Scheduling I/O Requests with Deadlines: a Performance Evaluation

Robert K. Abbott

Digital Equipment Corp.
151 Taylor St. (TAY1)
Littleton, MA 01460

Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

A real-time computing system allocates resources to tasks with the goal of meeting individual task deadlines. The CPU, main memory, I/O devices and access to shared data all should be managed with the goal of meeting task deadlines. This paper examines the I/O scheduling problem in detail. We present a detailed, realistic model for studying this problem in the context of a system which executes real-time transactions. The model takes advantage of the fact that reading from the disk occurs before a transaction commits while writing to the disk usually occurs after the transaction commits. We develop new algorithms that exploit this fact in order to meet the deadlines of individual requests. The algorithms are evaluated via detailed simulation and their performance is compared with traditional disk scheduling algorithms.

1 Introduction

Traditional real-time systems typically are designed to meet all (hard) deadlines and to hold the data they need in main memory. Our goal is to expand this view to include other types of systems where there are real-time constraints. In particular, we are interested in systems where data must be *consistent* and *persistent*. This means that the data must be stored on non-volatile storage, usually disks. It also means that the applications modify the data via *transactions*, i.e., collections of actions that must be executed as an atomic operation. In such systems it is very difficult to meet all deadlines. One reason is that the disk introduces unpredictable seek and rotational delays. A more important reason is that in many applications it is impossible to predict the data and computational requirements of transactions, i.e., as a transaction executes and reads data, it decides what to do next. Thus, our view of such systems is that deadlines are *soft*. This means the transaction management system must make every effort to meet all deadlines, but it is understood that in overload situations or when transactions have tight deadlines, some will be missed.

We believe that there are many applications that can utilize such persistent, real-time data management facilities. For example, a radar system may need to compare images of objects against a database of known aircraft types. In a program trading application, transactions must complete trades and record them in a database by a given time. Even in conventional applications such as banking, it will be possible to use deadlines to give transactions different priorities.

Designing a computing system that supports the execution of transactions with deadlines presents many new problems. Chief among these problems is the management and scheduling of system resources under real-time performance metrics. For example, the CPU, main memory, I/O devices and access to shared data all should be managed with the goal of meeting individual transaction deadlines. In previous work we have examined in detail CPU scheduling and concurrency control for real-time transactions[1,2]. In [2] a simple model of an I/O system was used to study the effect of using real-time priorities to schedule disk accesses. Our results indicated that priority based disk scheduling could be beneficial to overall performance. However the interaction between reads and writes needed to be carefully controlled. This paper examines in more detail the problem of scheduling I/O requests with deadlines.

Consider a common method for modeling a transaction in a centralized database system: a transaction is an alternating sequence of data actions (reads and updates) and compute actions. The sequence terminates with a COMMIT action where log processing is done and locks are released. Finally, the modified pages produced by the transaction are written to the disk resident database. If we make the assumption that memory is large enough to hold all uncommitted updates, then the following observations are true:

- Requests to read pages from the disk resident database are always performed *before* the requesting transaction is committed.
- Requests to write modified pages back to the disk are always performed *after* the transaction which created the modified pages has committed.

These observations are particularly important in the context of a system which executes real-time transac-

CH2933-0/90/0000/0113$01.00 © 1990 IEEE

tions. It means that the I/O system will service two types of requests: reads and writes. Read requests are issued by uncommitted real-time transactions. These requests should receive service in accordance with the time constraints of the tasks that issued them. In general, this means that read requests issued by tasks with immediate deadlines are serviced before read requests issued by tasks with later deadlines. How the time constraints of the issuing tasks can be inherited by the read requests themselves is but one question. How should we evaluate system performance in meeting these time constraints is another.

Read requests have explicit time constraints which they inherit from the tasks that issued them. Write requests do not have explicit time constraints because they are not issued by real-time tasks. For example, a buffer manager writes modified pages to disk in order to maintain a suitable amount of free memory. Typically, the performance of a buffer manager is not measured with real-time metrics. A write request must be serviced but *when* it should be serviced is not clear. On the one hand, write requests should be serviced at an average rate that is equal to their average arrival rate. On the other hand, servicing write requests can interfere with the timely servicing of read requests. This interference should be minimized. How can we service write requests and still meet the deadlines of read requests? How can we ensure that write requests are serviced "often enough" even though they lack real-time constraints?

It is conceivable that some write requests in a real-time transaction system will have explicit hard deadlines. For example, a distributed real-time system may implement a timed atomic commit protocol [12]. In this multi-phase protocol, each phase must complete before the next can begin. The completion of a phase is recorded by writing a record to stable storage, i.e., the disk. However we believe that the above observations are true for most I/O requests. Therefore, throughout this paper we will refer to I/O requests with deadlines as Reads, and requests without deadlines as Writes. However our model and scheduling algorithms can work for write requests with deadlines as well.

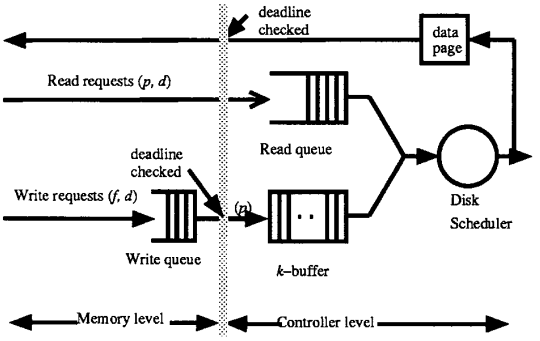
Finally, there is the problem of scheduling the disk head itself. Traditional algorithms perform seek optimization to meet non real-time performance goals. Will these same algorithms perform well under real-time metrics? What kinds of algorithms can be developed using deadline information? How well do they perform? Should read requests be handled differently from write requests? The questions and issues we have raised here are addressed in the rest of the paper.

2 Model and Assumptions

The basic model that we use to study the problem of scheduling I/O requests with deadlines is shown in Figure 2-1. Our model has two halves: memory, which is depicted on the left and the I/O subsystem which is shown on the right. The boundary between the two serves as a deadline checkpoint, a place where we evaluate if time constraints have been met.

Read and write requests are generated not by transactions directly but by a *buffer manager* that lies between the running transactions and the disk handler. This buffer manager receives read and write requests from the transactions, and in turn generates read and write requests for the disk handler.

Figure 2-1 Model for I/O Requests with Deadlines



If a transaction read request cannot be satisfied by a page in the buffer, then the request is forwarded to the disk handler. We model each read request by the pair (p, d), where p is the required page, and d is the deadline. The deadline d is determined by the buffer manager based on the deadline of the transaction that issued the read. Each request also identifies the page in the buffer that is to hold the newly read page. The problem of determining the deadline d from the transaction deadline is not considered in this paper.

Thus read requests are issued by the buffer manager, serviced by the disk handler, and processed by the disk. The data becomes available to the requesting transaction only when the page has been read into the memory buffer. So the deadline for a read request sets a time limit for the round trip from issuance, through the controller level read queue and back to main memory. The path of read requests is shown in Figure 2-1.

Note that the deadline is checked at the border between memory and the I/O system (disk handler) on the way back from the disk.

When the buffer manager needs to clear buffer space, or needs to force writes to disk, it flushes dirty buffer pages. This generates write requests for the disk handler. We do not model the particular page replacement or force policy used by the buffer manager itself. Rather, we assume that write requests are generated at an average rate λ_w. A write request (*f, d*) means the buffer manager needs to free frame *f* by time *d*. The contents of frame *f* is a modified data page *p*.

There are a several ways to model how a modified page is written out to disk. For example, page *p* can remain in memory while a write request is issued to the I/O system. The write request would eventually be scheduled at the disk and the data page is pulled from main memory and written on the disk. This model is an analog of the read request model. The buffer frame is not actually emptied until the write is performed at the disk. The deadline is a time constraint on the movement of data from the memory to this disk.

A serious flaw with this method is that a deadline is associated with an individual write request from the time it is issued until it is serviced at the disk. Unlike read requests, there is no natural way to assign these individual deadlines since write requests are not issued by real-time transactions. The buffer manager is concerned only with freeing buffer frames; it does not care about when specific pages are written to disk.

A second way to model how a buffer page makes its way to disk is shown in Figure 2-1. Under this model the modified page *p* is first copied into a frame in a fixed size buffer pool. We call this buffer the *k-buffer* where *k* is the number of available frames. This buffer is managed by the disk handler and is separate from the buffer used by the buffer manager. The *k*-buffer can be thought of as a buffer at the disk controller, or as a buffer at a lower level of the operating system than the buffer directly seen by transactions. (The latter is a common arrangement, where the database management system manages its own buffer for transactions, and the operating system underneath manages the *k*-buffer.)

Once a page is copied into the *k*-buffer, the frame *f* is free. Thus, the deadline is satisfied if this copy is done before time *d*. The copy can be done immediately so long as there is an empty place in the *k*-buffer. If there is not, then the page must wait in the frame *f* until a place becomes available. Note that in this case the deadline applies only to the buffer to buffer copy operation, and not to the eventual disk write. Section 3 discusses how we create time constraints for emptying the *k*-buffer so that copy operation deadlines are met.

The model we have presented is relatively simple; yet we think it captures the essential aspects of the disk handler and buffer manager interactions. A more complete model could be postulated, one that included the details of the buffer manager, CPU scheduling, and the transactions themselves. Such a model would let us study the performance of the overall system. However, since our model uncouples the disk handler from the rest of the system and reduces the number of irrelevant parameters, we believe it is better suited to studying the finer points of disk scheduling.

One last observation we make concerns the the use of deadlines at the level of individual I/O requests. Clearly, the top level transactions have deadlines. But how are these deadlines translated into deadlines for the individual read requests made by the buffer manager? Would it be better to use fixed priorities for the read requests, as suggested by [4]?

Regarding the first question, there are many ways to translate deadlines. The simplest is to give the I/O request the same deadline as the issuing transaction. A more sophisticated approach would account for future requests to be issued by the transaction as well. For example, if it is known that a transaction typically issues 10 read requests, the first one would get a tenth of the remaining time to the final deadline, and so on. But if no access pattern information is available, the simple approach is probably the best. That is, in this case the I/O requests should be prioritized by the deadline of the issuing transaction. Although the issue of assigning deadlines is important, we do not discuss it further in this paper.

Regarding the second question, a fixed priority scheme is an alternative to deadlines. Fixed priorities are natural in some applications, but in applications where deadlines exist, we believe it is more natural to work with deadlines and to carry them through to lower level tasks like I/O requests. Furthermore, deadlines give us more flexibility: 1) we are not constrained to a fixed set of priorities, 2) we can tell when a request is unfeasible and can be ignored, and 3) they may be more useful in a distributed system where the priorities of one computer may be different from those of others.

3 Managing the *k*-Buffer

Our model contains two types of I/O requests: those with deadlines and those without. The normal mode of operation is to give a higher priority to servicing the requests with deadlines. The requests without deadlines are serviced either when there are no read requests or when the buffer is in danger of overflowing.

This section discusses two heuristic techniques that can be used to trigger service for write requests. These policies provide a way to decide if the next I/O request to service will be a read or a write. They do not choose which particular request to service. This decision is left to the scheduling policies in Section 4.

3.1 Space Threshold

The motivation of the Space Threshold policy is to maintain a minimum amount of free space in the write buffer at all times, so that each arriving write request will find an open slot in the buffer. The amount of free space that is maintained is a parameter that can be tuned to accommodate the expected "burstiness" of the write arrival pattern. A space threshold heuristic is applied as follows:

IF Free Space < Threshold **OR**
there are no read requests
THEN Service write requests.
ELSE Service read requests

Preferential service is granted to read requests so long as the space threshold has not been exceeded. Otherwise, write requests are serviced. Write requests are also serviced when there are no read requests in the system.

3.2 Time Threshold

The motivation of the Time Threshold technique is to create an artificial deadline for a write event. We use the term "write event" to denote the action of writing the contents of a buffer slot to disk. Note that a write event, and hence the write event deadline, is not associated with a particular write request. The closeness of the write event deadline reflects the urgency of emptying a buffer slot. If the buffer is relatively empty then the deadline would be far off. If the buffer is nearly full then the deadline would be much sooner. Let D_W be the deadline for the write event. The Time Threshold approach is applied as follows:

IF D_W < The earliest read deadline **OR**
there are no read requests
THEN Service a write request
ELSE Service read requests

Again, the method for choosing which read or write request to service is decided by one of the algorithms in Section 4.

In contrast to the Space Threshold approach, Time Threshold always gives consideration to the timing requirements of read requests, regardless of how full the write buffer is. However, when the buffer is almost full, it is unlikely that any read request will have an earlier deadline than D_W. The problem now is to assign appropriate deadlines for write events. We chose to adopt a simple linear function which uses the state of the write buffer (amount of free space) and an estimate of write arrival patterns in order to create write event deadlines.

3.2.1 Linear Function for Setting D_W

Using this approach, the closeness of the deadline varies linearly with the amount of free space in the write buffer. The average inter-arrival time of write requests is the slope. Thus $D_W = t + (1 / \lambda_W) * (\text{free space} + 1)$. λ_W is the average arrival rate of write requests and t is the current time. Thus if a write request arrives and fills the last slot in the buffer, i.e., free space = 0, then D_W is set to $(1 / \lambda_W)$ seconds from the current time. If a buffer can be emptied before the next write arrival is expected then the buffer will not overflow.

Note that other, more complex functions to set D_W are possible. However, these functions seem unnecessary given that the Space Threshold and a simple linear function for the Time Threshold worked well in practice (See Section 6, 7).

4 Disk Scheduling

This section presents a number of algorithms for scheduling I/O requests. The first three are traditional disk scheduling algorithms that have been studied before, although not in the context of real-time performance metrics. This group of scheduling policies does not use deadline information to make scheduling decisions. Seek optimization, if done at all, is done with the use of track information. The algorithms that we have chosen are generally acknowledged to perform well under one or more of the traditional non real-time performance metrics: average response time, throughput, fairness to requests. Thus their performance can be used as a reference against which the real-time scheduling algorithms can be compared. The second group of three are new algorithms that are specifically designed for the real-time environment. These algorithms do make use of deadline information.

In the following subsections we explain each scheduling policy and illustrate how it can be combined with the policies for managing the k-buffer (Section 3) in order to create a complete algorithm. The set of requests which the algorithm is trying to schedule is composed of the read requests, each a pair (p, d), and the writes, each simply (p). We use p to denote the track number where the page is located on disk, and d is the deadline for read requests. If the Space Threshold technique is used for managing the k-buffer then F denotes the free space threshold. If the Time Threshold technique is employed, then D_w denotes the write event deadline.

4.1 Three Traditional Algorithms

We chose to study three traditional disk scheduling algorithms under real-time performance metrics. They are First-Come-First-Served (FCFS), Shortest-Seek-Time-First (SSTF), and the elevator algorithm SCAN The FCFS algorithm orders requests by arrival time. The SSTF algorithm chooses the requests that is closest to the current head position. In the SCAN algorithm, the head sweeps back and forth across the disk surface servicing all requests that lie ahead of it. When the head reaches an edge of the disk surface, the direction is reversed. The performance of these algorithms under non real-time metrics has been studied extensively [5,6,7,10,11].

The traditional algorithms schedule from a single pool of requests. There is no distinction between read and write requests. Also these scheduling models do not use a buffer pool to buffer write requests. Applied as is to our model, the algorithms do not use a special policy for managing the k-buffer.

Although these algorithms do not require the use of a buffer management technique they can be modified to make better use of the k-buffer by combining them with the Space Threshold technique. For example, combining FCFS with Space Threshold produces the following algorithm:

IF Buffer free space $< F$ **OR**
there are no read requests
THEN Service the closest write request.
ELSE Service the read request with
the earliest arrival time.

The FCFS scheduling policy is applied only to read requests. The SSTF policy is used to service writes so that the k-buffer can be emptied as quickly as possible. Notice that write requests are only serviced when the free space threshold has been exceeded or when there are no read requests to service. The corresponding algorithms for SSTF and SCAN are formed similarly.

4.2 Three Real-Time Scheduling Algorithms

4.2.1 Earliest Deadline First (ED)

The Earliest Deadline algorithm is an analog of FCFS. Read requests are ordered according to deadline and the request with the earliest deadline is serviced first. Since no positional information is used to make scheduling decisions, ED will have the same expected seek time profile as FCFS. Unlike the first three algorithms, it is necessary to use ED with one of the k-buffer management techniques. This is because ED uses only deadline information to make scheduling decisions. Since write requests do not have deadlines, one of the k-buffer management techniques must be employed to guarantee that the k-buffer is emptied.

Combining ED with the Space Threshold technique for managing the k-buffer produces the following algorithm:

IF Buffer free space $< F$ **OR**
there are no read requests
THEN Service the closest write request
ELSE Service the read request with
the earliest deadline.

Combining ED with a Time Threshold technique for managing the k-buffer produces the following algorithm: (Recall that D_W denotes the write event deadline.)

IF $D_W <$ the earliest read deadline **OR**
there are no read requests
THEN Service the closest write request
ELSE Service the read request with
the earliest deadline

4.2.2 Earliest Deadline SCAN (D-SCAN)

This algorithm is a modification of the traditional SCAN algorithm. In D-SCAN the track location of the read request with the earliest deadline is used to determine the scan direction. The head seeks in the direction of the read request with the earliest deadline servicing all read requests along the way (these will be for requests with later deadlines) until it reaches the target track. After the target request is serviced, the scan direction is updated towards the direction of the read request with the next earliest deadline. We have chosen to use earliest deadline to select the scan direction, however any real-time priority scheme could be used, e.g., least slack.

Unlike the traditional SCAN algorithm the new scan direction may be the same as the previous scan direction. Also, the scan direction can change before the target track is reached. This will happen if a request with an earlier deadline than the target request, and a track location behind the current head position arrives after the scan direction is chosen. Also D-SCAN does not scan to the last request in a particular direction. If the requests with the earliest deadlines are grouped within a small region, then D-SCAN will scan only in that region. However as time progresses, the deadlines of requests at other portions of the disk will be the earliest and the head will scan over those portions of the disk.

Like ED, D-SCAN must be combined with one of the k-buffer management techniques to ensure that write requests are serviced. Using the Time Threshold technique produces the following algorithm:

IF D_W < the earliest read deadline **OR**
there are no read requests
THEN Service the closest write request
ELSE Service the closest read request
in the scan direction.

4.2.3 Feasible Deadline Scan (FD-SCAN)

The FD-SCAN algorithm is similar to D-SCAN except that only read requests with feasible deadlines are chosen as targets that determine the scanning direction. A deadline is feasible if we estimate that it can be met. More specifically, a request that is n tracks away from the current head position has a feasible deadline d if $d \geq t + Access(n)$ where t is the current time and *Access(n)* is a function that yields the expected time needed to service a request n tracks away.

Each time that a scheduling decision is made, the read requests are examined to determine which have feasible deadlines given the current head position. The request with the earliest feasible deadline is the target and determines the scanning direction. The head scans toward the target servicing read requests along the way. These requests either have deadlines later than the target request or have unfeasible deadlines, ones that cannot be met. If there is no read request with a feasible deadline, then FD-SCAN simply services the closest read request. Since all request deadlines have been (or will be) missed, the order of service is no longer important for meeting deadlines and SSTF is used to efficiently service the outstanding requests as quickly as possible.

Like ED, and D-SCAN, FD-SCAN must be combined with a buffer management policy to ensure that write requests are serviced. Combining FD-SCAN with the Time Threshold technique produces an algorithm similar to that produced by D-SCAN.

5 Simulation Model

To test the algorithms, we built a program to model real-time I/O requests in a system with a single data disk. We make the realistic assumption that log writes are directed toward a separate device. Our program was built using CSIM, a process-oriented simulation language [8].

5.1 Device Model.

The program models a single-head disk device at the track level; we do not model sectors within tracks. This is reasonable since none of the algorithms under study performs rotational optimization. The names and meaning of the four parameters that control the I/O system configuration are shown in Table 5-1.

Table 5-1 Device Parameters

Parameter	Meaning	Base Value
Tracks	# of tracks on disk	1000
BufferSize	# of pages in k-buffer	10
SeekFactor	Seek time scaling factor	0.6 ms
DiskConstant	Rotational latency + transfer	15.0 ms

The access time for an I/O request n tracks away from the current head position is expressed by the equation:

$$Access(n)=Seek(n)+Rotational\ latency+Transfer\ time.$$

In our model, rotational latency and transfer time are grouped together in the single parameter *DiskConstant*. In today's disk technology, seek times are nonlinear with seek distance[3, 9]. Accordingly, we use the following function for the access time:

$$Access(n) = DiskFactor \times \sqrt{n} + DiskConstant$$

Using the values from Table 5-1 and assuming an average seek distance of 333 tracks, the average access time for our modeled device is 26 ms. The average access time will be used in the construction of individual request deadlines.

Table 5-2 Workload Parameters

Parameter	Meaning	Base Value
Read_Rate	Read arrival rate	
Write_Rate	Write arrival rate	
Min_Slack	Min slack for reads	10 ms
Max_Slack	Max slack for reads	100 ms

5.2 Workload Model

The names and meanings of the parameters that control the workload characteristics are shown in Table 5-2. Requests for I/O service arrive from an open source with exponentially distributed inter-arrival times with mean arrival rates denoted by *Read_Rate* and *Write_Rate* for read and write requests respectively. Each request needs to access a single track which is chosen uniformly from the range [1, *Tracks*]. If the request is a write, then the request is placed in the k-buffer. The recording of write missed deadlines (buffer overflows) and the setting of the write event deadline are done as explained in Sections 2 and 3. If the request is a read, then it is placed in a queue and its deadline is chosen as follows. A slack time is chosen uniformly from the range [*Min_Slack*, *Max_Slack*]. The equation for computing the deadline is:

$$Deadline = Arrival\ time + Average\ access + Slacktime$$

In reality, deadlines can be unreasonable, or impossi-

ble to meet. In our experiments we want to avoid scenarios where deadlines are either, (1) so slack that any scheduling algorithm will meet them, or (2) so tight that no algorithm could meet them. Thus we tried to choose deadlines that leave some room for "intelligent" scheduling. Our experiments will then show which algorithms have this "intelligence."

The base values are not meant to represent a particular workload but were selected as reasonable values within a range. Also our experiments vary the values of the parameters to learn how the algorithms perform under different workload characteristics.

5.3 Data Generation and Metrics

In the following sections we discuss some of the results from the experiments that we performed. Due to space considerations we cannot present all of our results but have selected the graphs which best illustrate the differences and performance of the algorithms. For each experiment we ran the simulation using 40 different random number seeds. Each run continued until 3000 I/O requests were processed. Numerous performance statistics were collected and averaged over the 40 runs.

The primary metric that we use to measure performance is percentage of missed deadlines. Recall that our goal is to schedule I/O requests so that they meet their *individual* response time goals, or deadlines. Measuring the percentage of requests that miss their deadlines is a good performance metric for this goal. Since our model contains two types of deadlines, we measure the performance of each separately using the following equations:

$$\%\textit{Missed Read Deadlines} = \frac{\textit{Missed Read deadlines}}{\textit{Number of Reads serviced}} \times 100$$

$$\%\textit{Missed Write Deadlines} = \frac{\textit{Missed Write deadlines}}{\textit{Number of Write arrivals}} \times 100$$

6 Results for Read Requests Only

In this set of experiments we studied performance behavior under a workload that contained only I/O requests with deadlines, thus no *k*-buffer was used. Deadlines were chosen according to the method described in Section 5.

6.1 Experiment 1: *Min_Slack* =50, *Max_Slack* =50

In the first experiment, the *Min_Slack* and *Max_Slack* parameters were both set to 50 ms. Thus each request had a deadline that was approximately 76 ms from its arrival time (average access plus 50 ms). This also guarantees that arriving requests have later deadlines than requests already in the queue. The parameter *Read_Rate* was varied from 22 requests per second to 40 requests per seconds in increments of 2. Figure 6-1 graphs %Missed Read Deadlines for all six scheduling algorithms. Because of the way that deadlines are assigned, FCFS and ED have exactly the same behavior. The data for these two algorithms is omitted for arrival rates greater than 36 since these two algorithms are unable to meet the throughput demands due to inefficient use of the disk.

These parameter settings describe a very difficult workload where many deadlines are missed. One could argue that such a scenario is unrealistic. However, we believe that for designing real-time schedulers, once must look at precisely these high-load situations. Even though they may arise infrequently, one would like to have a system that misses as few deadlines as possible when these peaks occur.

The results show that FD-SCAN consistently has the best performance across all load settings. At the highest setting, FD-SCAN misses approximately 6.5% fewer deadlines that SSTF, the second best algorithm. This represents a performance improvement of about 15%. The SCAN and D-SCAN algorithms are the next best with SCAN being slightly better.

The SSTF and SCAN algorithms work well because they move the disk head efficiently and thus use the disk resource efficiently. The mean seek distance, and thus mean response time, is significantly lower for these two algorithms, Figure 6-2. Importantly, the mean seek distance decreases significantly as the load increases. This is not surprising since it is exactly how these algorithms were designed to work. The SSTF and SCAN algorithms are excellent baseline algorithms to try to beat precisely because they use the disk resource efficiently. However, they schedule the requests randomly with respect to deadlines. Our goal is to learn which algorithms can beat SSTF and SCAN by doing intelligent deadline scheduling, and yet still use the disk efficiently

In contrast to SSTF and SCAN, FCFS and ED move the disk arm very inefficiently, performing a random seek for every request. As the load increases, these two algorithms are unable to maintain throughput.

The two real-time algorithms D-SCAN and FD-SCAN try to do intelligent deadline scheduling and move the disk arm efficiently. In this experiment, Figure 6-1, we see that D-SCAN performs similarly to SCAN. Because of the way deadlines are assigned, D-SCAN scans the disk in much the same way as SCAN. First, recall that ordering requests by deadline is equivalent to ordering by arrival time (this is only for the case where *Min_Slack = Max_Slack).* Consider the disk

head at some point during a scan. The requests behind the head (in the wake of the scan) will be recent arrivals. The requests lying in front of the head will contain some recent arrivals but more older requests as well. The scan direction will not change until these older requests are serviced. Once they are, the oldest requests are now at the other end of the disk and the scan reverses direction. Because a newly arrived request cannot change the direction of scan, D-SCAN operates much like SCAN.

The FD-SCAN algorithm has the best performance because it gives high priority to requests with feasible deadlines. Thus FD-SCAN may change direction to service a recent arrival if it determines that the older requests lying before it have unfeasible deadlines. When all requests have unfeasible deadlines, FD-SCAN defaults to SSTF which is an efficient way to service the requests and empty the queue as rapidly as possible.

6.2 Experiment 2: *Min_Slack* =10, *Max_Slack*=100

In a second experiment we set *Min_Slack* = 10 ms and *Max_Slack* = 100 ms. This now allows a significant range of deadlines. The load was varied as it was in the first experiment. Figure 6-3 graphs %Missed Read Deadlines for the six algorithms. This time we see that ED misses the fewest deadlines when the arrival rate is low. At this rate, all algorithms will have high mean seek distances because mean queue length is small. Therefore ED is not handicapped, relative to the other algorithms, by its high mean seek time. However, as the load exceeds 32 requests per second, performance deteriorates swiftly due to inefficient use of the disk.

Another interesting observation is that D-SCAN performs better than SSTF and SCAN when the load is low and just as well when the load is high. The variability in deadlines allows D-SCAN to make intelligent choices and still use use the disk efficiently. At the higher load settings, FD-SCAN performs better than all other algorithms.

In Figure 6-4 we graph the Mean Tardy Time for all six algorithms. (Only requests that miss their deadlines contribute to this metric.) The performance of ED is interesting since it has the lowest Mean Tardy time when the load is low, but a very high Mean Tardy time when the load is high. This is characteristic for ED; it can meet most deadlines when the load is low but it rapidly saturates and misses most deadlines, by a lot, when the load is high. Similar remarks apply to FCFS, although it does not perform as well as ED.

Algorithms SSTF, SCAN and D-SCAN have similar mean tardy times because they do not miss many deadlines and they tend to limit the maximum response time experienced by a request. The sweeping behavior of both SCAN and D-SCAN bounds repsonse time, eve for requests at the edge of the disk. Algorithm SSTF also bounds response time because requests are uniformly distributed along the disk surface, and hence the head does not get "stuck" in one area of the disk.

Interestingly, FD-SCAN has the highest Mean Tardy time, Figure 6-4. Although FD-SCAN misses the fewest deadlines (see Figure 6-3), it misses them by a greater amount. The Mean Tardy time is greater because requests with unfeasible deadlines can wait for a long time before they are serviced. In fact, requests with unfeasible deadlines are only serviced when they lie between the disk head and the current scanning target, whenever a request with a feasible deadline exists. Thus a request, A could lie only one track from the current head position but not be serviced because there is no request B with a feasible deadline to "pull" the head over request A. Contrast this to SSTF which would service A because it is so close.

The preceding discussion suggests that we examine how the algorithms perform for certain areas of the disk. Figure 6-5 graphs the distribution of missed deadlines for 10 disk areas at *Read_Rate* = 36 requests per second. Disk area 1 contains tracks 1-100, area 2, 101-200, and so on. The FCFS and ED algorithms are very fair, each area accounts for 10% of the total number of missed deadlines. The SCAN algorithm is also relatively fair. The FD-SCAN algorithm is remarkably unfair to requests in the two outermost disk areas. A center area of the disk accounts for only 6% of the missed deadlines while each outermost area accounts for more than 16%.

Since requests on the outer tracks require longer seeks, they are more likely to miss their deadlines. The SSTF, SCAN, D-SCAN and FD-SCAN policies all favor the center tracks. However, SCAN regularly services the outer tracks. Requests that arrive while the head is there will be serviced quickly and meet their deadlines. The D-SCAN policy only services the outer areas when the earliest deadline request is located there. The FD-SCAN policy discriminates against the outermost areas severely, scanning them only when a request with a *feasible* deadline is there. However it is very bad only to the outermost areas. The missed deadline distribution for the next to outermost areas is comparable to SSTF, SCAN and D-SCAN.

7 Results for Read and Write Requests

In this set of experiments, we studied performance behavior under a workload that contained both read and write I/O requests. We are interested in comparing the performance of the algorithms that do not manage the *k*-buffer to those that do manage the buffer. Also we want to learn which algorithms are best overall.

7.1 Experiment 1: Vary *Read_Rate*

In the first experiment *Write_Rate* was set at 10 requests per second while *Read_Rate* was varied from 12 to 30 requests per second in increments of 2. Note that the cumulative arrival rate is the same as in the first set of experiments. The slack parameters had the base values shown in Table 5-2. Figure 7-1 graphs %Missed Read Deadlines for both the traditional and the buffer adapted versions of SSTF and SCAN. The buffer adapted versions use the Space Threshold technique with the threshold set to 1. It is obvious that the buffer adapted versions miss far fewer read deadlines than the traditional versions. Figure 7-2 graphs %Missed Write Deadlines for the same experiment. Note that although the buffer adapted versions do miss some write deadlines, it is less than 2.5% of the total write arrivals. The traditional versions never let the buffer overflow.

Figure 7-3 shows %Missed Read Deadlines for the three real-time algorithms ED, D-SCAN and FD-SCAN, and the three buffer adapted traditional algorithms. The real-time algorithms use the Time Threshold technique for buffer management. The others use the Space Threshold technique. (We do not include the Space Threshold versions of the real-time algorithms because the resulting graph would be too crowded. Also, we found that, for this experiment, the Time Threshold version performed slightly better than the Space Threshold version.) Although ED performs well at lower load settings, its performance quickly deteriorates once *Read_Rate* exceeds 22 requests per second. At the highest rate, ED is no better than FCFS(B). The D-SCAN and SCAN(B) algorithms have nearly identical performance. The second best performer is SSTF(B) and the best is FD-SCAN. At *Read_Rate* = 30, FD-SCAN misses approximately 3% fewer read deadlines than SSTF(B). This represents an improvement of 12%.

Figure 7-4 shows %Missed Write Deadlines for the same six algorithms. The buffer adapted algorithms SSTF(B) and SCAN(B) permit significantly buffer overflows than D-SCAN or FD-SCAN at the two highest load settings. This happens because D-SCAN and FD-SCAN use the Time Threshold technique for managing the *k*-buffer. (We are not saying that all Time Threshold techniques would behave like this, perhaps some would be better. However, the one that we tested, namely the linear technique, exhibits this performance.) When Read_Rate is high there is a greater chance that a read request will have an earlier deadline than D_w, the write event deadline. Thus reads receive priority and more write deadlines are missed. Recall that FD-SCAN missed the fewest read deadlines, Figure 7-3. When the two measures (missed reads and missed writes) are combined in a weighted average, SSTF(B) and FD-SCAN have nearly identical performance.

7.2 Experiment 2: Vary *Write_Rate*

In a second experiment, *Read_Rate* was fixed at 12 requests per second. and *Write_Rate* was varied from 10 to 28 requests per second. The other parameters were unchanged. Figure 7-5 graphs the weighted average of %Missed Deadlines for both reads and writes. Note that all the algorithms that make effective use of the write buffer outperform the traditional versions of SSTF and SCAN. One reason for the lack of differentiation among the better performing algorithms is that they all are missing very few deadlines anyway. It is easy to meet deadlines because the request stream consists mostly of write requests, and meeting write deadlines (keeping the buffer from overflowing) is much easier to do than meeting the individual read deadlines. However, when *Write_Rate* is large enough, buffer overflows become more of a problem. In this experiment, %Missed Write Deadlines increases from less than 1% at 22 writes per second to roughly 6% at 28 requests per second for the algorithms that manage the *k*-buffer. This accounts for most of the increase in %Missed Deadlines in Figure 7-5.

7.3 Experiment 3: Vary *Buffer_Size*

In a third experiment, *Read_Rate* was fixed at 20 requests per second, *Write_Rate* at 16, and *Buffer_Size* was varied from 5 to 14. %Missed Read Deadlines versus *Buffer_Size* is shown in Figure 7-6. First, note that SSTF and SCAN do not change. These algorithms do not manage the *k*-buffer, thus changing the buffer size will not affect %Missed Read Deadlines. Second, all of the algorithms that manage the *k*-buffer perform much better than SSTF and SCAN. These algorithms can delay servicing writes in order to meet the deadlines for reads. The amount that they can delay, and thus the effective advantage to read scheduling, increases significantly as the buffer size increases. Finally, we note that FD-SCAN performs the best with SSTF(B) and D-SCAN tied for second.

Also, no algorithms missed any write deadlines for any of the buffer size settings.

7.4 Experiment 4: Vary *Space_Threshold*

In a fourth experiment, *Read_Rate* was fixed at 16 requests per second, *Write_Rate* at 20, *Buffer_Size* at 10, and the *Space_Threshold* parameter was varied from 1 to 4. (Recall that *Space_Threshold* controls when the emptying of the *k*-buffer is triggered for those algorithms that use the Space Threshold technique for managing the *k*-buffer.) Thus, when *Space_Threshold* = 1, the emptying of the *k*-buffer is triggered only when the buffer is full (free space < 1). When *Space_Threshold* = 4, the emptying of the *k*-buffer is triggered when there are only three empty spaces left.

Figure 7-7 graphs %Missed Read Deadlines for the three traditional algorithms which use Space_Threshold. The FD-SCAN algorithm, which uses Time_Threshold, is also shown. Note that the graph for FD-SCAN is flat since it is not affected by the *Space_Threshold* parameter. The other three algorithms miss more read deadlines as the *Space_Threshold* parameter increases. Although we do not show the graph, these algorithms also miss *fewer* write deadlines. However when *Space_Threshold* = 3, these three algorithms miss less than 1 percent of write deadlines and increasing *Space_Threshold* further does not improve write performance any more. Moreover, the ability to make the best use of all of the *k*-buffer is hampered. Similar behavior was observed for algorithms using Time_Threshold when the function that sets the write deadline was varied from being optimistic to pessimistic.

8 Conclusions

In this paper we have presented a number of new algorithms for scheduling I/O requests with deadlines. The algorithms were evaluated via detailed simulation and compared with traditional scheduling algorithms. One new algorithm, FD-SCAN, had consistently the best performance in a wide variety of experiments.

We also investigated a model for handling read requests differently from write requests. This model buffers write requests in a separate queue from read requests. Two techniques for managing the buffer were examined and both were found to be effective. The overall approach of buffering writes was especially helpful for meeting read deadlines. We believe that systems. e.g., real-time information systems, that produce read requests with deadlines and write requests without, can use this approach to great benefit.

References

[1] Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-Time Transactions: a Performance Evaluation," *Proceedings of the 14th VLDB Conference*, pp. 1-12, 1988.

[2] Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[3] Bitton, D. and J. Gray., "Disk Shadowing," *Proc. 14th VLDB Conference.*, pp 331-338.

[4] Carey, Michael, R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," Computer Science Dept, Univ. of Wisconsin, TR-828, March 1989.

[5] Coffman, E.G., and M. Hofri, "On the Expected Performance of Scanning Disks," SIAM Journal of Computing, Vol. 11, No. 1, Feb. 1982, pp. 60-70.

[6] Coffman, E.G., L. Klimko and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times," SIAM Journal of Computing, Sept. 1972, pp. 269-279.

[7] Geist, Robert and Stephen Daniel, "A Contimuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 77-92.

[8] Schwetman, Herb, "CSIM Reference Manual," MCC.

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

[10] Teorey, Toby J. and Tad B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, Vol 15, no. 3, pp. 177-184.

[11] Wilhelm, Neil C., "An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empircial Model for Disk Accesses," *Communications of the ACM*, Vol 19, no. 1, pp 13-17.

[12] Davidson, S., I. Lee, and V. Wolfe, "A Protocol for Timed Atomic Commitment," *IEEE ICDCS* 1989, pp 199-206.

Acknowledgements

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.

Figure 6-1 ***Min_Slack=50, Max_Slack=50***

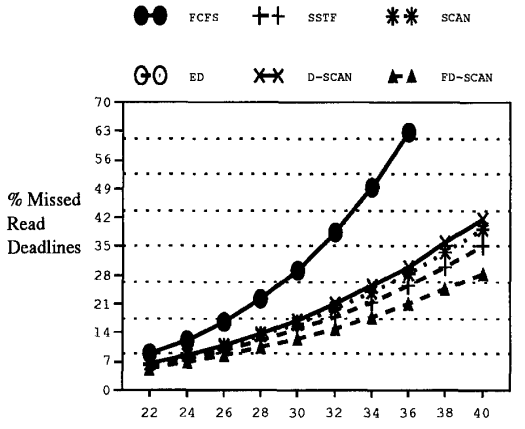


Figure 6-2 ***Min_Slack=50, Max_Slack=50***

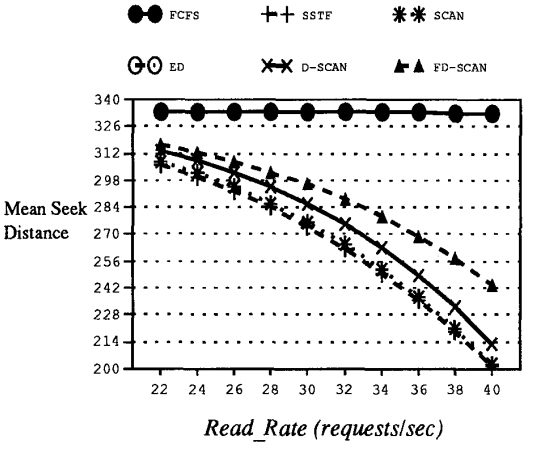


Figure 6-3 ***Min_Slack=10, Max_Slack=100***

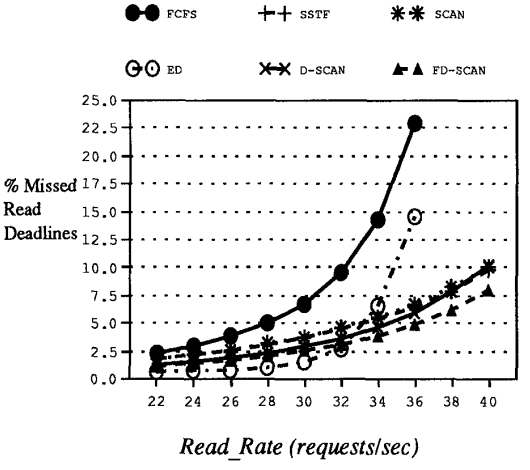


Figure 6-4 ***Min_Slack=10, Max_Slack=100***

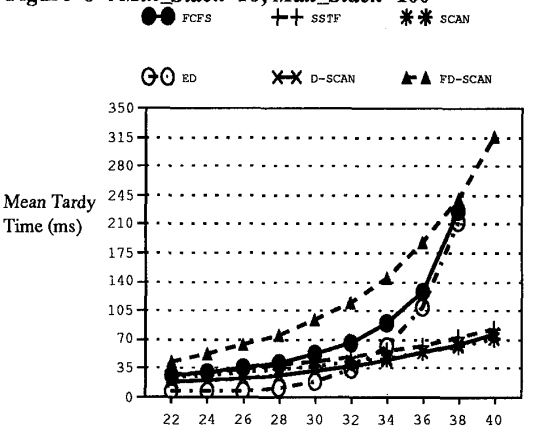


Figure 6-5 ***Min_Slack=10, Max_Slack=100***

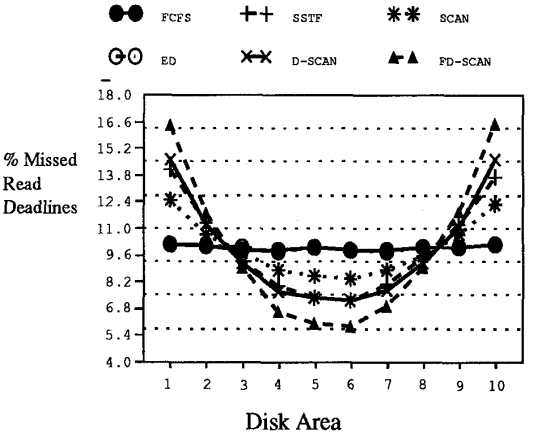


Figure 7-1 Vary ***Read_Rate***

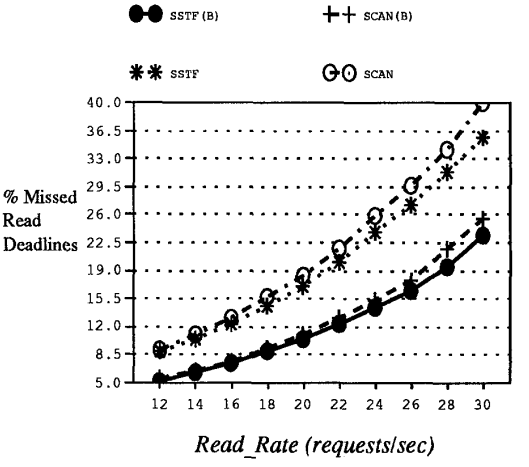


Figure 7-2 Vary *Read_Rate*

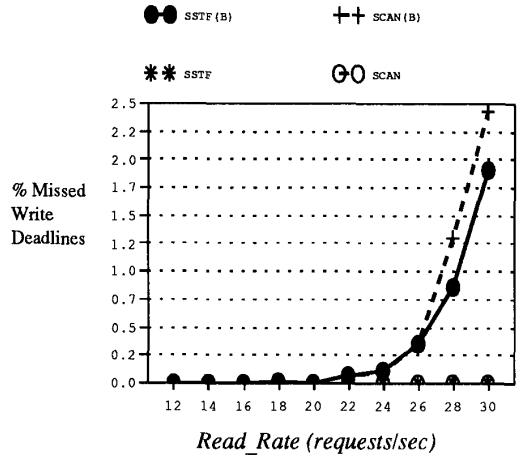


Figure 7-3 Vary *Read_Rate*

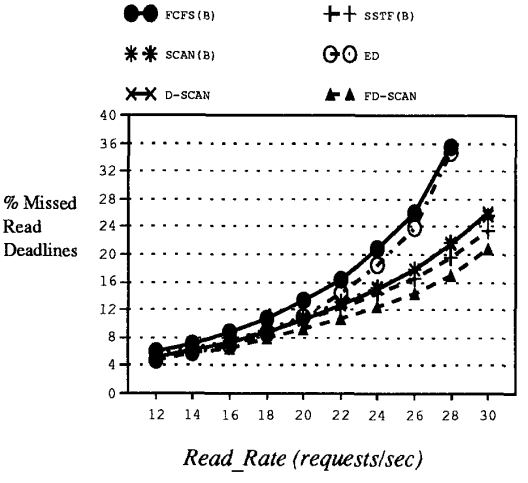


Figure 7-4 Vary *Read_Rate*

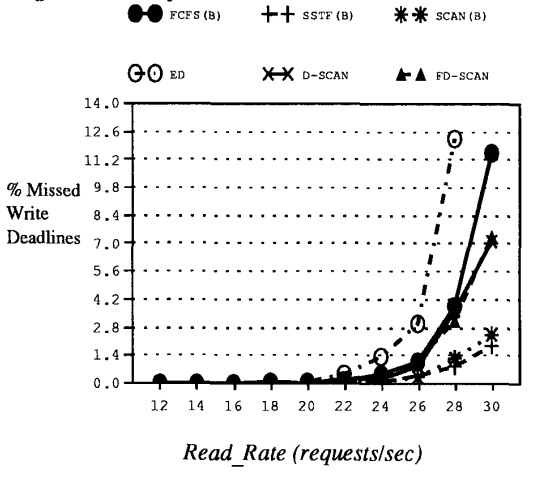


Figure 7-5 Vary *Write_Rate*

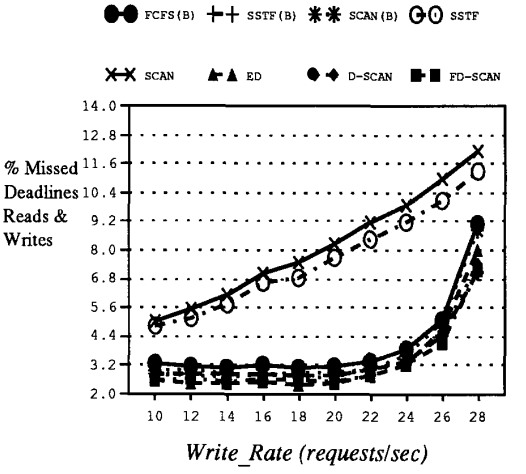


Figure 7-6 Vary *Buffer_Size*

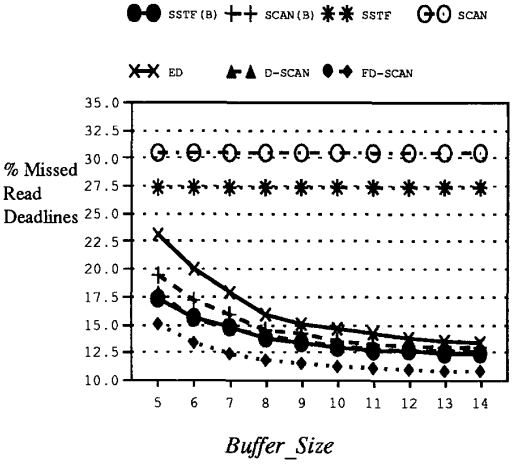


Figure 7-7 Vary *Space_Threshold*