk resident. An interesting question is how the relative size of the memory with respect to the database affects the performance of the various scheduling algorithms.

In this experiment we varied the value of *MemSize* from 200 to 400 in increments of 50. The other parameters had the values shown in Tables 1 and 2. Thus the size of the memory resident fraction of the database varied from 0.5 to 1.0 of the total database.

Note that the proportion of memory resident database influences the assignment of deadlines to transactions. This happens because deadlines are assigned with respect to the total service time, both CPU and IO, required by a transaction. As the memory increases, the IO service requirement decreases (pages are more likely in memory), and the deadlines are shortened proportionately. In this way we can compare the scheduling of sets of transactions with similar task urgencies in system configurations of various memory sizes.

Our expectation, as memory size increases, is that all scheduling algorithms will perform better. The reason is that transactions will be doing much less IO and the time spent waiting for service in IO queues decreases. (CPU utilization and waiting time are unchanged.) Thus transactions will receive quicker service and are more likely to meet their deadlines.

Figure 11 graphs the results for the four algorithms which schedule all transactions and use LS to assign priority. It confirms our expectation that algorithms will perform better as the memory size increases. The results for other algorithms are similar.

Effect of Fraction of Pages Updated

Another new issue which this paper addresses is that of shared locks and exclusive locks. Recall that *Update* determines the fraction of pages in a transaction's dataset which are modified. These pages form the writeset and are locked in exclusive mode. Pages which are not locked exclusively are locked in a shared mode and form the readset of the transaction. The value of *Update*, by controlling the size of the read and write sets, impacts the probability of conflict between two concurrent transactions. The value of *Update*, by determining the size of the writeset, also contributes to the IO load of the system. This is because updates are written to disk after a transaction commits.

In this experiment we varied the value of *Update* from 0.2 to 0.8 in increments of 0.1. When $Update = 0.2$, only 20% of a transaction's pages are locked exclusively. When $Update = 0.8$ 80% of the pages accessed by a transaction are updated. The other parameters had the values shown in Tables 1 and 2. In this experiment the IO load varies from 0.49 to 0.91.

Figure 12 shows the four different types of concurrency control with the ED priority policy and with IO scheduling done by priority. The general shape of the curves is as expected: when *Update* is large, the performance is poor because the rate of conflict is high and because the overall IO load is high. As we saw earlier ED performs best with the concurrency strategies W and WP which cause fewer restarts than HP and CR. When *Update* is small, the conflict rate is lower and all algorithms perform equally well.

Figure 13 plots ED/W and ED/WP with priority IO scheduling against ED/W and ED/WP with FIFO IO scheduling. Somewhat surprising is the result that the versions with IO scheduling perform worse than the versions without IO scheduling in the high update (right side) and thus high IO load part of the scale. The result is explainable, however, when we note that at this end of the scale most of the IO, roughly 62%, consists of writing modified pages back to disk. As we discussed in Section 3, giving high priority to writes can delay the service of read requests with lower priorities. One conclusion that we can draw from this is that it may be advisable to give IO writes lower priorities than reads. We do not investigate this issue any further in this paper.

Performance Under a Sudden Load Increase

In the previous experiments, we studied the performance of the various algorithms under an increasing but steady load. In this experiment we study the algorithms under a different type of load increase namely a batch of arrivals in an otherwise idle system. To simulate this input step function we programmed the simulator so that a set of transactions arrived all at the same time. The system then executed all the jobs in the set. The number of jobs in the set varied from 20 to 60 in steps of 10. The parameters controlling job characteristics were the same as shown in tables 1 and 2 except for the following differences: *Pages* = 10, *CompFactor* = 10, and *MaxSlack* = 30. The major difference here is the change in *MaxSlack*. This was necessary to produce a set of transactions with a large amount of variability in deadlines.

In reality, we would not expect all jobs in an overload to arrive at the same instant, nor would we expect the system to be completely idle at this time. However, our idealized step function model lets us study the impact of an overload without distracting second order effects. If an algorithm performs well under a step function, we could also expect it to do well in a system that has plenty of spare capacity under normal circumstances but is suddenly faced with a flurry of jobs (e.g., a radar system facing a sudden all-out enemy attack).

A good strategy for operation in this situation might be to employ an overload management policy to abort transactions which have missed their deadlines. If these transactions must be executed anyway then they should be restarted only after the spike has passed. We may also want to limit priority inversions from occurring. The reasoning is that if a high priority job blocks, it will almost certainly miss its deadline if forced to wait on a low priority job. Instead we may want to abort the low priority job and restart it later when the load has decreased.

Figure 14 shows the four different concurrency control policies for both the ED and LS priority assignment policies. IO scheduling is done according to job priority. It is easy to see that ED/HP is clearly the best algorithm. HP is also the best version of the LS algorithms The same results were observed when no overload management policy was used, although the effect was not as pronounced.

Effect of IO Scheduling

In this experiment we compared the versions of scheduling algorithms which use transaction priorities to schedule IO requests against versions which service IO requests using FIFO. To better study the effects, we changed the base values of three parameters to create a configuration where the disks are much more loaded then the CPU. The parameters and the new values were: *Pages* = 13, *CompFactor* = 5, and *Update* = 0.2. (The other parameters had the values shown in tables 1 and 2.) Thus the average transaction reads more pages and does less computation. The fraction of pages updated is less so the disks service more read requests than writes. The system load was varied by increasing the arrival rate from 6 jobs/sec. 8 jobs/sec in steps of 0.5. Hence, the CPU utilization varied from 0.39 to 0.52, and the IO utilization varied from 0.68 to 0.91.

Figure 15 plots the four algorithms which use LS and FIFO against the same four algorithms using LS and Priority IO scheduling. The results clearly show that using transaction priorities to schedule IO requests can yield significant performance improvements over scheduling using FIFO. This is especially true for the two concurrency algorithms which use promotion, i.e., WP and CR. This is understandable since a transaction which inherits the higher priority of a blocking transaction will receive better service from the disk and will execute faster. This unblocks the high priority transaction sooner, and it has a better chance of meetings its deadline.

Priority is better than FIFO but both methods ignore disk head position and do not minimize the average seek time. Would a disk scheduling algorithm which minimizes average seek time, (e.g., SCAN) but is ignorant of transaction time constraints, yield better performance in a real-time system than our algorithm which considers transaction time requirements? We cannot answer this question directly but we have performed an experiment which indicates how much faster the average seek time of a SCAN type algorithm would have to be for it to match the real-time performance of our Priority algorithm. In this experiment we fixed the parameter settings at the values they had in the previous experiment and set *ArrRate* = 8 job/sec. Then we varied the increased the speed of the disks by decreasing the value of *IOtime* from the initial 25 ms. to 20 ms. in steps of 1. Figure 16 plots the performance of LS/WP/FIFO, which, like a SCAN type algorithm, does not use transaction priorities for IO scheduling. The horizontal line is the performance of LS/WP/Priority when $IOtime = 25$ ms. The two curves intersect at $IOtime = 22.8ms$ indicating that a SCAN type algorithm must have an average seek time roughly 9% less than Priority in order for it to yield comparable real-time performance. Of course it is possible to design a SCAN type algorithm which also uses transaction priorities, but we have not investigated this possibility.

5. Conclusions

In this paper we have presented various transaction scheduling options for a real-time database system. Our simulation results have illustrated the tradeoffs involved, at least under one representative database and transaction model. Before reaching some general conclusions, we would like to make two observations.

The first observation is that our base parameters represent a high load scenario. One could argue that such a scenario is "unrealistic." However, we believe that for designing real-time schedulers, one must look at precisely these high load situations. Even though they may not arise frequently, one would like to have a system that misses as few deadlines as possible when these peaks occur. In other words, when a "crisis" hits and the database system is under pressure is precisely when making a few extra deadlines could be most important.

It could also be argued that some of the differences between the various scheduling options is not striking. In many cases, the difference between one option and another one is a few percentage points. If we were discussing transaction response times, then a say 10 percent improvement would not be considered impressive by some. However, our graphs show missed deadlines (in most cases) and we believe that this is a very different situation. Again, the difference between missing even one deadline and not missing it could be significant. Thus, if we do know that some scheduling options reduce the number of missed deadlines, why not go with the best one?

And which are the best options? It is difficult to make any absolute statements, but we believe the following statements hold under most of the parameter ranges we tested. (All our additional results not shown in this paper also substantiate these statements.)

When the load is continuous and steady:

(a) Of the tested priority policies for real-time database systems, Least Slack (LS) is the best overall. It always performed better than FCFS, and was better than ED under the higher load ranges. Earliest deadline (ED) is the second choice for assigning priorities.

(b) Of the concurrency control policies we tested, Wait Promote (WP) is the best overall. It performs very well when combined with either LS or ED. Conditional Restart (CR) is the second choice for LS. CR also performs well with ED at low loads but poorly at high loads. Wait is the second choice for ED. High Priority (HP) does not do well with either LS or ED.

(c) Although LS performed better at minimizing the number of late transactions, ED is the best choice for minimizing the mean tardy time of commited transactions.

(d) When the IO system is heavily loaded, using real-time transacion priorities to schedule IO requests yields significant performance gains over scheduling IO requests in a FIFO manner.

(e) Using continuous evaluation to implement LS yields better performance when the load is low. Static evaluation achieves better performance when the load is high. Also, static evaluation is less sensitive to error in the runtime estimate than is continuous evaluation.

When the load is an "input step function":

(f) ED is the best priority policy to use. It performs better than LS at the higher load settings for three of the four concurrency control policies.

(g) HP is the best concurrency control policy. It is the best of the four when combined with either ED or LS. The second and third choices for concurrency are CR and WP.

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

1. Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," *Proceedings of the Conference on Very Large Database Systems*, pp. 1-12, VLDB, August 1988.

2. Bernstein, Philip A. and Nathan Goodman, "Timestamp-Based Algorithms For Concurrency Control In Distributed Database Systems," *Proceedings of the Conference on Very Large Databases*, pp. 285-300, IEEE, Montreal, Canada, Oct., 1980.

3. Coffman, Edward G. Jr. and Peter J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.

4. Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM*, vol. 19, no. 11, pp. 624-633, November 1976.

5. Garcia-Molina, Hector, "Using semantic knowledge for transaction processing in a distributed database," *Transactions on Database Systems*, vol. 8, pp. 186-213, ACM, June, 1983.

6. Jensen, E. Douglas, C. Douglass Locke, and Hideyuki Tokuda, "A time-driven scheduler for real-time operating systems," *Proceedings Real-time Systems Symposium*, pp. 112-122, IEEE, 1986.

7. Liu, C.L. and J.W. Wayland, "Scheduling algorithms for multiprogramming in hard real-time environment," *Journal of the ACM*, vol. 20, pp. 46-61, ACM, January, 1973.

8. Sha, Lui, Ragunathan Rajkumar, and John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *Tech. Report*, vol. CMU-CS-87-181, CMU, Pittsburgh, December 1987.

9. Zhao, W., K. Ramamritham, and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *Transactions on Computers*, vol. C-36, pp. 949-960, IEEE, August 1987.

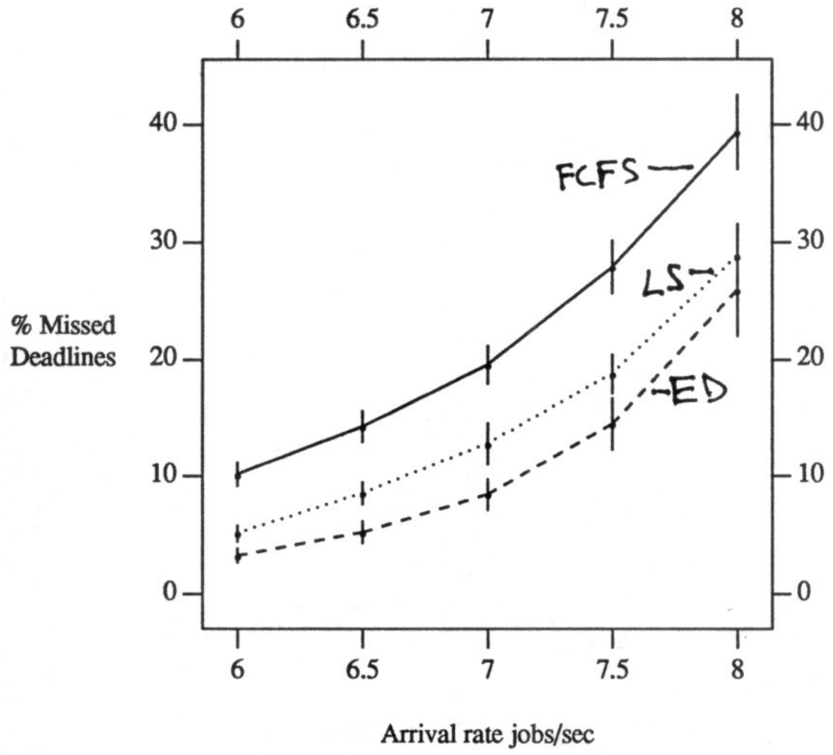

Figure 1. FCFS/W, ED/W, and LS/W.

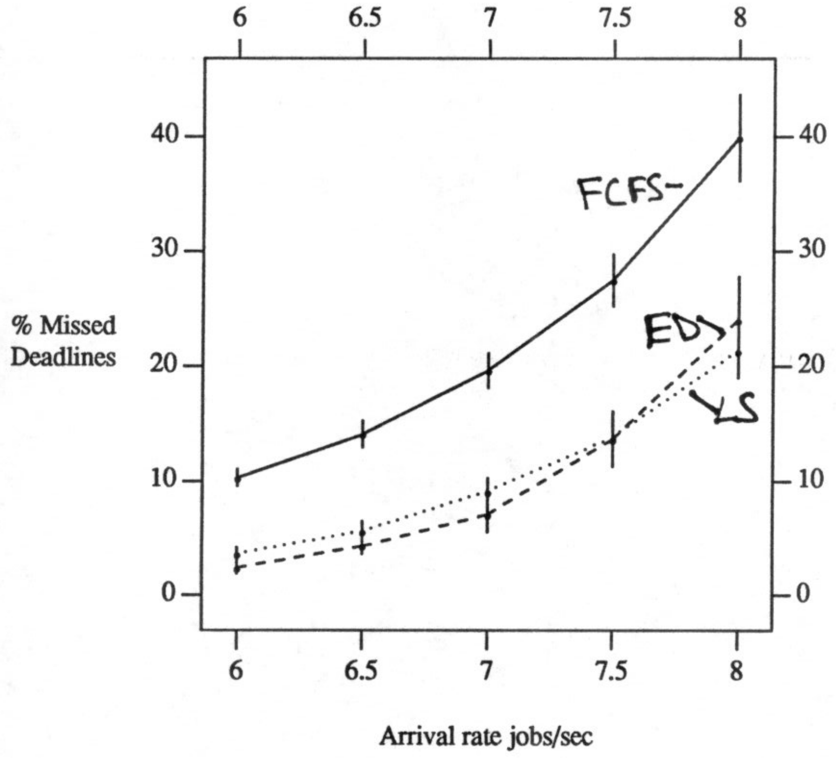

Figure 2. FCFS/WP, ED/WP, and LS/WP.

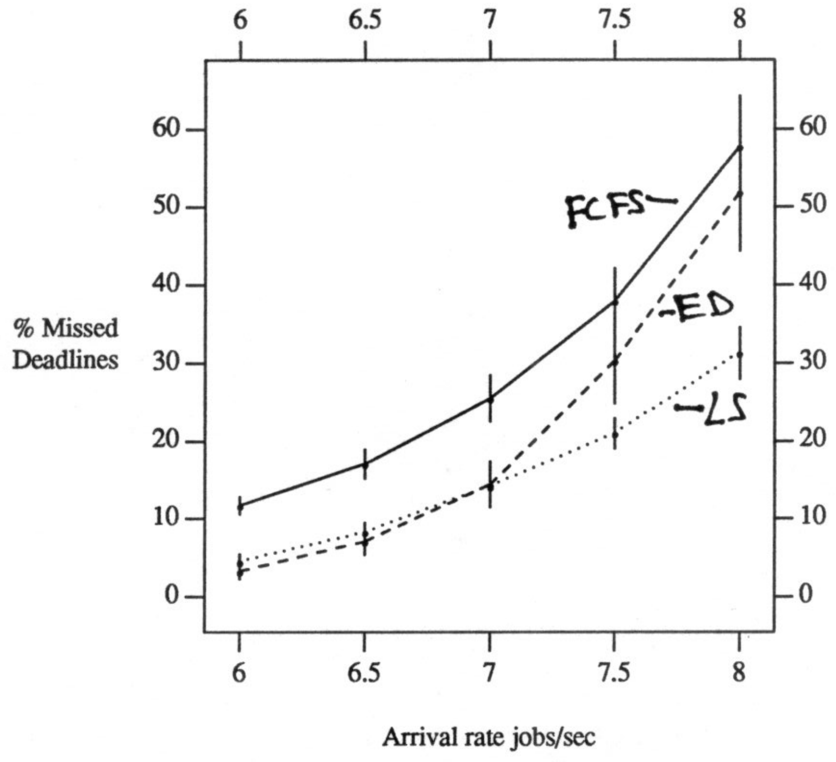

Figure 3. FCFS/HP, ED/HP, and LS/HP.

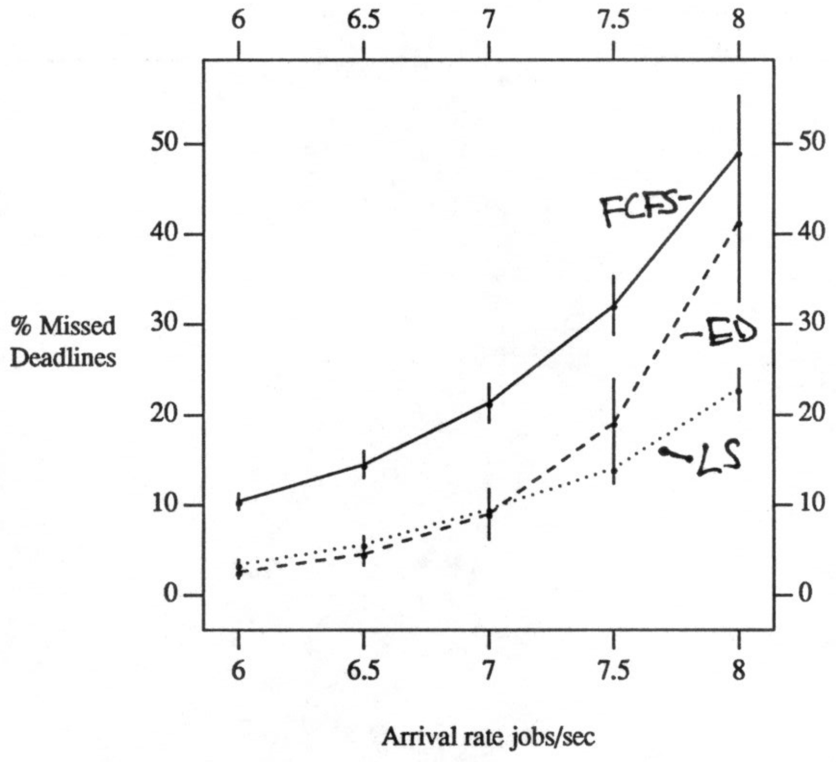

Figure 4. FCFS/CR, ED/CR, and LS/CR.

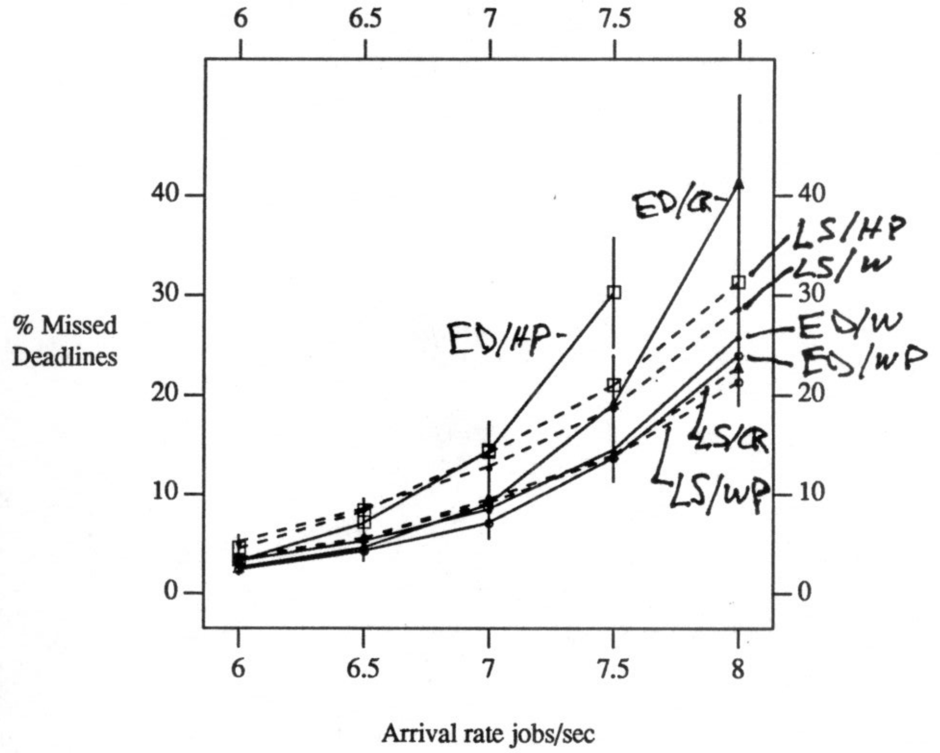

Figure 5. ED vs LS, each paired with W, WP, HP, CR.

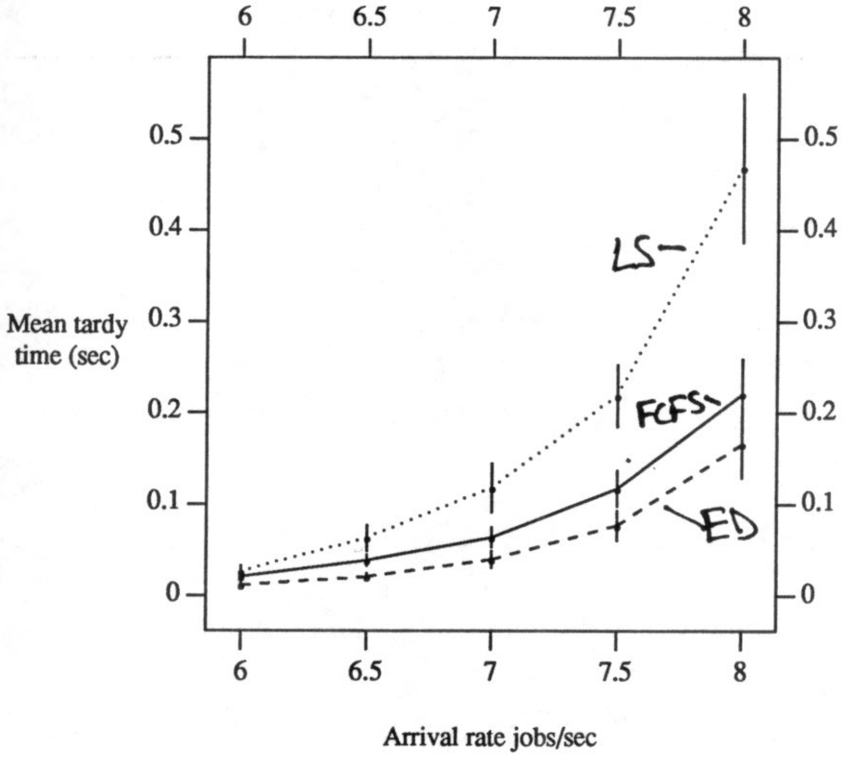

Figure 6. FCFS/W, ED/W, and LS/W.

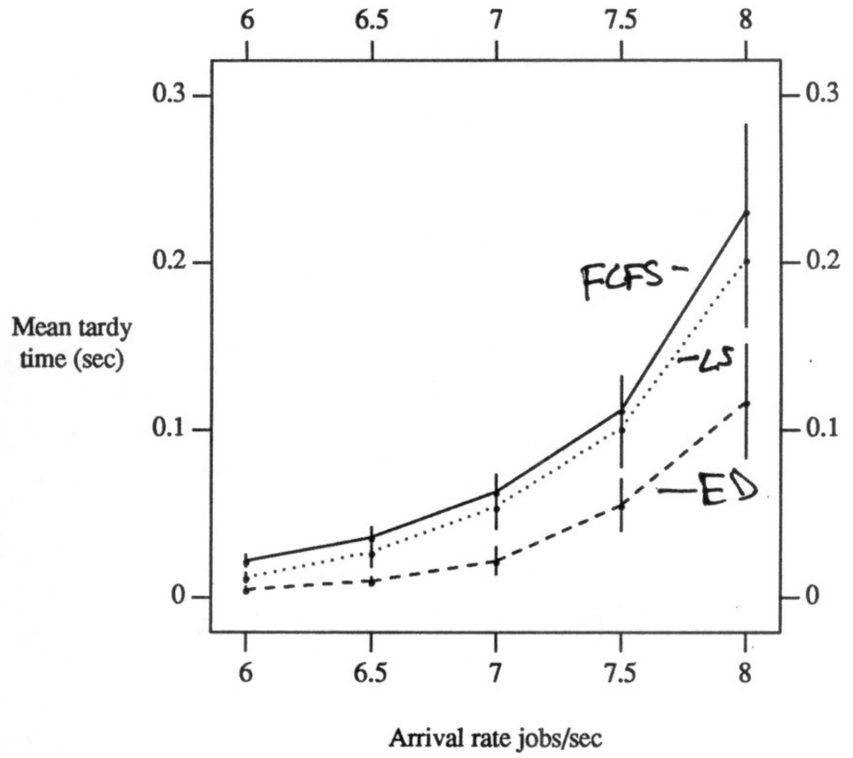

Figure 7. FCFS/WP, ED/WP, and LS/WP.

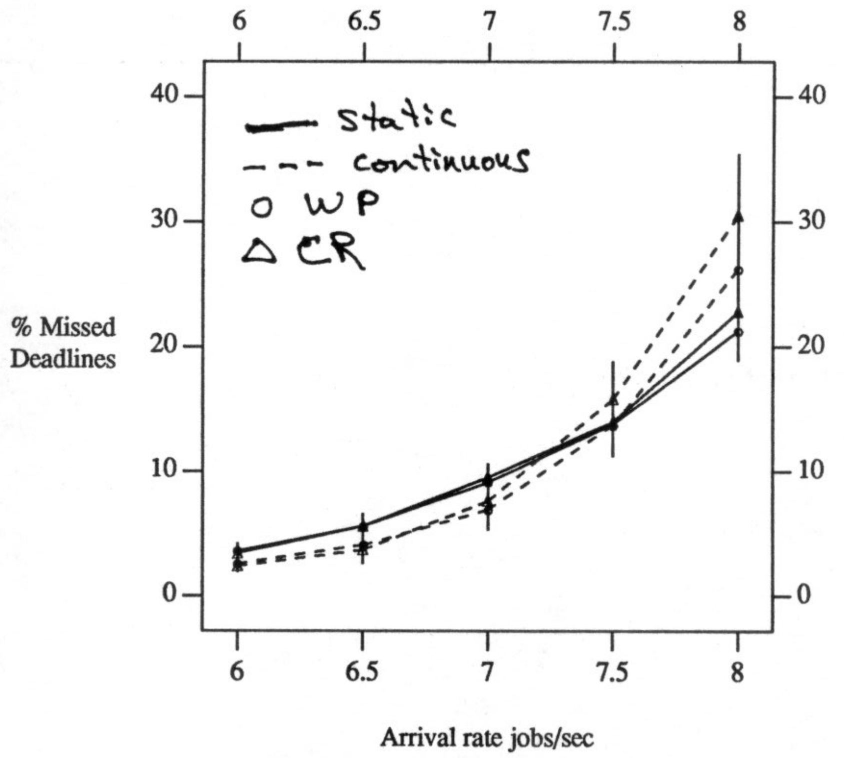

Figure 8. LS/WP and LS/CR under static and continuous evaluation.

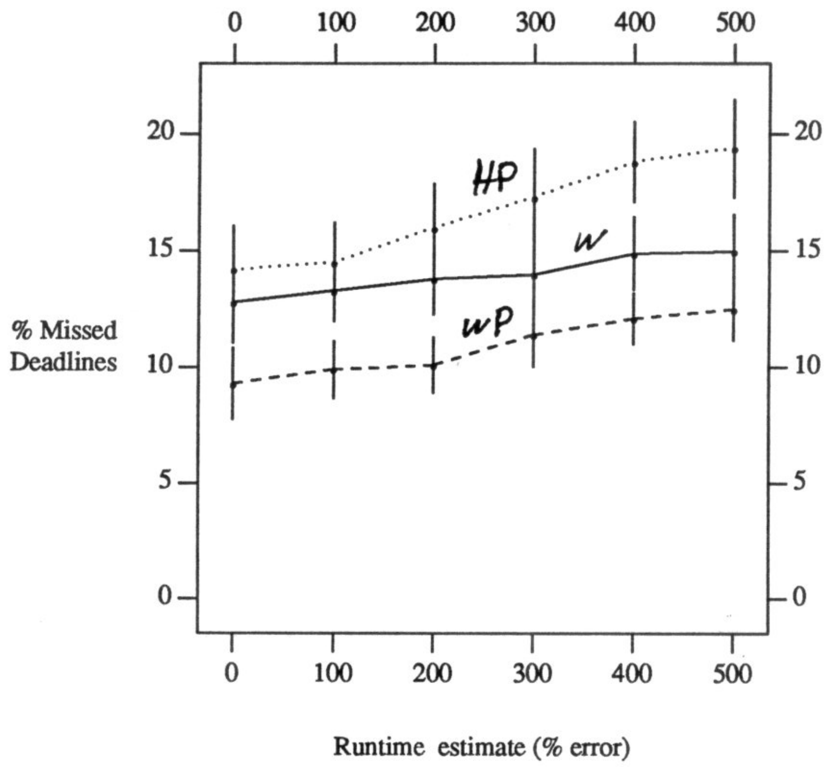

Figure 9. LS/W, LS/WP, and LS/HP under static evaluation.

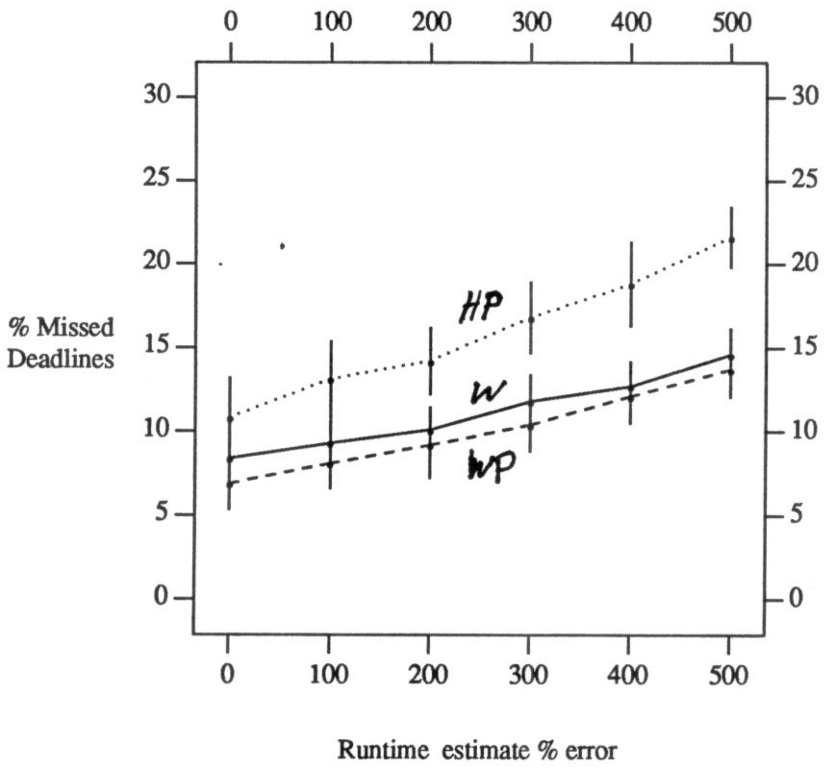

Figure 10. LS/W, LS/WP, and LS/HP under continuous evaluation..

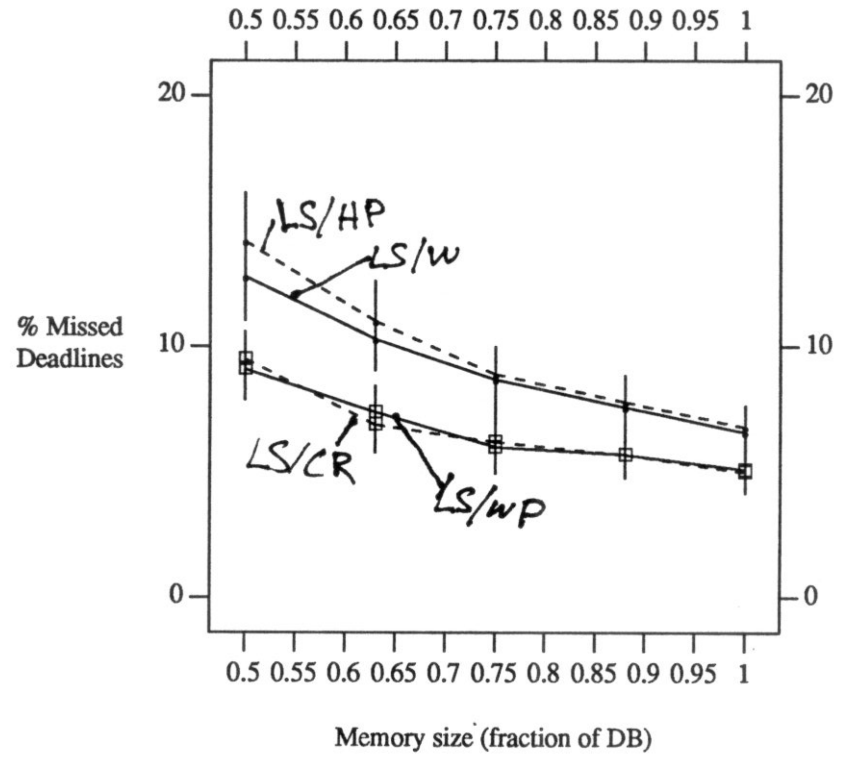

Figure 11. LS/W, LS/WP, LS/HP, and LS/CR.

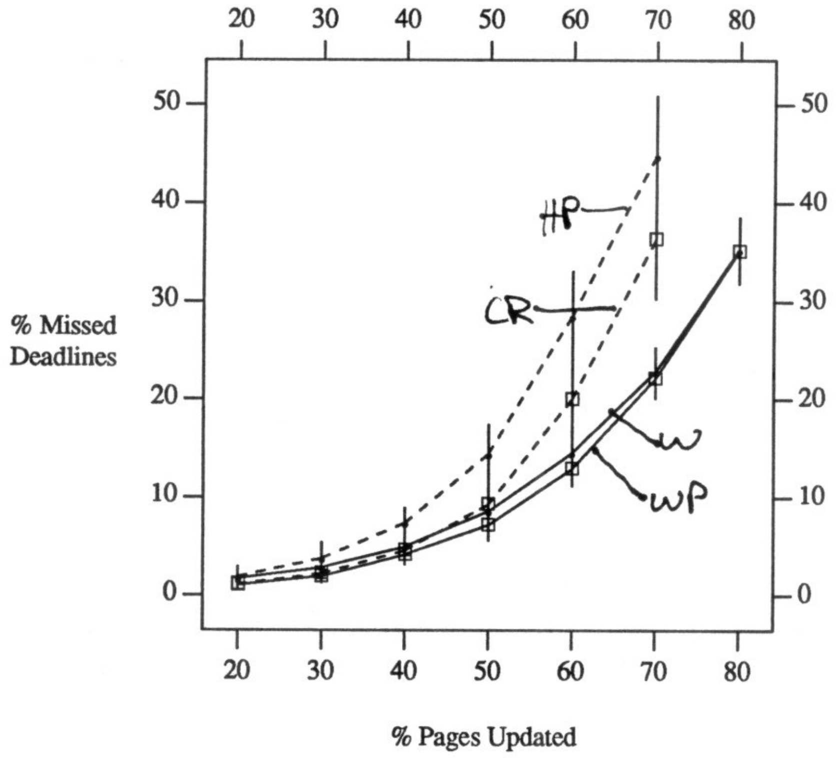

Figure 12. ED/W, ED/WP, ED/HP, and ED/CR.

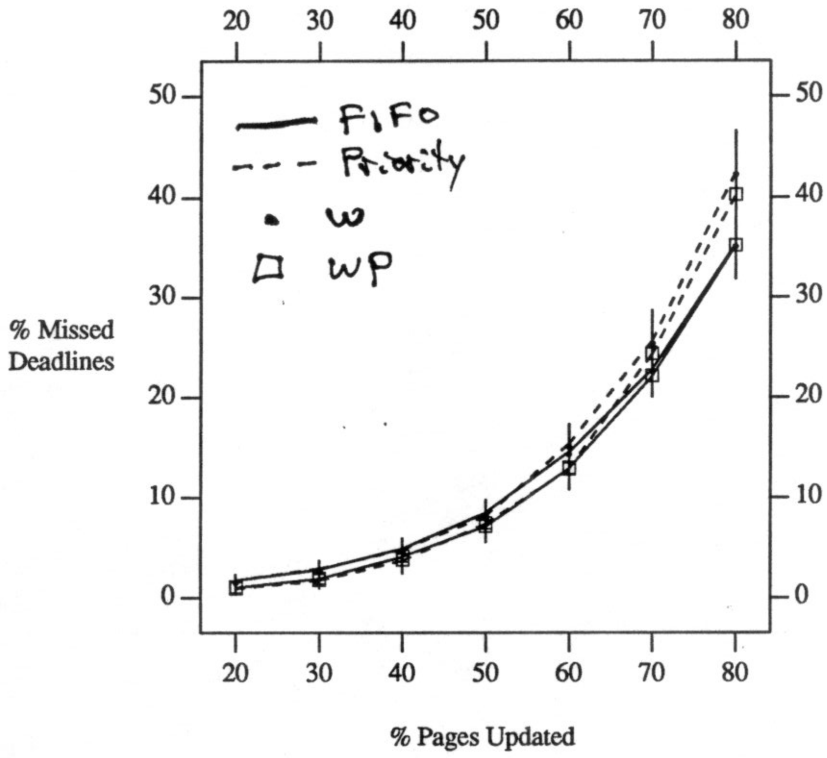

Figure 13. ED/W and ED/WP, each with FIFO and Priority IO scheduling.

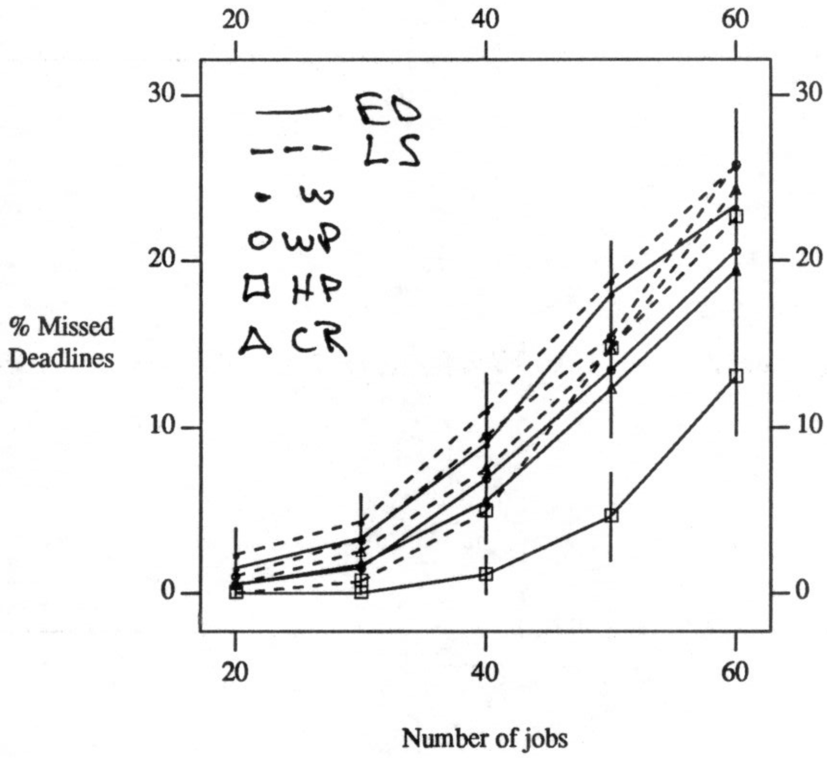

Figure 14. ED and LS, each with W, WP, HP, and CR, and Priority IO scheduling.

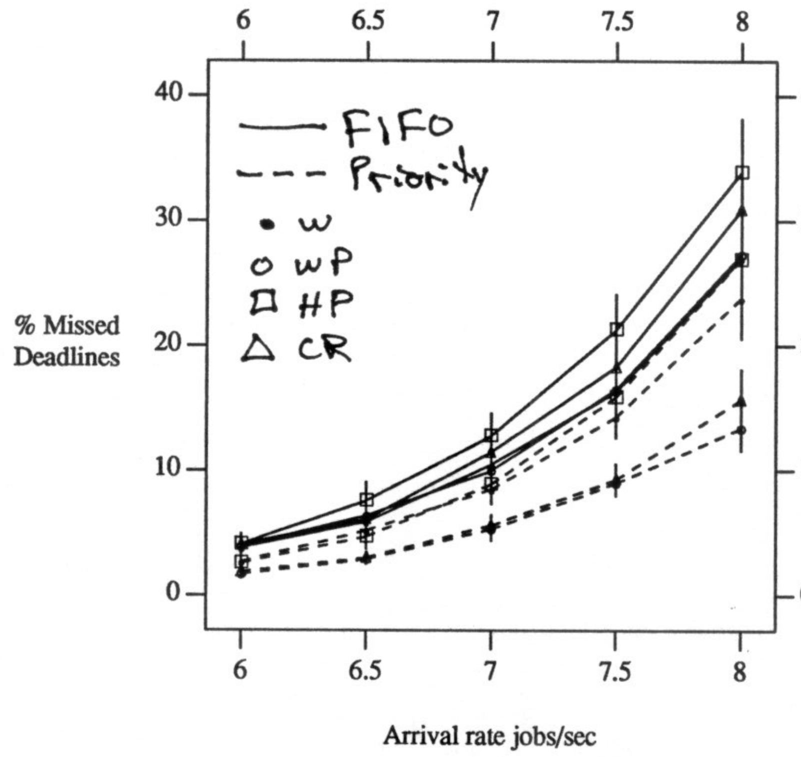

Figure 15. LS with W, WP, HP, and CR, under FIFO and Priority IO scheduling.

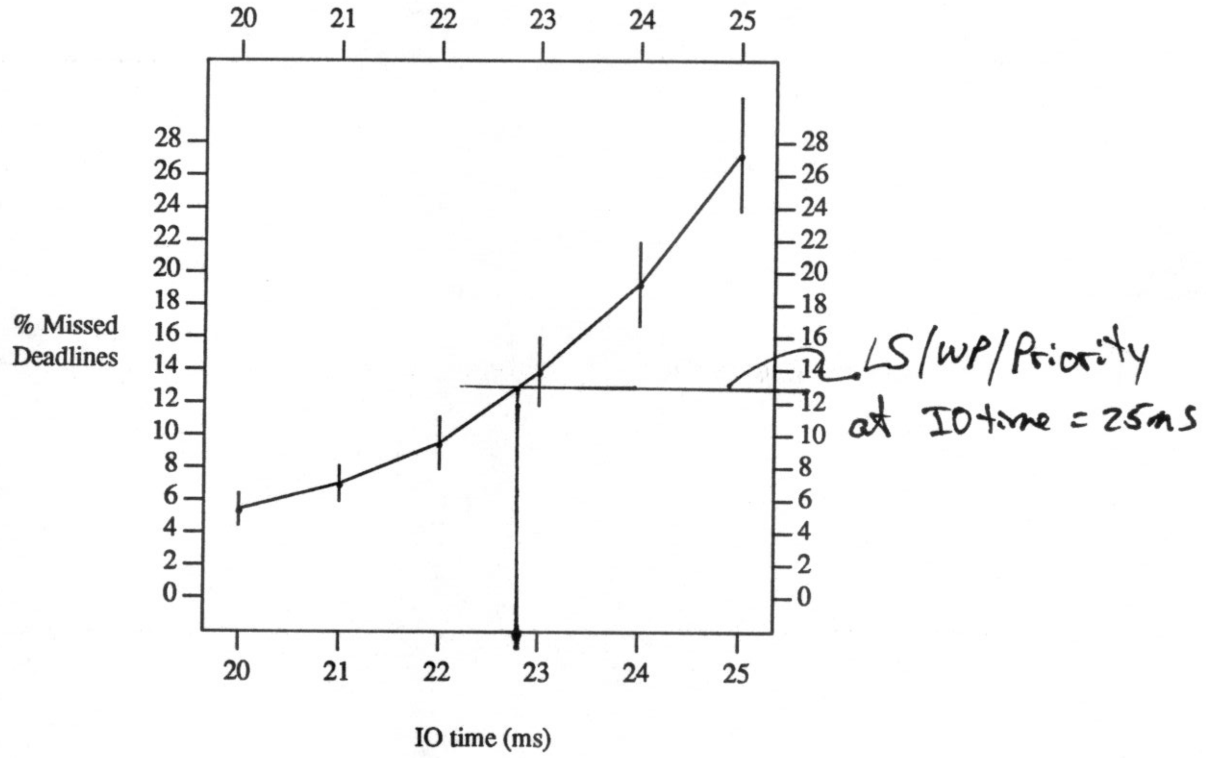

Figure 16. LS/WP.