Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems

S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-77

Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems *

Shenze Chen John A. Stankovic
James F. Kurose Don Towsley

Department of Computer & Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

In this paper, we present two new disk scheduling algorithms for real-time systems. The two algorithms, called SSEDO(for *Shortest Seek and Earliest Deadline by Ordering*) and SSEDV(for *Shortest Seek and Earliest Deadline by Value*), combine *deadline* information and *disk service time* information in different ways. The basic idea behind these new algorithms is to give the disk I/O request with the earliest deadline a high priority, but if a request with a larger deadline is "very" close to the current disk arm position, then it may be assigned the highest priority. The performance of SSEDO and SSEDV algorithms is compared with three other proposed real-time disk scheduling algorithms ED, P-SCAN, and FD-SCAN, as well as four conventional algorithms SSTF, SCAN, C-SCAN, and FCFS. An important aspect of the performance study is that the evaluation is not done in isolation with respect to the disk, but as part of an integrated collection of protocols necessary to support a real-time transaction system. The transaction system model is validated on an actual real-time transaction system testbed, called RT-CARAT. The performance results show that SSEDV outperforms SSEDO; that both of these new algorithms can improve performance of up to 38% over previously-known real-time disk scheduling algorithms; and that all of these real-time scheduling algorithms are significantly better than non-real-time algorithms in the sense of minimizing the transaction loss ratio.

*This work is supported, in part, by the Office of Naval Research under contract N00014-87-K-796, by NSF under contract IRI-8908693, and by an NSF equipment grant CERDCR 8500332.

1 Introduction

Recently, interest in real-time systems research has been growing. In a real-time system, each computational entity (e.g., a task, a process, a thread, or a transaction) has a deadline when submitted to the system. These entities must be scheduled and processed in such a way that they complete by their corresponding deadlines. Generally, two categories of real-time systems can be identified. In a *hard real-time* system, missing a deadline may lead to a catastrophe, while in a *soft real-time* system, missing a deadline may only reduce the 'value' of the entity to the system. In this work we focus on soft real-time systems, in which the computational entities are scheduled to maximize the 'value' they impart to the system, but they are not guaranteed to make their deadlines.

Since performance criteria for real-time systems are quite different from that of conventional systems, and since research on real-time systems is still in its infancy, there are many new and challenging issues raised in designing such a system. One of these challenging issues is real-time I/O scheduling. Because I/O devices are orders of magnitude slower than CPU speeds, the improvement of I/O efficiency is extremely important to the performance of a real-time system. This motivates our interest in examining the real-time disk scheduling problem.

Although extensive work has been done on issues like real-time CPU scheduling [6,23,1,17,29,29,19,8] and real-time communications [18,24,9,15,20,30,22,16,14], interestingly enough, few papers have dealt with the I/O the problem in a real-time environment. Previous work on this form of I/O scheduling for real-time systems can be found in [3] and [7]. Abbott [3] suggested a real-time disk scheduling algorithm called FD-SCAN (*Feasible Deadline SCAN*), which is a variant of the SCAN algorithm. Specifically, FD-SCAN differs from SCAN in the way that it dynamically adapts the scan direction towards the request with the earliest feasible deadline, where a deadline is said to be *feasible* if it is estimated that it can be met. In [7], Carey gives a priority disk scheduling algorithm which is also based on the SCAN algorithm. In this work they only looked at a conventional database system. Therefore, their performance criteria are quite different from ours. However, in order to examine the performance of the priority SCAN algorithm in a real-time environment, we implemented such an algorithm, P-SCAN, and tested it in our experiments. In addition, it is obvious that the *earliest deadline* (ED) algorithm could be used for real-time I/O disk scheduling.

In this paper, we present two new real-time disk scheduling algorithms, SSEDO and SSEDV, and compare their performance with previously known real-time disk scheduling algorithms, ED, P-SCAN, and FD-SCAN, as well as with four conventional algorithms,

SSTF, SCAN, C-SCAN, and FCFS. An important aspect of the performance study is that the evaluation is not done in isolation with respect to the disk itself, but rather as part of an integrated collection of protocols necessary to support a real-time transaction system. The transaction system model is validated on an actual real-time transaction system testbed, called RT-CARAT. The performance measures of interest are the transaction loss probability and the the average response time for committed transactions under different I/O scheduling algorithms. The results show that the SSEDV algorithm outperforms the SSEDO algorithm; that both of these new algorithms can improve performance up to 38% over the three previously suggested real-time disk scheduling algorithms, or up to 53% over the four conventional algorithms; and that all of these real-time scheduling algorithms are significantly better than non-real-time algorithms in the sense of minimizing the transaction loss ratio.

The remainder of this paper is organized as follows: Section 2 describes the real-time transaction system model and its validation. Section 3 introduces the two new real-time disk scheduling algorithms SSEDO and SSEDV, and then discusses the other algorithms examined in this paper. The performance results are presented in Section 4. Section 5 summarizes this paper.

2 The Real-Time Transaction System Model

2.1 The Model Description

Our system model takes an integrated view of real-time transaction performance. While it is our intent to specifically study real-time disk I/O scheduling algorithms, we do so in a complete system setting. The overall metric of interest is to minimize the transaction loss probability, i.e., the probability that a transaction does not meet its deadline; note that this is only partially affected by the disk I/O policy. Consequently, we study 9 disk I/O scheduling algorithms in a system with fixed algorithms for concurrency control, lock conflict resolution, commit processing, CPU scheduling, deadlock detection, deadlock resolution, transaction restart, and transaction wakeup. We now describe the system model in detail.

Since we are interested in the disk I/O scheduling problem for real-time transaction systems, the database is assumed to be very large and disk resident. Specifically, the transaction system is modeled as a closed system which consists of multiple users, a CPU, and a disk for storing the database (Fig.1). All log information is placed on a separate log disk. The log disk does not show up in the figure because its cost is accounted for and subsumed as part of the CPU operation.

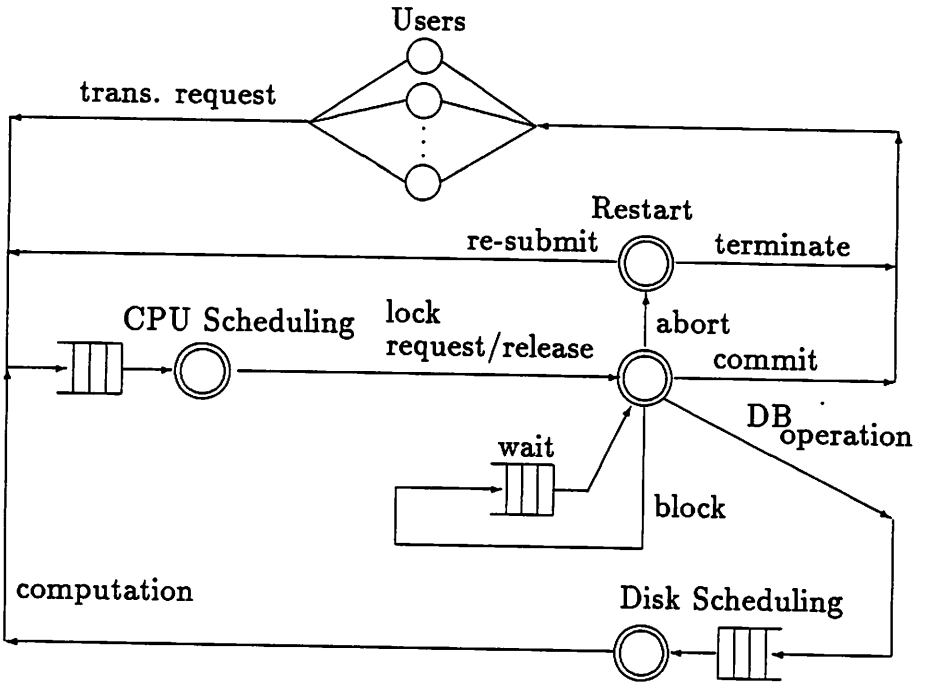


Figure 1: The System Model

In this system model, each user spends a random amount of time in a *think* state before generating a transaction. Each transaction is considered to have the same importance and is assigned a deadline when submitted to the system. If a transaction cannot commit before its deadline, it is said to be lost and is removed from the system immediately. The user who submitted the lost transaction returns to the *think* state. A transaction consists of a random number of operational steps. Each step performs an access to the database, (only if the desired page is not in the main memory buffer), and manipulates the data items retrieved. In some of our experiments, we assume that the number of steps for a transaction is known to the system as soon as the transaction arrives. This assumption is often reasonable since the knowledge of a transaction's length can sometimes be obtained from the compiler or some other transaction preprocessor by counting the transaction's I/O requests.

Generally, we assume that transactions randomly access records and that multiple records are stored on each page of data. We also assume that the size of a page is the same as the size of a disk block. The random access assumption is removed when we examine the locality effect of transactions' accesses. In case of transactions randomly accessing the database, since the database is assumed large, the memory buffer hit ratio is low and therefore a disk I/O is required most of time. This better enables us to investigate the impact of I/O scheduling algorithms when the system is I/O bound.

To maintain the consistency of the database, serializability is enforced by using the *two-phase locking* protocol. Specifically, each page is associated with a lock. Each transaction step needs to request and hold a lock before being allowed to access the database. For any transaction, all locks obtained at each step will be held until it commits or aborts. Two types of locks are identified, shared locks and exclusive locks. A shared lock can be granted to multiple users for inquiry(read) operations, and an exclusive lock is granted to only one user for update(write) operations. A lock conflict may occur when a requested lock has been granted to other transactions, and either the requesting or the holding transaction is to perform an update operation. In order to handle this situation, a lock conflict resolution policy must be defined. The policy we use is as follows:

- **Lock Conflict Resolution**
  When there is a lock conflict, we always let the requesting transaction suspend, i.e., join a queue waiting for the lock. There is no preemption. This is based on the assumption that all transactions are of the same importance so preemption is not warranted. When each transaction carries, in addition to a deadline, a different importance value, preemption has been shown to be a good strategy [12].

The locking scheme used may cause deadlocks. A simple cycle detection deadlock detection algorithm is employed in our model.

When a deadlock is detected, a victim is selected to be aborted according to some rule. Our resolution strategy follows a simple rule:

- **Deadlock Resolution**
  If a deadlock is detected, then the requesting transaction is aborted. In this case, the transaction releases all resources held and proceeds according to the restart strategy.

The restart strategy is:

- **Restart Policy**
  A transaction aborted from a deadlock is restarted as long as its deadline has not yet expired. Transactions aborted for other reasons, such as user termination, arithmetic overflow, and execution exceptions, are removed from the system and return to the *think* state. In the former case, a smarter policy might be to restart a transaction only if its estimated minimum processing time is less than the remaining time before missing its deadline. For the latter case, a more sophisticated strategy might restart a transaction if it aborted due to arithmetic overflow and/or execution exception, after doing some exception handling.

The transaction wakeup and commit strategies are as follows:

- **Wakeup Strategy**

  When a transaction commits, aborts, or misses its deadline, it releases all of its resources. If some of these resources (pages) are needed by more than one other transaction, then we select the one with earliest deadline among all waiting transactions to wakeup, and grant it the associated lock.

- **Transaction Commit**

  When a transaction commits, it releases all its resources after completion of logging and goes back to the *think* state. If there are any dirty pages in the memory buffer, they are flushed to the disk. Obviously, these flushing operations do not deserve a high priority since the transaction issuing these operations has already committed. On the other hand, from the buffer manager's standpoint, flushing operations should complete as soon as possible to free up buffer space. This means that they should not have a low priority. In a transaction system, since committing is more important than the buffer, it seems that flushing writes should get a low priority. However, in the simulations we do not degrade the priority of flushing writes because such a strategy would only be applicable to transaction systems, and our new disk I/O algorithms are more general and apply also to non-transaction real-time systems.

- **Buffer Manager**

  The buffer manager is not explicitly modeled in order to simplify the simulation programs. Instead, a *buffer hit probability*, which is assumed to be 0.1, is used to determine whether a disk access is required.

CPU scheduling is an important issue when dealing with real-time transaction systems. We chose the following algorithm.

- **CPU Scheduling**

  The *earliest deadline* (ED) algorithm is used for CPU scheduling. As mentioned in the literature [2], a major drawback of this algorithm is that it may assign the highest priority to a transaction that has already missed or is about to miss its deadline. To partially overcome this weakness, we propose a modified ED algorithm which, instead of scheduling according to a transaction's deadline, schedules by the *step deadline* of a transaction given the number of steps of a transaction is known to the system at the time of submission. Specifically, let L, a, and n be a transaction's (absolute) deadline, arrival time, and the number of operational steps. A simple way to assign a step deadline for step i is to set

$$step\_deadline(i) = a + \frac{i}{n}(L - a) \quad (1)$$

Note that under this policy, the earlier the step, the smaller the step deadline, and therefore, the higher the priority for the transaction when executing its earlier steps. Also note that the deadline for the last step is always equal to the transaction's deadline. One consideration for using the step deadline is that if two transactions have the same deadline, but one has more steps than the other, then the longer transaction will have higher priority for its earlier steps. We tested the effect of using step deadline on both CPU and I/O scheduling. The results show that this strategy does indeed improve the system performance (see 4.2.4).

In summary, in order to achieve good performance in the sense of minimizing the transaction loss ratio, there is a collection of algorithms that must be developed including CPU scheduling, commit processing, I/O disk scheduling, concurrency control, lock conflict resolution, deadlock resolution, restart, and wakeup. Since our main goal is to examine the impact of I/O scheduling in a real-time transaction system, many of the algorithms used in our study are typical or sometimes simple, but suitable for a real-time system.

2.2 Model Validation

In order to validate our basic model of an integrated set of protocols required to perform real-time transaction processing, we conducted a series of experiments on an actual real-time database testbed, called RT-CARAT [12,13]. Given the the same workloads and algorithm settings (including CPU scheduling, lock conflict resolution, deadlock resolution, restart strategy, and commit processing), our simulation results concerning the basic simulation model (which uses the FCFS disk service strategy) match quite well with that of the testbed. On the other hand, it was not possible for us to validate the simulation results comparing all the nine I/O disk scheduling algorithms because the disk controller in the testbed could not be modified.

RT-CARAT is currently running on a VAX Station II/GPX under the VMS operating system. The database consists of 3000 pages, with each page containing 6 records. The number of users is fixed at 8. Each user continuously generates transactions with a fixed number of operational steps. When a previous transaction commits or aborts, a user submits his next transaction immediately. In the validation runs there was no *thinking time* between transaction submissions. On RT-CARAT, each operational step accesses 4 records, which are randomly located in the database, and therefore up to 4 I/O requests may be generated for each step. In our model, since we assume each operational step only

needs to access the database once, we set the total number of steps for each transaction to four times of that in RT-CARAT so that a transaction will generate the same number of I/O requests. Although RT-CARAT implements various real-time CPU scheduling algorithms, conflict resolution protocols, deadlock resolutions, and wakeup strategies, the I/O disk scheduling algorithm used, is the basic strategy employed by the underlying VMS operating system and its disk controller. In fact, since the VAX Station II/GPX uses an RQDX3 controller and the RD54 disk drive [27], both of which are very early models, I/O requests are simply served in an approximate first come first serve fashion.

In setting deadlines for transactions we follow two steps: first, a selected workload is placed on RT-CARAT in a non-real-time database setting and the mean transaction response time and its standard deviation are measured. In other words we make a preliminary run to determine the transactions' mean response time (*avg_resp*) and its standard deviation (*std_devi*), assuming the transactions have no deadlines. This is done with a basic set of protocols that do not use deadlines to make decisions. Then a base deadline value (*base_line*) is calculated by

$$base\_line = avg\_resp - std\_devi$$

and each transaction's deadline is randomly selected from the range $[base\_line, f*base\_line]$, where f is a factor that enables us to test a wide range of deadlines. We set f to 3 in the validation experiments.

In order to compare the performance of the simulation model and the testbed, the database configuration and the collection of algorithms supporting a real-time transaction system are configured to be the same on both. The system overhead, such as context switching, interprocess communication, locking mechanism, and logging, are considered and assimilated into the computation time for each step in our model. In order to compare their performance, the step computation time in our model is tuned so that both systems have approximately the same disk utilization in the non-real-time cases. The statistics on RT-CARAT show that the average CPU time for each step is approximately 41ms and that is the value we used in our simulation experiments.

The comparison of the performance of RT-CARAT and our simulation model is shown in Fig.2. The X-axis gives the number of steps for each transaction in our simulation model. In Fig.2(a) we see that transaction loss ratio in our simulation model is slightly less than that in RT-CARAT when the system is lightly loaded, and slightly higher than that of RT-CARAT when the workload increases. One reason for this is that the variance of transaction response time in our model is less than that in RT-CARAT when the load is low, and it increases faster than its counterpart in RT-CARAT as the load increases.

In Fig.2(b) we see that average transaction response time in both models is the same. In Fig.2(c), we see that the average disk utilizations are nearly the same on both systems. On the other hand, CPU utilization drops faster in our simulation model than in the testbed.

3 Description of Various Disk Scheduling Algorithms

In this section, we describe all nine disk scheduling algorithms examined in this paper. We first introduce the two new real-time disk scheduling algorithms, SSEDO and SSEDV. We then describe 3 other real-time disk scheduling algorithms, ED, P-SCAN and FD-SCAN. It should be noted that the *earliest deadline strategy* (ED) is a special case of each of the two new real-time algorithms. Finally, we very briefly describe four traditional disk scheduling algorithms.

3.1 Two New Real-Time Disk Scheduling Algorithms

3.1.1 Motivation

In a real-time system, timing constraints are important to consider when making scheduling decisions. For real-time task scheduling, Towsley et.al.[26] have proved that the *earliest deadline* policy is optimal in the sense of minimizing the loss probability among all policies which are independent of task service times, given these task service times are *i.i.d.* and exponentially distributed. For disk scheduling, however, the independence and exponential assumptions are no longer valid.

For example, disk service times of I/O requests depend on the scheduling discipline employed. A simple example can be found in considering the *First Come First Service* (FCFS) and *Shortest Seek Time First* (SSTF) disciplines, where the disk service time under the former discipline is believed longer than that of the later [10]. Second, the successive service times are not independent since a request's service time depends on the current disk arm position (the cylinder address of the previously served request). Third, disk service times do not follow the exponential distribution, in general. From these observations, we can expect that a proper disk scheduling algorithm for a real-time system should take into account not only the time constraint but also the disk service time.

3.1.2 The SSEDO and SSEDV Algorithms

Based on the above considerations, we propose two new real-time disk scheduling algorithms, SSEDO (for *Shortest Seek and Earliest Deadline by Ordering*) and SSEDV (for *Shortest Seek and Earliest Deadline by Value*), for a single disk device.

Let

r_i : be the I/O request with the i-th smallest deadline at a scheduling instance;

d_i : be the distance between the current arm position and request r_i's position;

L_i : be the absolute deadline of r_i.

Note that an I/O request's deadline can be the same as that of the computational entity (e.g., a transaction in our model) issuing the request, or can be calculated by using the entity's deadline and other information, as was shown for the step deadline in section 2.1.

The two algorithms maintain a queue sorted according to the (absolute) deadline, L_i, of each request. A window of size m is defined as the first m requests in the queue, i.e., the window consists of m requests with smallest deadlines. Hence we may also refer to these two algorithms, SSEDO and SSEDV, as window algorithms. The advantage of defining a window can be seen in section 3.1.3.

- **SSEDO Algorithm**

  At a scheduling instance, the scheduler selects one of the requests from the window for service. The scheduling rule is to assign each request a weight, say w_i for request r_i, where $w_1 = 1 \leq w_2 \leq ... \leq w_m$ and m is the window size, and to choose the one with the minimum value of $w_i d_i$. We shall refer to this quantity $w_i d_i$ as the *priority value* associated with request r_i. If there is more than one request with the same priority value, the one with earliest deadline is selected. It should be clear that for any specific request, its priority value varies at each scheduling instance, since d_i, r_i's position with respect to the disk arm position, is changing as the disk arm moves.

  The idea behind the above algorithm is that we want to give requests with smaller deadlines higher priorities so that they can receive service earlier. This can be accomplished by assigning smaller values to their weights. On the other hand, when a request with large deadline is "very" close to the current arm position (which means less service time), it should get higher priority. This is especially true when a request is to access the cylinder where the arm is currently positioned. Since there is no seek time in this case and we are assuming the seek time dominates the service time, the service time can be ignored. Therefore these requests should be given the highest priority. There are various ways to assign these weights w_i. In our experiments, the weights are simply set to

$$w_i = \beta^{i-1} \quad (\beta \geq 1) \qquad i = 1, 2, ..., m.$$

where β is an adjustable scheduling parameter. Note that w_i assigns priority only on the basis of the *ordering* of deadlines, not on their absolute or relative *values*. In addition, when all weights are equal ($\beta = 1$), we obtain an approximate SSTF algorithm which converges to pure SSTF as the window size becomes large. When the window size is equal to one, the algorithm is the same as the ED algorithm. In the following section, we will see that the performance of the system is improved dramatically when a window size of three or four is chosen.

- **SSEDV Algorithm**

  In the SSEDO algorithm described above, the scheduler uses only the ordering information of requests' deadline and does not use the differences between deadlines of successive requests in the window. For example, suppose there are two requests in the window, and r_1's deadline is very close but r_2's deadline is far away. If r_2's position is "very" close to the current arm position, then the SSEDO algorithm might schedule r_2 first, which may result in the loss of r_1. However, if r_1 is scheduled first, then both of r_1 and r_2 might get served. On the other extreme, if r_2's deadline is almost the same as r_1's, and the distance d_2 is less than d_1 but greater than d_1/β, then SSEDO will schedule r_1 for service and r_2 will be lost. In this case, since there could be a loss anyway, it seems reasonable to serve the closer one (r_2) for its service time is smaller. Based on these considerations, we expect that a more intelligent scheduler might use not only the deadline *ordering* information but also the deadline *value* information for decision making. This leads to the following algorithm: associate a priority value of $\alpha d_i + (1-\alpha) l_i$ to request r_i and choose the request with the minimum value for service, where l_i is the *remaining life time* of request r_i, defined as the length of time between the current time and r_i's deadline L_i and $\alpha (0 \leq \alpha \leq 1)$ is a scheduling parameter.

  Again when $\alpha = 1$, this approximates the SSTF algorithm, and when $\alpha = 0$, we obtain the ED algorithm.

A common characteristic of the SSEDV and SSEDO algorithms is that both consider *time constraints* and *disk service times*. Which part plays a greater role in decision making can be adjusted by tuning the scheduling parameters α or β, depending on the algorithm.

3.1.3 Scheduling Costs of SSEDO and SSEDV Algorithms

For the two algorithms defined above, since the priority value of each request in the window needs to be evaluated at every scheduling point, we might hope to estimate their scheduling

cost. However, the overhead involved in scheduling should be "small" since we don't want the disk to be idle for a long time waiting for the scheduling algorithm to finish, as this might dramatically increase I/O response time and therefore the transaction loss ratio.

From an implementation point of view, the ED queue required for the SSEDO and SSEDV algorithms can be maintained by the operating system or the device controller when requests arrive at a cost $O(\log n)$, where n is the number of requests in the queue. For a fixed window size m (usually $m = 3$ is enough as shown in the next section), the priority value calculation cost is $O(1)$. That is why we define a window, rather than compute all the requests in the queue which will result in a cost of $O(n)$, and this is motivated by the approximate calculations used in [11]. Since the $O(\log n)$ queue maintenance cost at arrival instances can be performed in parallel with disk operations, and since there is only a $O(1)$ cost at scheduling instances, scheduling cost can be kept low. In addition, the $O(1)$ priority value computation cost can be further reduced by scheduling the next request while performing a disk service. This is feasible since the cylinder address (and, therefore, the service time) of the request under service is known. Hence, the seek distance and remaining lifetime of all the requests waiting in the window can be calculated in advance. The only exception is when the transaction in service misses its deadline while the service is proceeding. In this case, the resulting arm position is unpredictable. This will invalidate the pre-calculated distance and the remaining lifetime calculations, and necessitate a rescheduling. A more intelligent scheduler might test if a request will miss its deadline before providing the service, since disk service time is predictable. In this way, only those requests which can successfully complete their service will be selected to use the disk. This can avoid the above drawbacks and allow full parallelism of disk scheduling and disk service.

3.2 Other Real-Time Disk Scheduling Algorithms

Three other real-time disk scheduling algorithms, ED, P-SCAN and FD-SCAN, have been suggested in the literature.

3.2.1 ED Algorithm

In the ED algorithm, I/O requests are served based on their deadlines. The request with an earliest deadline has the highest priority.

3.2.2 P-SCAN Algorithm

The priority SCAN (P-SCAN) strategy is based on the idea suggested by Carey [7]. Specifically, all requests in the I/O queue are divided into multiple priority levels. The SCAN algorithm is used within each level, which means that the disk serves any requests that it passes in the current served priority level until there are no more requests in that direction. On the completion of each disk service, the scheduler checks to see whether a disk request of a higher priority is waiting for service. If found, the scheduler switches to that higher level. In this case, the request with shortest seek distance from the current arm position is used to determine the scan direction. It is observed in [7] that a small number of priority levels is preferred in order to provide reasonable I/O performance. How to assign priority to a I/O request is an interesting question and which is not explained in [7]. In our experiments, all I/O requests are mapped into three priority levels according to their deadline information. Specifically, we assume transactions' relative deadlines are uniformly distributed between LOW_DL and UP_DL, where LOW_DL and UP_DL are lower and upper bounds for transaction deadline settings. If a transaction's relative deadline is greater than $(LOW\_DL + UP\_DL)/2$, then it is assigned the lowest priority. If the (relative) deadline is less than $(LOW\_DL + UP\_DL)/4$, then the transaction receives the highest priority. Otherwise the transaction is assigned a middle priority. This strategy is shown to be better than the *one-third* strategy which evenly divides the deadline range into 3 intervals and maps each interval into a priority level.

3.2.3 FD-SCAN Algorithm

Recently, Abbott [3] proposed another variant of the SCAN algorithm, called FD-SCAN. In FD-SCAN, the track location of the request with earliest feasible deadline is used to determine the scan direction. A deadline is *feasible* if we estimate that it can be met. Determining the feasibility of a request's deadline is simple since once the current arm position and the request's track location are known, its service time can be determined. A request's deadline is feasible if it is greater than the current time plus the request's service time. At each scheduling point, all requests are examined to determine which has the earliest feasible deadline. After selecting the scan direction, the arm moves toward that direction and serves all requests along the way.

3.2.4 Scheduling Costs of P-SCAN and FD-SCAN Algorithms

As stated before, scheduling cost is an important factor for on-line scheduling algorithms in a real-time environment. Unfortunately, real-time variants of the SCAN algorithm may

suffer from the cost standpoint, since at each scheduling point, they need to check all requests to determine the scan direction. For P-SCAN, this problem can be alleviated by maintaining a separate queue for each priority level at the cost of a higher maintenance overhead. At each scheduling point, the scheduler just checks whether there is any request waiting in some queue with a higher priority. For FD-SCAN, however, overhead appears to be a problem, since the scheduler must check whether there were any new arrivals carrying the earliest feasible deadline while serving the last request. Even if there are no such new arrivals, it still needs to check whether the previous target request is feasible, and if not, a new direction must be determined. Although this can be performed in parallel with the disk service, there is a possibility that a new request with earliest deadline may arrive after having made the decision. The scheduling cost is $O(n)$ for FD-SCAN in the worst case, where n is the number of requests waiting for service.

3.3 The Four Classical Disk Scheduling Algorithms

The four classical scheduling algorithms described below are well-known. They have been discussed extensively in the literature. Here we simply list them for ease of reference.

- **FCFS**

  This is the simplest strategy in which each request is served in a first-come-first-served fashion.

- **SCAN**

  This is also known as the *elevator* algorithm in which the arm moves in one direction and serves all the requests in that direction until there are no further requests in that direction. The arm then changes its scan direction and repeats the operations.

- **C-SCAN**

  The circular SCAN algorithm works in the same way as SCAN except that it always scans in one direction. After serving the last request in the scan direction, the arm returns to the start position (typically an edge of the disk) without servicing requests and then begins scanning again.

- **SSTF**

  The SSTF, for *shortest seek time first*, algorithm simply selects the request closest to the current arm position for service.

A common feature of all these classical scheduling algorithms is that none of them takes the time constraint of requests into account. As we shall see this results in poor performance of these classical algorithms in real-time systems.

4 Performance Comparisons of Various I/O Scheduling Algorithms

In order to compare the performance of various disk scheduling algorithms in an integrated real-time transaction system as described in Section 2, we conducted a series of simulation experiments on the nine algorithms. In section 4.1, we first describe the system parameter settings for our experiments, including: system configuration, transaction characteristics, deadline settings, and the parameters for the two new algorithms, SSEDO and SSEDV. The experimental results are given in section 4.2. In particular, we study the performance over a wide range of workloads, assess the two new algorithms' sensitivity to window size and algorithm parameters, determine the impact of scheduling via step deadlines, examine the system behavior by varying transaction deadline settings and read probability, investigate the locality of transaction accesses, and report the importance of real-time scheduling by comparing real-time and non-real-time scheduling.

4.1 System Parameter Settings

4.1.1 System Configuration

In our model, the disk has 1000 tracks. The database consists of 6000 pages, which are uniformly distributed on the disk with each page corresponding to a disk block. Disk service time is defined by,

$$S = X_s + X_r + X_t \tag{2}$$

where X_r is the rotation latency which is uniformly distributed among $[0, 16.7]$ milliseconds, X_t is the transfer time which is considered constant and equal to 0.8 ms, and X_s is the seek time defined by,

$$X_s = \begin{cases} a + b\sqrt{i} & i > 0; \\ 0 & i = 0; \end{cases} \tag{3}$$

where a is the arm moving acceleration time (8 ms), b is the seek factor (0.5 ms), and i is the number of tracks for the arm to move [21,5]. The average disk service time is 25 ms.

4.1.2 Transaction Characteristics

Transaction characteristics and load to the system are defined as follows: the number of users, which limits the maximum number of transactions in the closed system, ranges from 4 to 20. Each user may think a random of time, which comes from an exponential distribution with mean 1 second, before generating a transaction. A transaction consists

of a random number of operational steps which is uniformly distributed between 1 and 20. Each step needs to access the database once and is followed by a manipulation of those data items fetched. The computation time of each step is assumed to be 15 ms, and the time needed to abort a transaction is 5 ms. With these parameter settings, the workloads we use show a I/O bound characteristic, which can satisfy our goal of examining the disk scheduling algorithms.

Further, each transaction step may perform a read or an update operation to the database. While a transaction is progressing, it brings those pages desired into a main memory buffer and does a corresponding inquiry or update there. In the case of update operations, the updated pages are also written on a separate log disk. When a transaction commits, all of its dirty pages are flushed to disk. In our model, the main memory buffer is assumed large enough to accommodate all of the pages desired by currently active transactions. In most of our experiments, the read probability, p_r, is set to 0.8, since most of time users are inquiring the database. We also investigate the effect of varying the read probability from 0 to 1. To examine the effect of locality of a transaction's accesses, we define the *probability of sequential access*, p_s, to be the probability that a transaction step is to access the same cylinder as that of its previous step. There is no locality assumed among different transactions. The p_s is usually set to 0 except when we study the locality effect in section 4.2.7, where the p_s is varied from 0 to 0.8.

4.1.3 Deadline Settings

The deadline setting for each transaction in our model depends on the system load and a transaction's length. The system load can be characterized by the number of users in the closed system(with the mean thinking time fixed), and the transaction length corresponds to the number of steps. Specifically, we roughly estimate a transaction's minimum system time by,

$$T_{min} = (CPU\_Time + I/O\_Time) * Num\_Steps$$

where CPU_Time and I/O_Time are the estimated average CPU and disk service time for each step under FCFS strategy. Then the transaction deadline is set by,

$$Trans\_Deadline = T_{min} * \eta$$

where η is a *r.v.* drawn from a uniform distribution on $[DL\_Factor, f * DL\_Factor]$. The DL_Factor is a *deadline factor* selected proportional to the number of users in the system, $DL\_Factor = k * Num\_Users$. In most of our experiments, k is equal to 0.25 and f is 4. However, we also examine the case where transactions' deadlines are very tight and/or very loose by varying DL_Factor.

The deadline setting for each I/O request is inherited from that of the transaction issuing the request in most of experiments. However, several experiments are performed by using a step deadline for each I/O request as will be discussed in section 4.2.4.

4.1.4 Parameters for SSEDO and SSEDV

The scheduling parameters, α and the β, for SSEDO and SSEDV algorithms are set to 2 and 0.8 in most cases. One of our experiments shows the sensitivity of changing α and β under two different workloads. Another experiment gives the results for varying the window size while other parameters are the same. The results show that a window size of 3 or 4 is satisfactory, and therefore usually the window size m is set to 3.

4.2 Simulation Results

In this section, we report our experimental results. The five real-time disk scheduling algorithms, SSEDO, SSEDV, ED, P-SCAN, FD-SCAN, and the four conventional algorithms, SSTF, SCAN, C-SCAN, and FCFS, are examined in these experiments. Results of each experiment is averaged over 20 runs. In each run 1050 transactions are executed. 95% *confidence intervals* are obtained by using the method of independent replications. Confidence interval widths are less than 11% of the point estimates of the loss probability in all cases (these intervals are not shown in our figures in order to clearly show the results). For each run, the execution of the first 50 transactions are considered the transient phase and is excluded from our statistics. In order to avoid congested plots, the nine curves are divided into two groups, classical algorithms and real-time algorithms, and plotted separately. The curve for FD-SCAN algorithm is shown in both plots to provide the basis of comparison. In the following, all transactions and I/O requests are scheduled according to the transaction deadline except where explicitly indicated.

4.2.1 Resource Utilizations and I/O Queue Length

By using those parameter settings specified in 4.1, the CPU and disk utilizations for some workloads under several scheduling algorithms are provided in Table I and II, respectively, and the mean I/O queue length is provided in Table III. Other algorithms fall between those shown in these tables. From these results, we can see there are only slight differences in CPU, disk utilizations and I/O queue lengths due to the different algorithms, except for the high load case where differences in CPU utilization begin to show up. However, we will show later that if we increase the step computation time, the I/O queue length may differ depending on the CPU and disk scheduling algorithms.

Table I: CPU Utilization

Users	FCFS	SCAN	SSTF	FD-SCAN	ED	SSEDV
4	0.3661	0.3721	0.3729	0.3746	0.3685	0.3736
8	0.5004	0.5592	0.5600	0.5284	0.4914	0.5261
20	0.5176	0.6539	0.6557	0.6097	0.4997	0.5462

Table II: Disk Utilization

Users	FCFS	SCAN	SSTF	FD-SCAN	ED	SSEDV
4	0.7419	0.7316	0.7292	0.7449	0.7484	0.7395
8	0.9906	0.9878	0.9849	0.9932	0.9955	0.9955
20	0.9997	0.9996	0.9996	0.9997	0.9997	0.9997

Table III: Average I/O Queue Length

Users	FCFS	SCAN	SSTF	FD-SCAN	ED	SSEDV
4	0.735	0.706	0.699	0.728	0.738	0.701
8	3.70	3.41	3.366	3.658	3.587	3.472
20	16.22	15.79	15.62	15.63	15.11	15.02

4.2.2 Performance of Various Disk Scheduling Algorithms

In this experiment, we explore the transaction loss probability of these nine algorithms under different system workloads. The read probability is set to 0.8. For SSEDO and SSEDV, the window size is 3. The scheduling parameters α and β are set to 0.8 and 2, respectively. The results are shown in Fig.3, from which we can see the SSEDO and SSEDV algorithms significantly reduce the transaction loss ratio compared to other SCAN-based real-time algorithms (up to a 38% improvement) or conventional strategies (up to a 53% improvement). Between these two, SSEDV is better than SSEDO since the SSEDV uses more timing information than the SSEDO for decision making. All of the real-time algorithms perform better than non-real-time ones. P-SCAN and FD-SCAN perform essentially at the same level, with one better at high load cases, but worse for low load cases. The ED algorithm is good when the system is lightly loaded, but it degenerates as soon as load increases. The conventional algorithms SSTF, SCAN and C-SCAN basically perform the same, and FCFS is the worst, by far.

The mean disk service time for this experiment is plotted in Fig.4. As expected, FCFS and ED both give the highest mean disk service time which is independent of the system

load, since the request which first joined the I/O queue or with the earliest deadline is randomly located on the disk, regardless the system workload. The smallest mean disk service time belongs to the SSTF, as its name suggests. SCAN algorithm shows almost the same performance as the SSTF, and the C-SCAN is a little bit higher [25]. Disk service times of the two variants of SCAN algorithm, P-SCAN and FD-SCAN, are basically the same. The reason for why they are higher than that of SCAN is clear. A common characteristic for SSTF and various SCAN algorithms is that their disk access time drops as the system load increase, since the increase of density of waiting I/O requests results in a decrease of the seek time under these algorithms. For SSEDO and SSEDV algorithms, their disk access times decrease as the load increases, but after a point they become constants. This turning point depends on the window size, where the candidate for next service can only be selected from the window. From Table III we can see when the number of users exceeds 8, the mean queue length will be over 3. As mentioned above, the window size is set to 3 in this experiment. Consequently, further increasing system workload will have no influence on the mean disk access time.

Fig.5 shows the average response time for committed transactions. From mean disk service times shown in Fig.4, it is not hard to understand why transactions under SSEDO and SSEDV take more time to finish than that of SSTF and various SCANs but less than FCFS and ED. FD-SCAN has the same mean response time as the SSTF, SCAN, and C-SCAN. This is because although the FD-SCAN has longer disk access time, its transaction loss ratio is less than that of SSTF and SCAN. Since lost transactions miss their deadlines while in progress, the already completed steps may be blocking other transactions, and therefore increase the response time of transactions which do commit.

From Table I, we see that for a fixed workload, there is no significant difference in disk utilizations under the different disk scheduling algorithms. Fig.6 depicts the performance of all the nine algorithms as a function of the disk utilization. The advantage of real-time algorithms over non-real-time ones is obvious. For real-time algorithms, the window algorithms SSEDO and SSEDV begin to show better performance than P-SCAN and FD-SCAN when the disk utilization is over 55%. Their advantage becomes more explicit when the disk is kept busy over 90% of the time. For example, at 95% disk utilization, SSEDV outperforms FCFS by 71% and FD-SCAN by 36%.

4.2.3 Sensitivity of Changing Window Size and Scheduling Parameters in SSEDO and SSEDV

With this experiment, we are interested in how the system performs when the window size and/or scheduling parameters α and β of are changed in the two window algorithms

SSEDO and SSEDV. Note that the leftmost points in Fig.7 correspond to the window size of 1, which is equivalent to the earliest deadline (ED) algorithm. By increasing the window size, we observe an improvement of the performance, especially in the case of high load. If the system is extremely lightly loaded, e.g., the average CPU and I/O queue length is less than one, we expect all algorithms to perform the same. As the load increases, the benefit of window sizes larger than one becomes more pronounced. This is because at a higher load, the improvement caused by the *shortest seek time* component of these two window algorithms increases. On the other hand, we also observe that a window size of three or four is good enough in most cases. Increasing the window size further contributes very little to the system performance, but increases the scheduling cost. We also performed experiments for the case of tight deadline settings, and similar results were observed. They are not shown in this paper because of space limitations.

Fig.8 shows the sensitivity of scheduling parameters α and β for the SSEDV and SSEDO algorithms, respectively. Intuitively, larger (smaller) value of α (β) implies greater bias towards the *shortest seek time* component, and smaller α (larger β) implies greater bias towards the *earliest deadline* component of the SSEDV (SSEDO) algorithms. From Fig.8(a), we can see a large α is preferred in a highly loaded system, and a moderate value of α is proper for a lightly loaded system. In either case, a range of 0.7 to 0.8 for α is acceptable. This observation is consistent to the conclusions drawn from the results of Fig.7. A similar observation can be made from Fig.8(b), where $\beta = 2$ is found appropriate in either high or low workload cases.

4.2.4 Impact of Scheduling by Step Deadlines

As mentioned in Section 2.1, if the system has knowledge of the number of steps required by each transaction, it can use this knowledge for CPU and/or I/O scheduling. That is, instead of using transaction deadlines, the scheduler can make its decision by using step deadlines. Recall that the step deadlines are only used to provide priority information for scheduling and that missing a step deadline, except for the last step, does not abort a transaction. Two experiments were conducted to show the effect of scheduling by step deadlines under SSEDV, SSEDO, and ED algorithms.

In the first experiment, in which step deadlines were incremented evenly, i.e., calculated by eq.(1), we show results for four combinations of scheduling by step deadlines. From Fig.9 we observed that the strategy of scheduling both CPU and I/O scheduling by step deadline (*CSDS*) performs consistently the best under all the three disk scheduling algorithms, and the worst case is when both CPU and disk are scheduled by transaction deadlines (CTDT). The other two alternatives, i.e., CPU scheduled by transaction deadline and disk by step

deadline (CTDS), and vice versa (CSDT), fall in between. For instance, in high load cases, scheduling by step deadline can achieve up to 53% improvement over scheduling by transaction deadline under SSEDV. Also, we observed that applying step deadline on I/O scheduling results in proportionately more improvement than using it with CPU scheduling and that generally, the larger the workload, the greater the improvement.

In the second experiment, we fix CPU scheduling by transaction deadline, but perform I/O scheduling by step deadline (CTDS). The objective is to determine how to assign step deadlines. One way, as defined in eq.(1), is to evenly divide transaction's deadline among the steps so that each step has the same relative deadline (EVEN). Another way is to let early steps have loose and later steps have tight relative step deadlines (ELLT). The motivation for this is that we may expect an almost finished transaction to have a relatively higher priority. A third alternative is to allow early steps to have tight and later steps to have loose step deadlines (ETLL). In this experiment, we use the following method to assign step deadlines with ELLT and ETLL: let L, a, and n be a transaction's (absolute) deadline, arrival time, and the number of operational steps, as before. In addition, denote b as the *base step length*, and Δ as the step increment; then for ETLL, we set the deadline for step i by,

$$step\_deadline(i) = step\_deadline(i-1) + b + i\Delta \qquad i = 1, 2, ...n. \tag{4}$$

where $step\_deadline(0) = a$, or in another words,

$$step\_deadline(i) = a + ib + \frac{i(i+1)}{2}\Delta \qquad i = 1, 2, ..., n. \tag{5}$$

Since the last step's deadline is the same as transaction's deadline, we have

$$\begin{aligned} step\_deadline(n) &= a + nb + \frac{n(n+1)}{2}\Delta & (6)\\ &= L & (7) \end{aligned}$$

or

$$\frac{L-a}{n} = b + \frac{n+1}{2}\Delta \tag{8}$$

When $\Delta = 0$, substituting eq.(8) into eq.(5) reproduces eq.(1). By changing the value of b and Δ (while maintaining the constraint from eq.(8)), we can adjust the looseness of step increment. In this experiment, b is equal to $2(L-a)/3n$, and Δ can be obtained from (5). For ELLT, in a similar manner, we have

$$\begin{aligned} step\_deadline(i) &= step\_deadline(i-1) + b + (n-i+1)\Delta & (9)\\ &= a + ib + \frac{(n+n-i+1)i}{2}\Delta \qquad i = 1, 2, ...n. & (10) \end{aligned}$$

The results are illustrated in Fig.10, and we can see that ELLT performs slightly better than the other two under both SSEDV and SSEDO under variety of workloads. However, in either case, step deadlines should not be too 'loose', otherwise the policies will degenerate to the case of scheduling by transaction deadlines which was shown to be the worst in Fig.9. For the ED algorithm, there is no significant difference between various ways of assigning the step deadlines, especially when the I/O load is high.

4.2.5 Varying Transaction's Deadline Settings

This experiment examines all the nine disk scheduling algorithms under various deadline settings, from very loose to extremely tight. At one extreme, every transaction is successfully served and none are lost. At the other end of the spectrum, almost every transaction misses its deadline. From the results shown in Fig.11, we observe that SSEDV and SSEDO algorithms perform the best over the entire range of deadline settings, and the FCFS remains the worst. As also shown in the figure, when deadlines are tight (e.g., approximately 60% of transactions being lost), the SSEDV still outperforms the FD-SCAN by 6% and the SCAN by 16%. SSTF, SCAN, and C-SCAN algorithms are shown to perform almost the same. Once again, FD-SCAN outperforms P-SCAN when the deadline is tight and is outperformed by P-SCAN when the deadline becomes loose. The average transaction response time for committed transactions is depicted in Fig.12. For the case of loose deadline settings, when nearly all the transactions can make their deadline, the mean transaction response time is proportional to the mean disk service time of various disk scheduling algorithms. On the other hand, when the deadline is tight, the SSEDV and SSEDO have a lower loss ratio than others, which helps to achieve the lower mean response time for committed transactions.

4.2.6 Varying Read Probability

The purpose of this experiment is to study how the various disk scheduling algorithms perform when the read probability, p_r, is changed. In our database model, an increase in update probability (i.e., a decrease in read probability) implies more lock conflicts since update requests require exclusive locks. This also increases I/O load because more flushing work of dirty pages is required. The experimental results are shown in Fig.13. As expected, when read probability increases, the transaction loss probability decreases for all the nine I/O scheduling algorithms. However, the performance order of these algorithms remains the same as in previous discussions except for the ED algorithm which has a proportionately greater improvement as the read probability approaches one.

4.2.7 Locality of Transaction Accesses

In this subsection, we examine the locality impact of transactions' accesses. We first investigate the system behavior when the transaction sequential access probability, p_s, varies from 0 to 0.8. In this experiment, we choose a high workload case (20 Users), and a read probability of 0.8. From Fig.14, we again see that the performance of all algorithms improve when p_s increases, and that SSEDV and SSEDO algorithms are consistently better than the others. Interestingly, the increased benefit for SSTF algorithm (42% improvement) is not as large as the benefit for the SCAN and C-SCAN algorithms (66% improvement) when the *sequential access probability* increases. Whereas for the ED algorithm the gain is quite large (90% improvement) when sequential accesses occur. A partial reason for this is that the main benefit gained by increasing p_s is the reduction in seek time, which the SSTF algorithm has already taken advantage of, but the ED hasn't.

Viewing the locality effect from another perspective, we rearrange the layout of the database on the disk so that the database only occupies a portion of the disk (this situation is found in the RT-CARAT testbed), say the center 100 tracks, rather than randomly scattered on 1000 tracks. The results are shown in Fig.15. If we compare these results with Fig.3, we can see that the SSEDV, SSEDO, and ED algorithms gain more than the others by this arrangement. Among these, the ED algorithm received tremendous improvement. The reason for this is that both the SSEDV, SSEDO and ED are *time-constraint-oriented* algorithms, which means they put more weight on the *time constraint* component rather than on the *disk service time* component like the various SCANs do. For example, the SSEDV and SSEDO algorithms might choose the request with earliest deadline for next service over 80% of the time, whereas the ED algorithm would chooses that request 100% of the time. The new layout reduces the disk service time. This will help the SSEDV, SSEDO and ED algorithms to make up their deficiency with respect to the *disk service time* component, and therefore improves their relative performance.

4.2.8 Importance of Real-Time Scheduling

This final experiment is designed to see how the performance varies when the system employs real-time or non-real-time scheduling algorithms in CPU and/or disk scheduling. To examine the effect of CPU scheduling, we increase the computation time for each step to 25 msec (which is approximately the same as the mean disk service time). Four combinations, CPU scheduled by FCFS and I/O by SCAN, CPU by FCFS and I/O by SSEDO, CPU by ED and I/O by SCAN, and CPU by ED and I/O by SSEDO, are investigated. Fig.16(a) gives the transaction loss ratio in each case. Obviously, when

both CPU and I/O scheduled by real-time strategies, the performance is much better than when no real-time algorithms are used for CPU and disk scheduling (with up to a 84% performance improvement in high load cases). We also observe that the use of a real-time disk scheduling algorithm is more beneficial than the use of a real-time CPU scheduling algorithm. The transaction response times for the cases of I/O scheduled by real-time algorithms are shown to be higher than that of I/O scheduled by non-real-time algorithms in Fig.16(b). This is because the disk service times for real-time algorithms (SSEDO) are higher than non-real-time algorithms (SCAN). Therefore, we conclude that real-time disk scheduling algorithms do reduce the transaction loss ratio at the cost of higher mean transaction response time than some non-real-time algorithms do.

Interestingly, under the parameter settings above, for the two cases of I/O scheduled by non-real-time algorithms, the CPU tends to be a bottleneck as the system load increases (Fig.17). In contrast, most transactions will be queued up at the I/O device if SSEDO is employed as the I/O scheduling algorithm. The shorter seek time of SCAN algorithm partially explains why this could happen.

5 Summary

In this paper we examined the performance of various disk scheduling algorithms in an integrated real-time transaction system. Specifically, we first proposed two new real-time disk scheduling algorithms SSEDV and SSEDO, and then compared their impacts on system (transaction) level performance with other suggested real-time disk scheduling algorithms ED, P-SCAN and FD-SCAN, as well as with the conventional algorithms, SSTF, SCAN, C-SCAN, and FCFS.

The main performance results are as follows:

- In a real-time system, I/O scheduling is an important issue with respect to the system performance. In order to minimize transaction loss probability, a good disk scheduling algorithm should take into account not only the *time constraint* of a transaction, but also the *disk service time.*

- The *earliest deadline* discipline ignores the characteristics of disk service time, and therefore does not perform well except when the I/O load is low.

- The window algorithms SSEDV and SSEDO consider two factors: *earliest deadline* and *shortest seek time.* The performance of SSEDV and SSEDO algorithms is shown to be consistently better than other known disk scheduling algorithms in the sense of minimizing transaction loss probability. We also showed that SSEDV algorithm

performs better than SSEDO, since SSEDV uses more knowledge concerning the time constraint.

- For the SSEDV and the SSEDO algorithms, increasing the window size and the proper adjustment of parameters α and β can improve system performance, but increasing the window size beyond a particular value results in only marginal performance improvement.
- For a transaction system, if the number of operational steps for each transaction is known to the system as soon as a transaction is submitted to the system, we can define step deadlines according to the transaction's deadline and its step number. Scheduling by step deadlines is shown to be better than scheduling by transaction deadlines.
- When transactions' read probability or sequential access probability are high, this improves system performance. In all cases, SSEDV and SSEDO algorithms are shown to be significantly better than the other disk scheduling algorithms considered. This conclusion also holds over a wide range of transaction deadline settings. In addition, by properly arranging the layout of the database on the disk, the SSEDV, SSEDO, and ED algorithms can improve performance to a proportionally greater degree than the other algorithms.
- The average transaction response time under SSEDV and SSEDO algorithms is higher than the SSTF and all the SCAN based algorithms, but lower than FCFS and ED.

Finally, with today's technology, the disk controller can be implemented to monitor the I/O load dynamically, and select a proper scheduling algorithm accordingly [4]. This technique can be used here with our SSEDV and SSEDO algorithms in a soft real-time environment. For example, when the I/O queue length is less than a threshold, the ED algorithm (window size 1 in SSEDV or SSEDO) might be used for scheduling, otherwise the window size would be set to 3 or 4. Alternatively, we might dynamically update the scheduling parameter α or β according to a queue length threshold.

Acknowledgment:

The authors wish to thank K. Ramamritham for his constructive comments on a draft of this paper. Thanks are also due to J. Huang for his invaluable help with the experiments on the RT-CARAT testbed.

References

[1] Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[2] Abbott, R. and Garcia-Molina, H., "Scheduling Real-Time Transactions with Disk Resident Data," *A Technical Report, CS-TR-207-89, Princeton University*, Feb. 1989.

[3] Abbott, R. and Garcia-Molina, H., "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *to appear on RTSS 1990*, Dec. 1990.

[4] Bates, K. H., "Performance Aspects of the HSC Controller," *Digital Technical Journal*, No.8, Feb. 1989.

[5] Bitton, D. and Gray, J., "Disk Shadowing," *Proc. of the 14 VLDB Conf.*, 1988, pp.331-338.

[6] Buchmann, A.P. et. al., "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, Feb. 1989.

[7] Carey, M. J., Jauhari, R. and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. of the 15th VLDB Conf.*, 1989.

[8] Cheng, S., "Dynamic Scheduling Algorithms for Distributed Hard Real-Time Systems," *PhD Thesis*, Univ. of Massachusetts, 1987.

[9] Cimimiera, L., Montuschi, P. and Valenzano, A., "Some Properties of Double-Ring Networks with Real-Time Constraints," *Proc. of Real-Time Systems Symposium*, Dec. 1989.

[10] Hofri, M., "Disk Scheduling: FCFS vs. SSTF Revisited," *ACM Communications*, Vol.23, No.11, Nov. 1980.

[11] Hong, J., Tan, X. and Towsley, D., "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Trans. on Computer*, Vol.38, No.12, Dec. 1989, pp.1736-1744.

[12] Huang, J., Stankovic, J. A., Towsley, D. and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *Proc. Real-Time System Symposium*, Dec. 1989.

[13] Kohler, W. and Jenq, B.P., "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systems," *Proc. of the Fall Joint Computer Conference*, November 1986.

[14] Kopetz, H. and Ochsenreiter, W., "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. on Computers*, Aug. 1987, pp.933-940.

[15] Kurose, J. F., M.Schwartz, M. and Yemini, Y., "Multiple-Access Protocols and Time-Constrained Communication," *Computing Surveys*, March 1984, pp.43-70.

[16] Lee, I. and Davidson, S. B., "Adding Time to Synchronous Process Communications," *IEEE Trans. on Computers*, Aug. 1987, pp.941-948.

[17] Locke, C. D., Tokuda, H. and Jensen, E. D., "A Time-Driven Scheduling model for Real-Time Operating System," *Technical Report, CMU*, 1985.

[18] Malcolm, N., Zhao, W. and Barter, C., "Guarantee Protocols for Communication in Distributed Hard Real-Time Systems," *Proc. of IEEE INFORCOM'90*, June 1990, pp.1078-1086.

[19] Ramamritham, K. and Stankovic, J. A., "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, 1(3), 1984.

[20] Ramamritham, K., "Channel Characteristics in Local Area Hard Real-Time Systems," *Computer Networks and ISDN Systems*, 1987, pp.3-13.

[21] Seltzer, M., Chen, P. and Ousterhout, J., "Disk Scheduling Revisited," *Proc. of USENIX, Winter'90* pp. 313-323.

[22] Sevcik, K. C. and Johnson, M. J., "Cycle Time Properties of the FDDI Token Ring Protocol," *IEEE Trans. on Software Engineering*, March 1987, pp.376-385.

[23] Son, S. H. and Chang, C. H., "Priority-Based Scheduling in Real-Time Database Systems," *Proc. of the 15th VLDB Conference*, 1989.

[24] Tokuda, H., Mercer, C. W., Ishikawa, Y. and Marchok, T. E., "Priority Inversions in Real-Time Communication," *Proc. of Real-Time Systems Symposium*, Dec. 1989.

[25] Teorey, T. J. and Pinkerton, T. B., "A Comparative Analysis of Disk Scheduling Policies," *ACM Communications*, Vol.5, No.3, March 1972, pp.177-184.

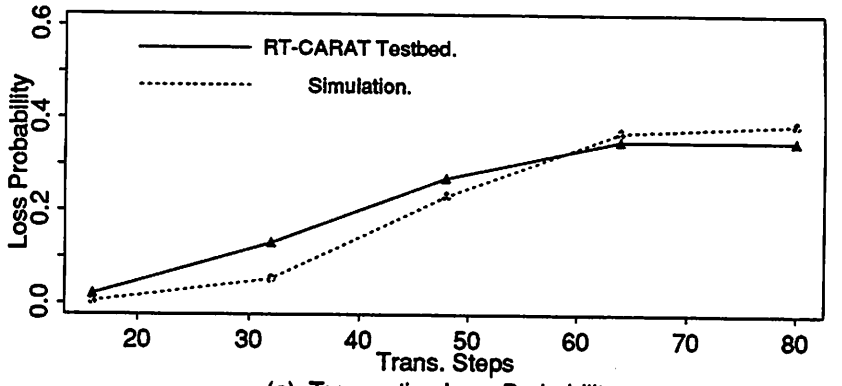
[26] Towsley, D. and Panwar, S. S., "On the Optimality of Minimum Laxity and Earliest Deadline Scheduling for Real-Time Multiprocessors," *Proc. Euromicro'90 Workshop on Real-Time*, June 1990, pp.17-24.

[27] Warchol, N. A. and Shirron, S. F., "The RQDX3 Design Project," *Digital Technical Journal*, No.2, March 1986.

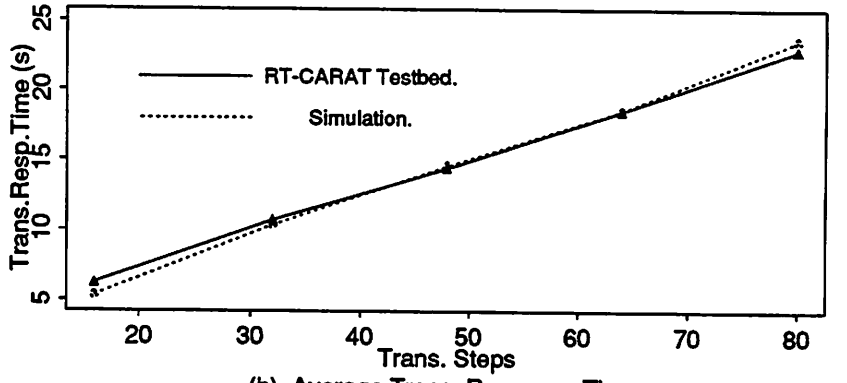
[28] Zhao, W., Ramamritham, K. and Stankovic, J. A., "Preemptive Scheduling under Time and Resource Constraints," *IEEE Trans. on Computers*, Aug. 1987, pp.949-960.

[29] Zhao, W., Ramamritham, K. and Stankovic, J. A., "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, SE-12(5), 1987.

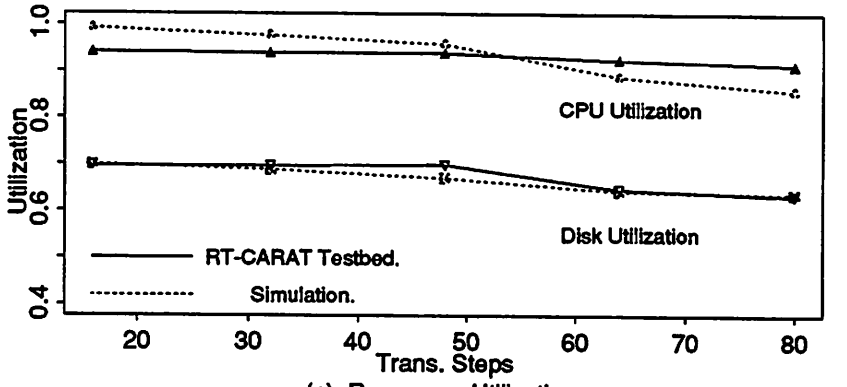
[30] Zhao, W. and Ramamritham, K., "Virtual Time CSMA Protocols for Hard Real-Time Communications", *IEEE Trans. on Software Engineering*, Aug. 1987, pp.938-952.



(a). Transaction Loss Probability.



(b). Average Trans. Response Time.



(c). Resources Utilization.

Figure 2: Results Compared with RT-CARAT Testbed. ($Users = 8, p_r = 0.8, p_s = 0$)

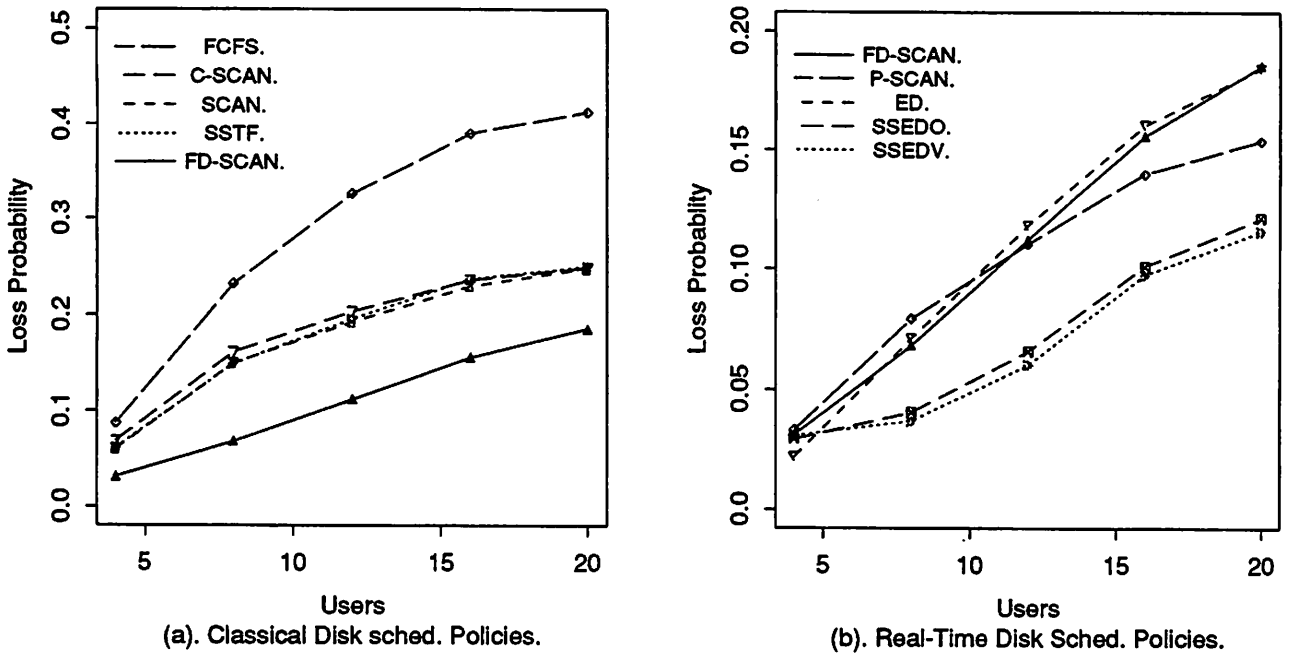


Figure 3: Performance of Various Disk Scheduling Algorithms. ($p_r = 0.8, p_s = 0$)

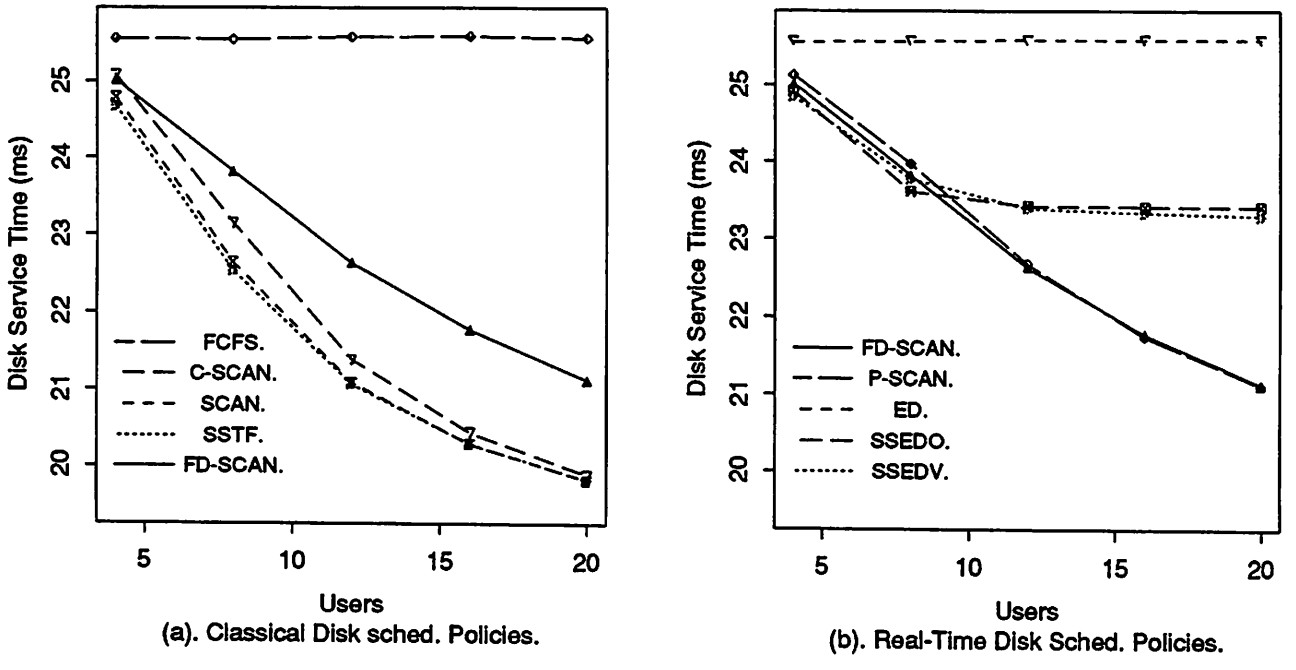


Figure 4: Mean Disk Service Time for Different Algorithms. ($p_r = 0.8, p_s = 0$)

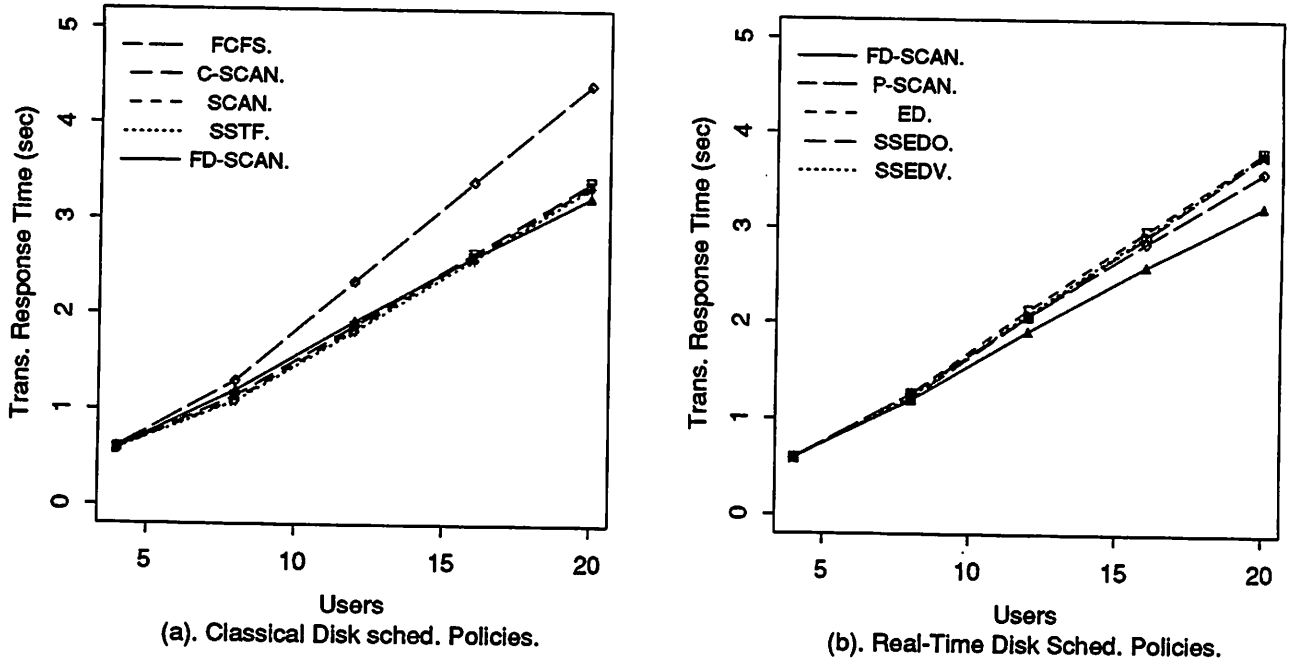


Figure 5: Mean Transaction Response Time for Different Algorithms. ($p_r = 0.8, p_s = 0$)

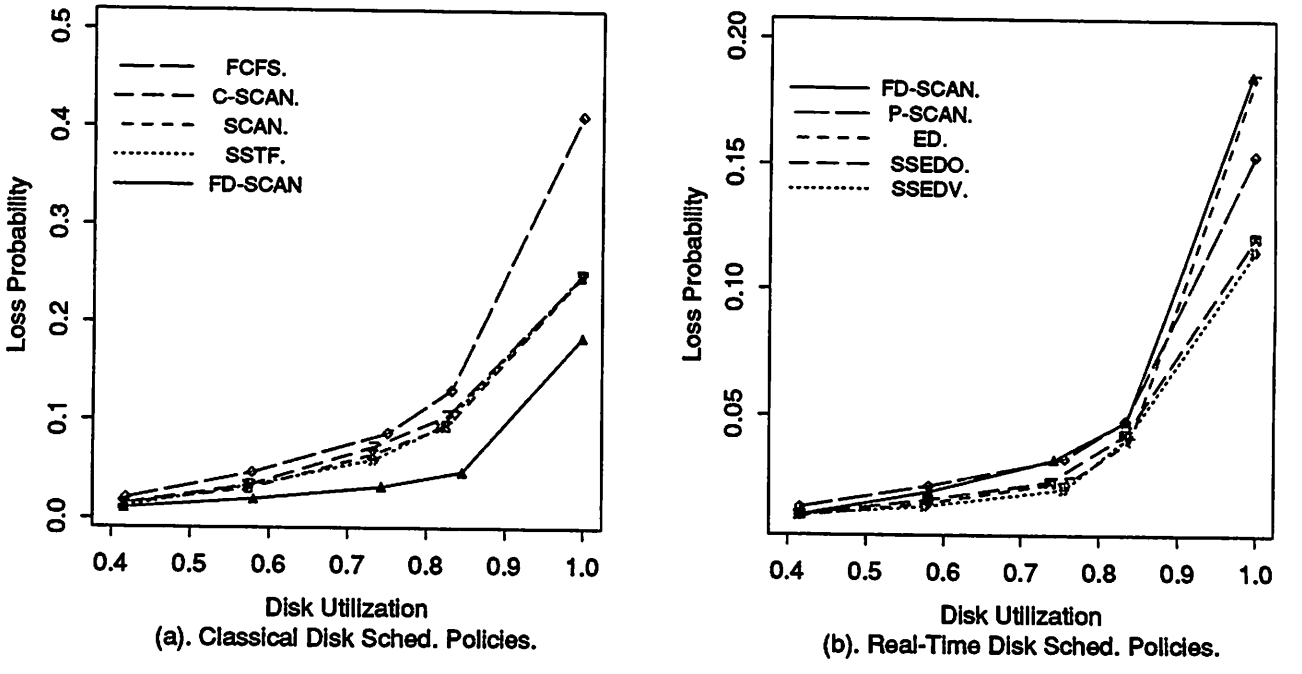


Figure 6: Performance with Fixed Disk Utilizations. ($p_r = 0.8, p_s = 0$)

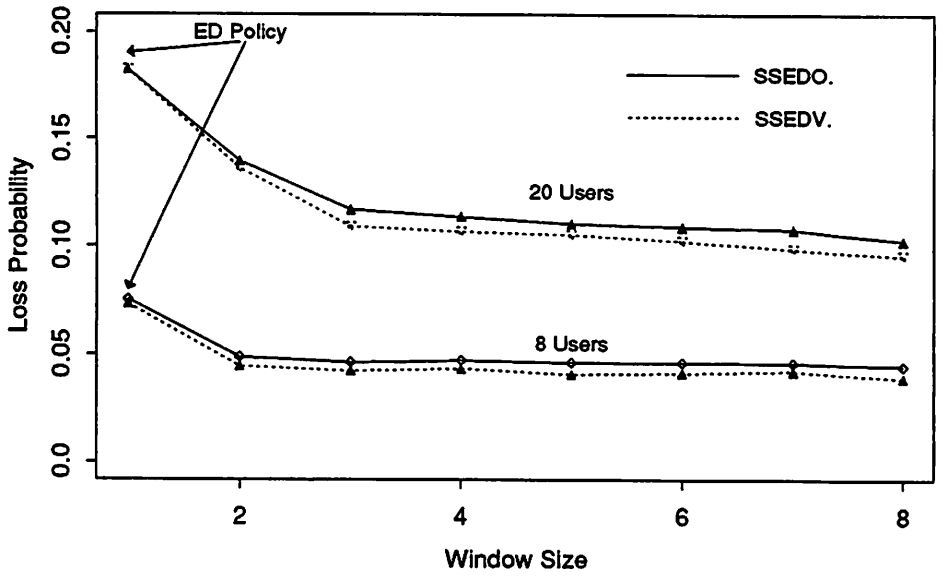


Figure 7: Performance of SSEDV and SSEDO with Different Window Size. ($p_r = 0.8, p_s = 0$)

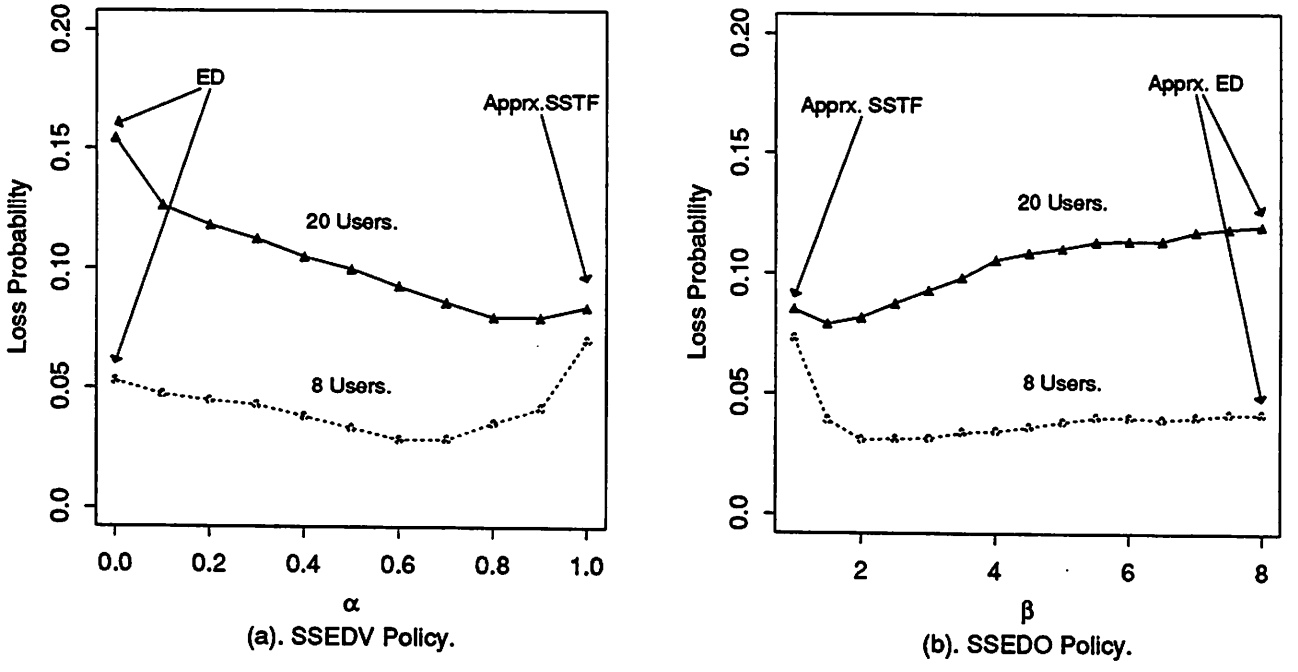
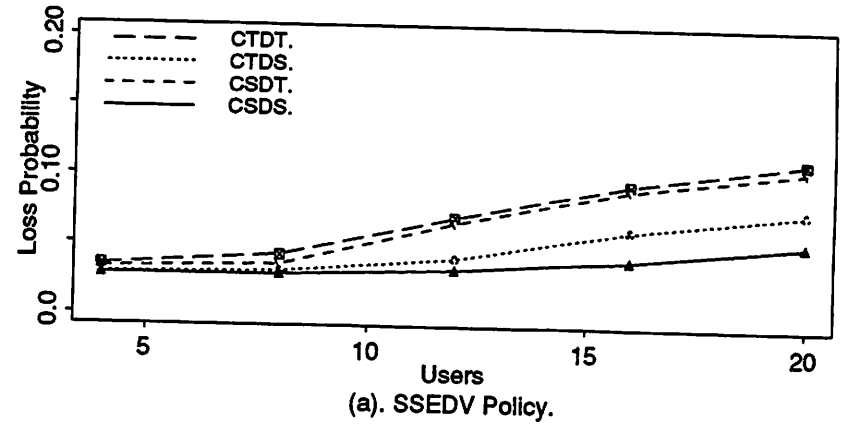
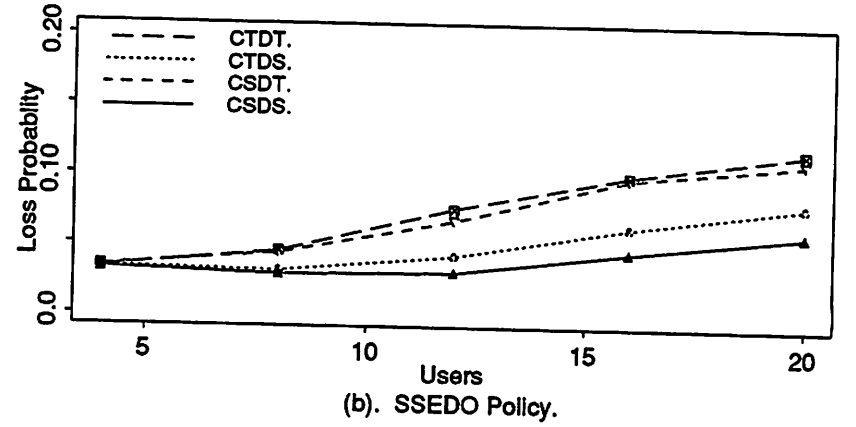


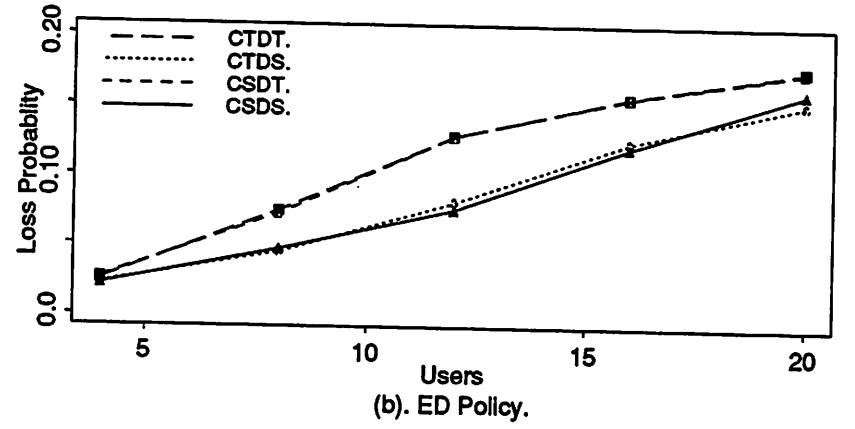
Figure 8: Sensitivity of Scheduling Parameters of SSEDV and SSEDO. ($p_r = 0.8, p_s = 0$)



(a). SSEDV Policy.

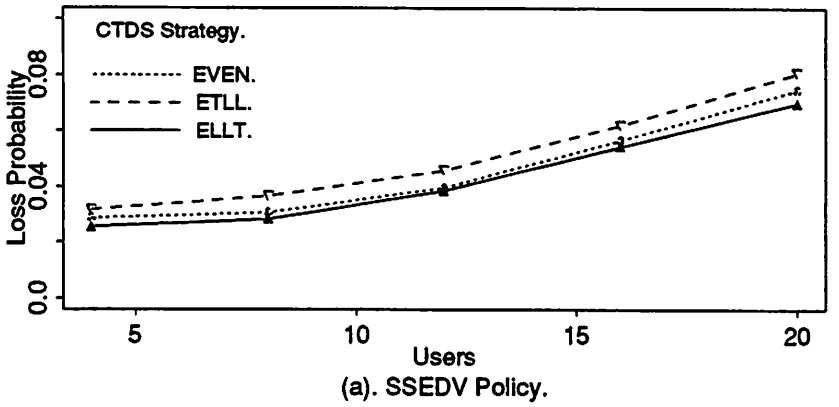


(b). SSEDO Policy.

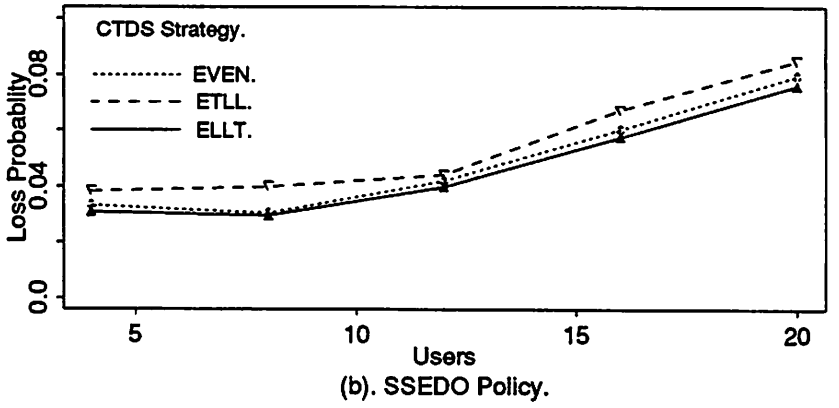


(b). ED Policy.

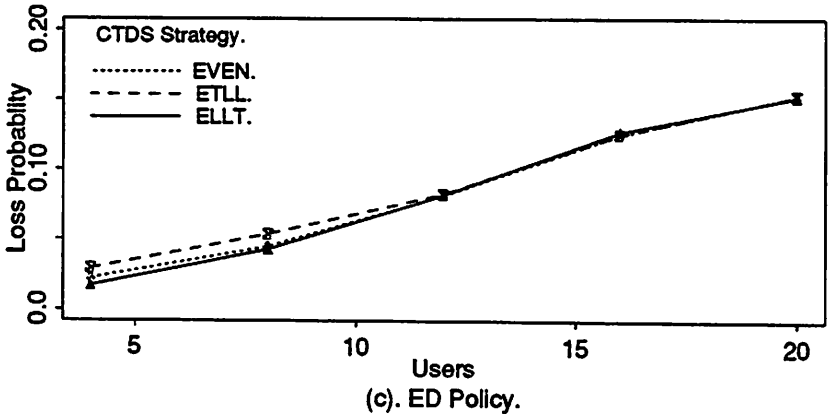
Figure 9: Scheduling by Step Deadlines under SSEDV and SSEDO. ($Users = 20, p_r = 0.8, p_s = 0$)



(a). SSEDV Policy.



(b). SSEDO Policy.



(c). ED Policy.

Figure 10: Step Deadline Assignment with SSEDV and SSEDO. ($Users = 20, p_r = 0.8, p_s = 0$)

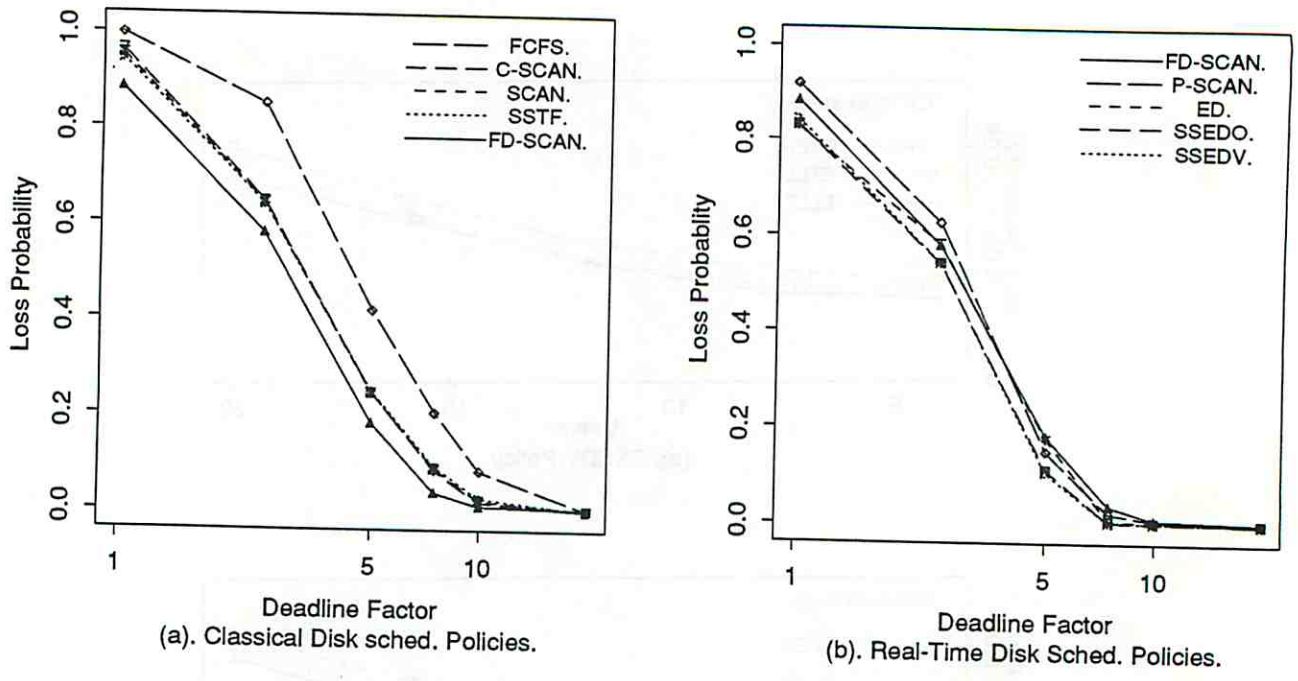


Figure 11: Performance with Various Deadline Settings. ($Users = 20, p_r = 0.8, p_s = 0$)

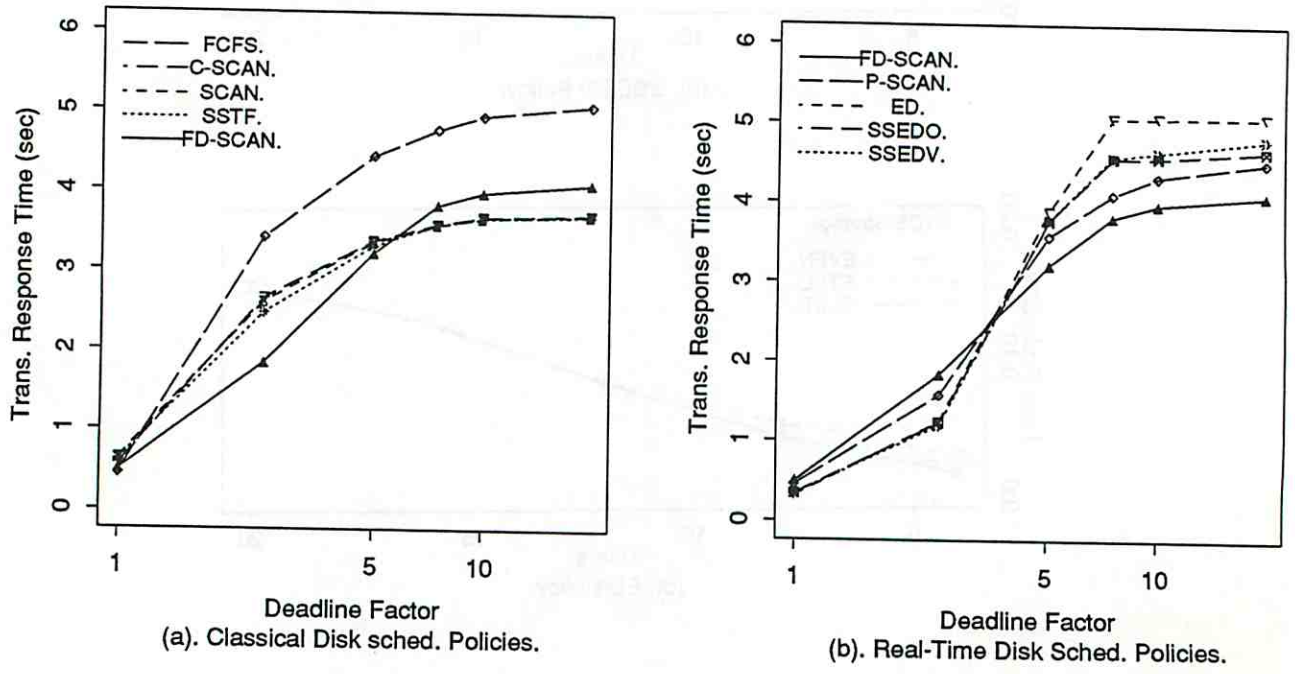


Figure 12: Mean Transaction Response Time with Various Deadline Settings. ($Users = 20, p_r = 0.8, p_s = 0$)

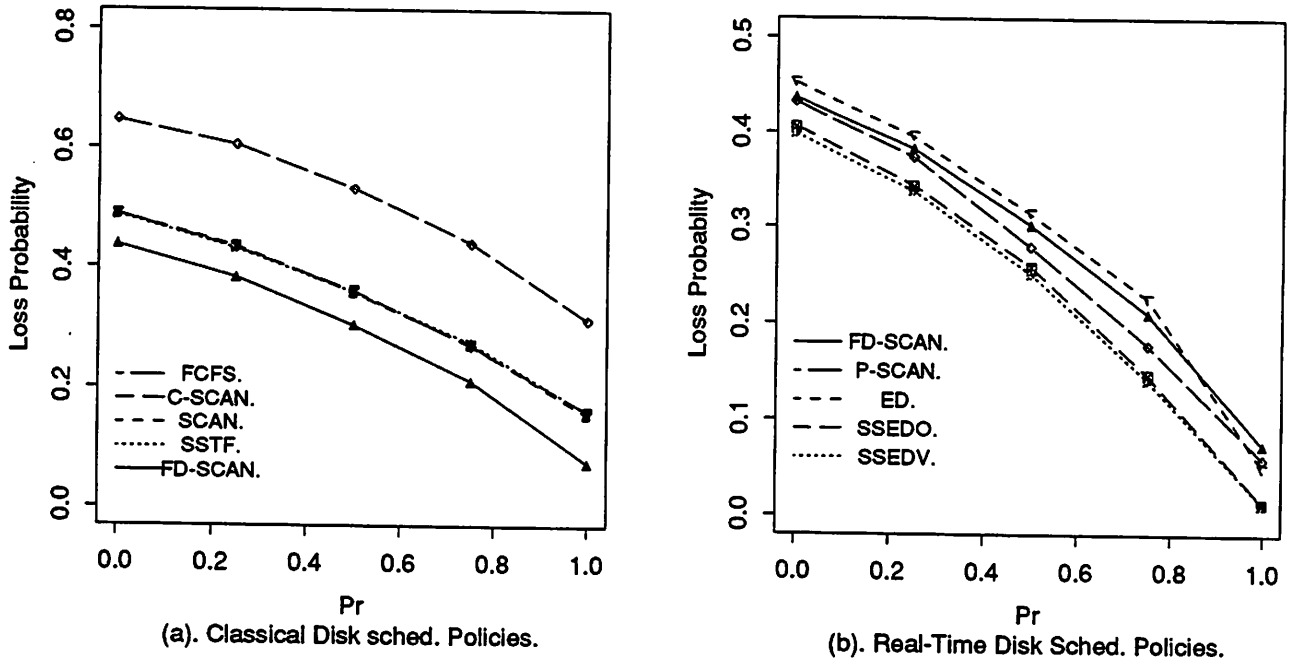


Figure 13: Performance under Different Read Probability. $(Users = 20, p_s = 0)$

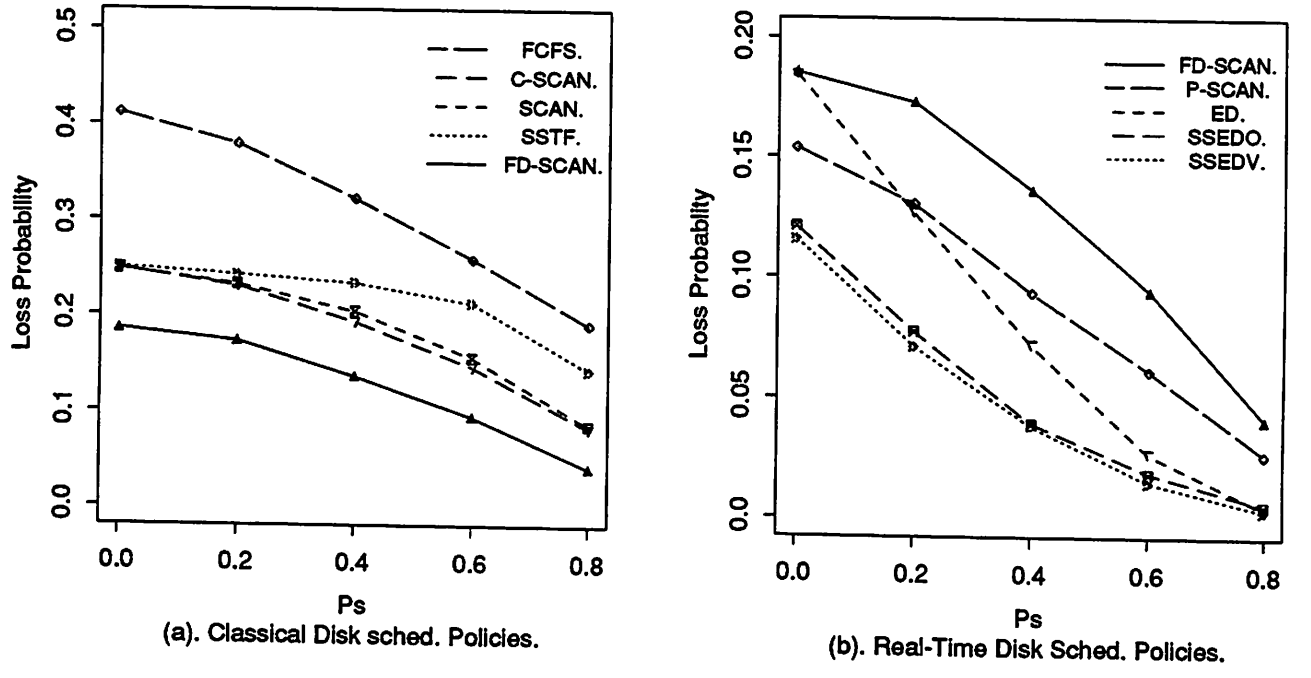


Figure 14: Performance under Different Sequential Access Probability. $(Users = 20, p_r = 0.8)$

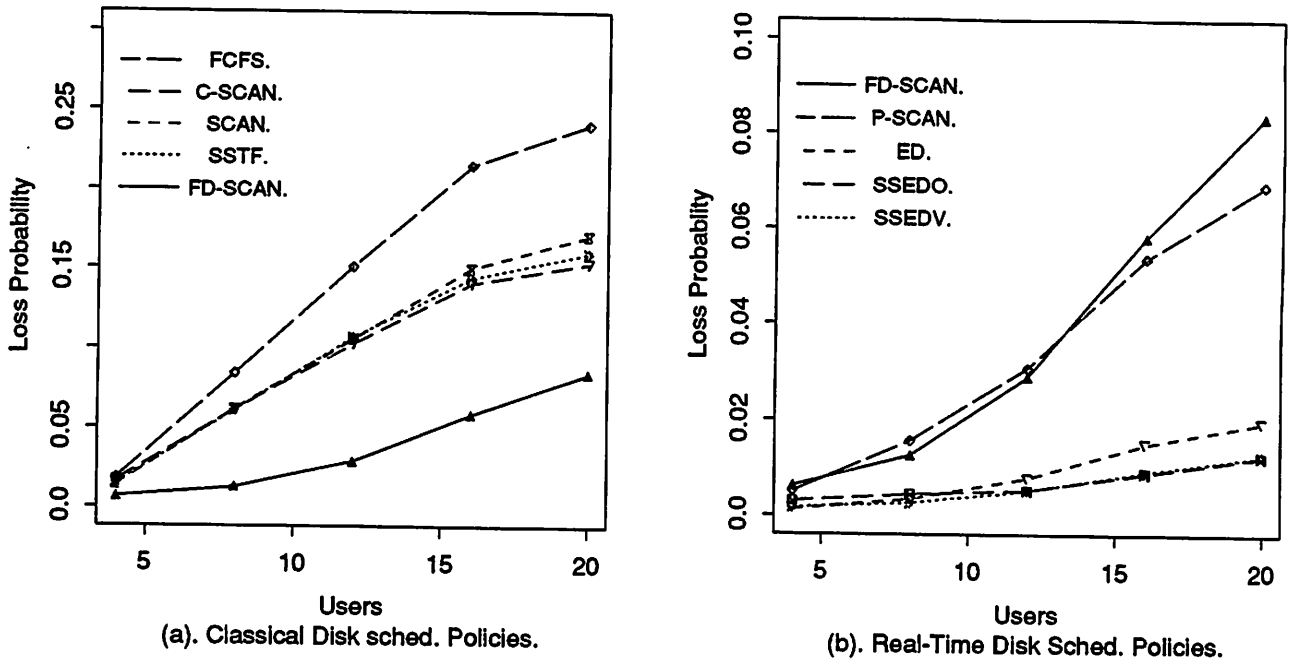


Figure 15: Performance under Different Database Layout. $(p_r = 0.8, p_s = 0)$

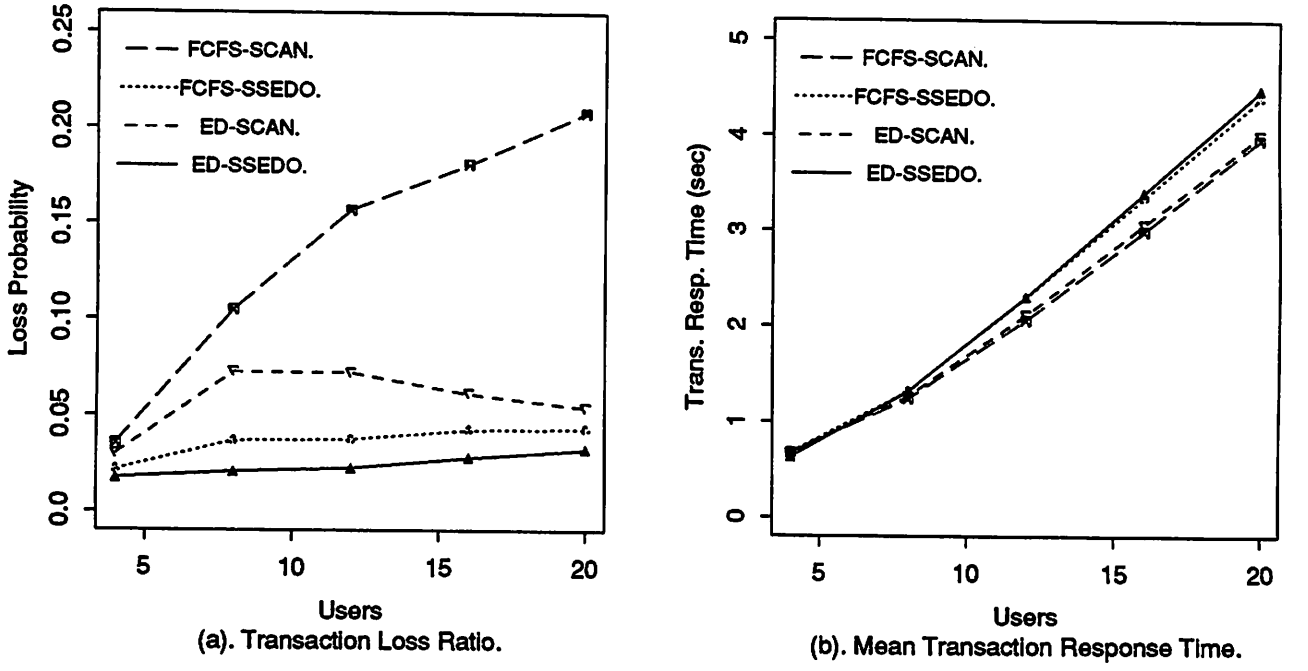


Figure 16: Importance of Real-Time Scheduling. $(p_r = 0.8, p_s = 0, Comp\_Time = 25ms)$

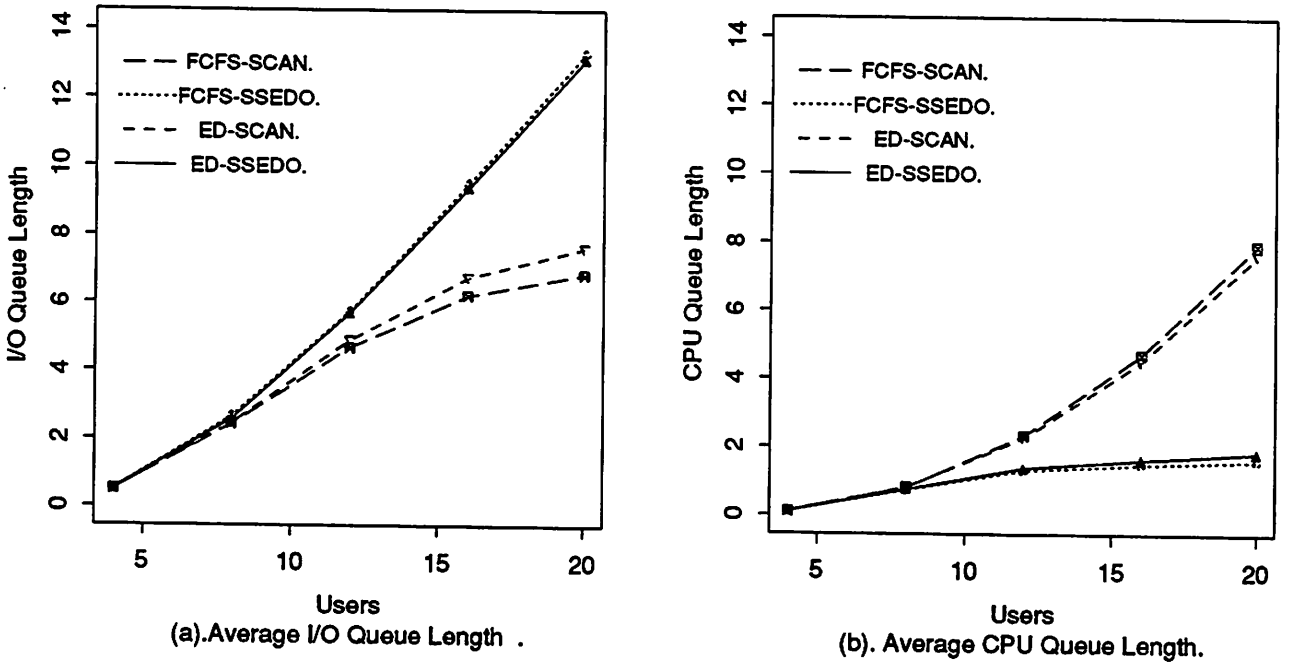


Figure 17: Average CPU and I/O Queue Length. ($p_r = 0.8, p_s = 0, Comp\_Time = 25ms$)