Disk scheduling algorithms based on rotational position

David M. Jacobson and John Wilkes

Concurrent Systems Project, Palo Alto, CA

Technical report **HPL–CSP–91–7rev1**

26th February 1991, revised 1st March 1991.

(Issued electronically 10th May 1995.)

When sufficient processor power is available, disk scheduling based on rotational position as well as disk arm position is shown to provide improved performance. A taxonomy of algorithms based on Shortest Access Time First (SATF) have been developed, and several of them have been explored through simulations. The access-time based algorithms match or outperform all the seek-time ones we studied. The best of them is Aged Shortest Access Time First(*w*), or ASATF(*w*), which forms a continuum between FCFS and SATF . It is equal or superior to the others in both mean response time and variance over the entire useful range.

Authors' notes

After writing this paper, we came across the work of Margo Seltzer, Peter Chen, and John Ousterhout [Seltzer90]: a set of very similar ideas that had been published a year before we wrote this report. We'd appreciate it if people citing our work also cited theirs.

Unfortunately, the original document used physical cut and paste for its graphics, so this electronic version includes scanned images of them.

There are a few changes to the original document; they are marked by a vertical bar in the left margin, like this.

1 Introduction

The most widely known algorithm for scheduling disk requests is the SCAN algorithm, in which the disk arm is alternatively moved from one edge of the disk to the other and back. As the arm crosses each cylinder, requests for it are serviced. The goal of this and most other disk scheduling algorithms is to reduce disk arm motion, which makes sense when seek times are the dominant performance factor, when it is impossible to know the rotational position, or when the processing power or interrupt structure of the host computer limits the amount or responsiveness of computation in the disk driver.

Two new conditions require the reconsidering of this design principle.

First, over the years the rotational speed of disks has increased only slightly, while the full stroke seek time has been shortened significantly. The following table shows some sample values for a late 1960's vintage drive and some more modern Hewlett-Packard ones (the smaller the diameter, the later the date of introduction): [1]

disk	*diameter*	*rotation speed*	*full-stroke seek time*	*rotations*
IBM 2314	14 in	3600 RPM	135 ms	8.1
HP 7935	14 in	2700 RPM	72 ms	3.2
HP 7937	8 in	3600 RPM	38 ms	2.3
HP 97560	5.25 in	4002 RPM	25 ms	1.7
HP C2235	3.5 in	3600 RPM	21 ms	1.3

Second, processor speed has increased tremendously, and the availability of abundant cycles enables us to consider more computationally-intensive scheduling algorithms.

Finally, this work was initially motivated by the needs of the DataMesh project at Hewlett-Packard Laboratories. This research effort is developing a parallel storage server from a smart storage surface: an array of closely-coupled disk:processor modules connected together with a fast, reliable mesh. Each module will have a disk, a 20 MIPS processor and around 16MB of primary RAM. The close coupling of CPU and disk means that the processor knows the rotational position of the disk at any time and when a request will be executed since there is no channel contention to introduce uncertainty.

These conditions make it practical to design a disk scheduling algorithm based on rotational position as the primary consideration. Although begun in the context of DataMesh, we believe the ideas are directly applicable to other systems, including device drivers and smart disk controllers.

2 Previous work

The simplest algorithm is to service requests in the order that they arrive, or *First Come First Served* (FCFS). This algorithm has poor performance for all but the lightest of loads, since it wastes a lot of time moving between areas on the disk relative to the time spent actually transferring data.

[1] Seek time data for the IBM 2314 was inferred from a seek time versus distance graph in [Frank69].

Better is to scan the request queue for the request that is nearest to where the head is positioned, and process that next. Traditionally, “nearest” is calculated from the difference in cylinder numbers, and this technique is known as *Shortest Seek Time First* (SSTF).

During periods of very high load, and particularly when many requests are arriving for the same area on the disk, the arm may “stick” in one region, leaving others requests waiting for a long time. This is known as the starvation problem. Even in the absence of complete starvation, this phenomenon increases the service time variance.

The SCAN algorithm, first proposed by Denning [Denning67], sweeps back and forth across the disk stopping at each cylinder with pending requests. A variation of SCAN is to sweep in only one direction, and when the end of the disk is reached, to seek back to the beginning.[2] Wong has shown that the bidirectional SCAN is at least as good as a unidirectional one for independent arrivals [Wong83], although the 4.3BSD version of Unix used a one-way algorithm because it gave better behavior with readahead [Leffler89].

SCAN also suffers from starvation, but to a lesser degree than SSTF. Various authors have proposed adaptations to overcome this problem. One approach with SCAN is to delay recent arrivals until the next sweep [Gotlieb73]. It is purported that some UNIX[3] system implementations control starvation by limiting the number of requests from any one cylinder that will be serviced before moving on.

The performance of FCFS, SSTF and SCAN has been extensively studied in the literature (e.g. [Denning67,Coffman72,Gotlieb73,Oney75,Wilhelm76,Hofri80,Coffman82]).

Geist and Daniel have proposed a continuum of algorithms called *V(R)*, where *R* is a parameter [Geist87]. The idea is to pick the next request according to SSTF, except to add a penalty of R times the total number of cylinders for reversing direction. It can be seen that *V(0)* is SSTF and *V(1)* is SCAN . They suggest *V(.2)* as a good compromise that performs better than SCAN, but avoids the high variance and starvation difficulties of SSTF .

Although most of the time requests arriving at a disk are the only ones in the queue, the presence of very large queue lengths is surprisingly common. Figure 1 plots the queue lengths experienced during a ten minute trace from a time sharing system.

3 Disk model

The analysis of the algorithms to be presented are parameterized according to the performance of an HP 97560 SCSI disk drive.[4] The HP 97560 is a 5.25" drive with 10 platters (19 data surfaces), 1964 physical cylinders, and 72 sectors (512 bytes each) per track, for a total formatted capacity of 1.37GB. The platters rotate at 4002 RPM; settling time for switching between tracks in the same cylinder is about 2.5 ms.

2. The notation for these algorithms is confused. Usually the two-way algorithm is called SCAN, but sometimes the word “elevator” is used. [Leffler89] uses “elevator” for the one-way algorithm.
3. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.
4. These figures presented here are derived from measuring a sample drive, and may not agree exactly with the published product specifications.

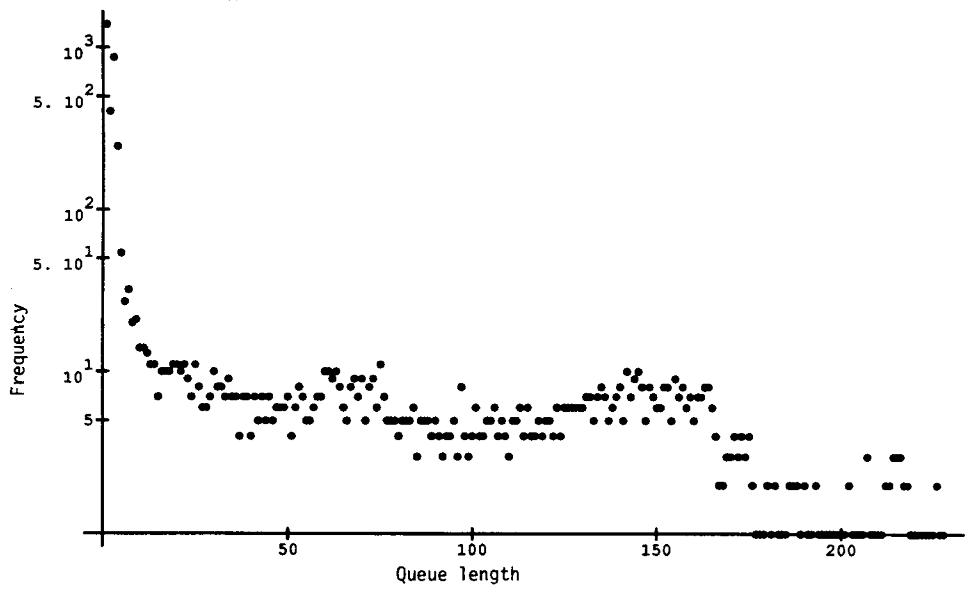


Figure 1. Frequency of different queue lengths.

During long seeks, most modern disks accelerate the disk arm up to some maximum velocity, then let it coast at constant velocity, then decelerate back to zero, and finally settle onto the desired track. For short seeks the arm accelerates up to the halfway point, then decelerates and settles, giving a seek time of the form $t = a + b\sqrt{s}$, where s is the seek distance in cylinders. For long seeks (ones where the arm reaches its terminal velocity), the seek time has the form $t = c + ds$.

Functions of the form given above were fit to the measured HP 97560 performance data, with the following results for seek time in milliseconds:

$$T_S = \begin{cases} 3.24 + 0.40\sqrt{s} & \text{if } s \le 383 \\ 8.20 + 0.0075\, s & \text{if } s > 383 \end{cases}$$

This is shown graphically in Figure 2.

4 Rotationally-sensitive scheduling algorithms

The total access time, which combines seek and rotational latencies, has traditionally been represented as $T_A = T_S + T'_R$, where T'_R is the rotational delay experienced once the head has arrived at the correct cylinder. However, for modern disks, the seek time is often less than this rotational latency, so it is quite natural to look for a formula that shows the combined cost with the rotational latency as the basis.

It is convenient to scale access times by τ, the time taken for the disk to rotate through one sector (0.21 ms for the HP 97560), since all transfers have to start and stop on sector boundaries. Using these units, the following equation gives the access time (repositioning cost) for a head movement

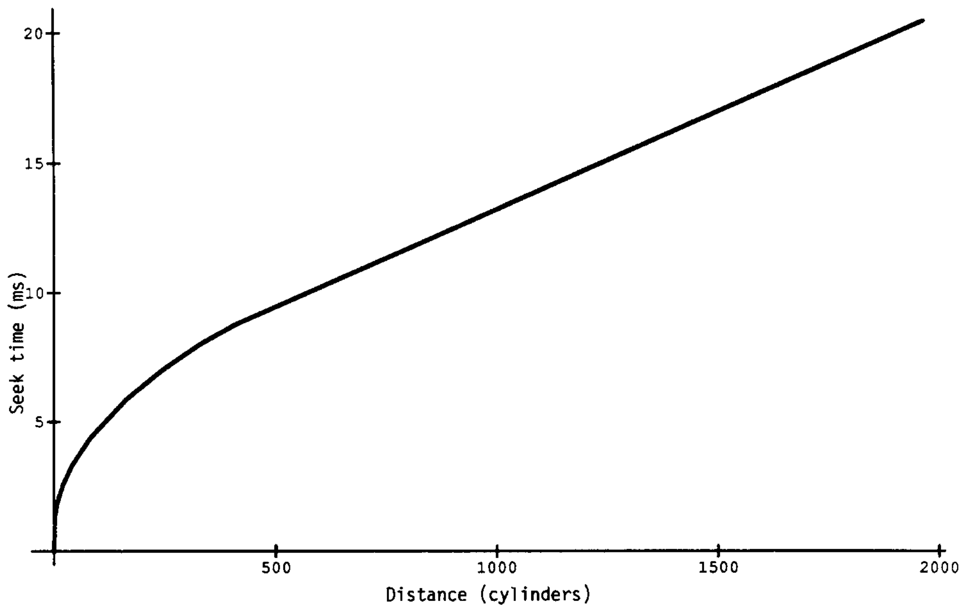


Figure 2. Seek time versus seek distance in cylinders for the HP 97560.

that traverses *c* cylinders and *s* sectors ($0 \le s < S$, where *S* is the number of sectors per track) from the current head position and where $T_S(c)$ is the seek time as shown above:

$$T_A(c,s) = s + S\lceil(T_S(c)/\tau - s) \div S\rceil$$

To demonstrate graphically the significant effect that rotation latency has on the behavior of the access time, figure 3 plots access time as a function of cylinder (horizontal) and sector (vertical) offsets. Notice the large area, 39.7% of the disk, that is accessible in just one rotational latency. This suggests strongly that rotational delay should be considered on an equal footing with seek time in the disk scheduling decision.

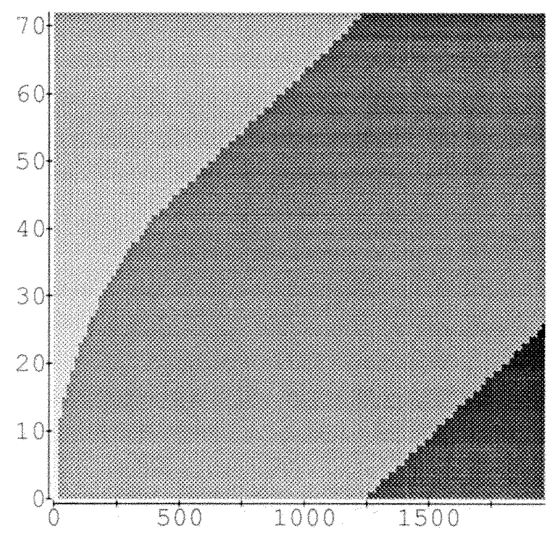
The remainder of this section introduces a scheduling policy that does just this; subsequent sections discuss variations on it and present the result of simulations of their performance.

4.1 Optimal Access Time

When a request is to be given to the disk, the *Optimal Access Time* (OAT) algorithm considers all possible combinations of the requests in the queue, and performs an optimal scheduling decision for all the requests. Unfortunately, there is no known efficient algorithm for optimal scheduling.[5] As a result, OAT is only useful as a theoretical upper bound on the scheduling performance.

Fortunately, “greedy” algorithms—which make the best first choice, then the best choice out of what remains, and so on—are good approximations to the fully optimal calculation. To test this experimentally, we issued 100 clumps of random requests to a greedy scheduler and an optimal scheduler, and then compared the average execution time for the resulting schedules. For clumps of length 4 to 6, the greedy algorithm resulted in schedules only 5% to 8% longer than optimal.

[5]. This problem is a special case of the well-known Traveling Salesman Problem, whose corresponding decision problem (Does there exist a schedule that can be completed within a stated bound?) is NP-Complete. Many special cases have been found to be NP-Complete[Garey79]; we conjecture that this one is also



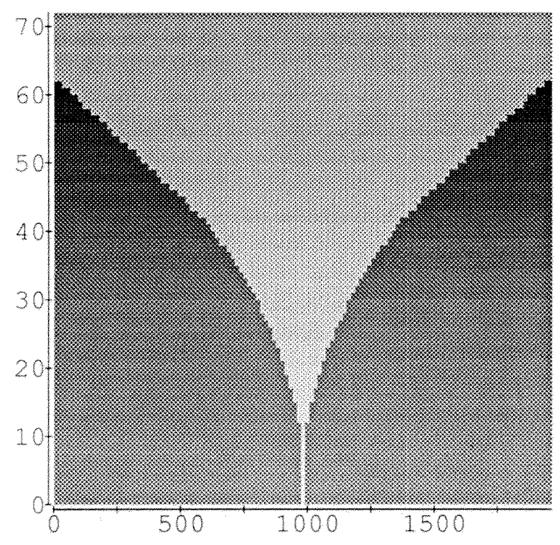


Figure 3. Access time as a function of cylinder and sector from sector zero of (left) cylinder zero and (right) the middle cylinder. The times are uniform horizontally in each region.

We then extended this simple experiment to consider a "short-sighted" OAT algorithm that optimally scheduled batches of 6 requests (taken from the head of the queue). We compared it with the full greedy algorithm that looks at all remaining requests on the queue; a "short-sighted" greedy algorithm that looked at only the first 6 requests on the queue; and a sliding window OAT algorithm that pulled in the next request from the queue as soon as the first had been scheduled and dispatched. The following table shows the results, relative to the full greedy algorithm.

algorithm	*relative performance*
short-sighted OAT	1.84
sliding window OAT	1.75
short-sighted greedy	1.73
full greedy	1.00

Clearly, examining the entire queue is an enormous win, even compared to doing optimal scheduling of mid-sized batches of requests. The reason is that OAT will often pick a more distant first request in order to gain time later on. But the servicing of requests does not occur in isolation: new requests are arriving during service, and the batching algorithm has no way of taking advantage of them, even if they produce a better overall schedule.

Given these results, we did not pursue the OAT algorithms further.

4.2 Shortest Access Time First

The *Shortest Access Time First* (SATF) algorithm is an approximation of OAT using a greedy scheduler with access time as the scheduling parameter, rather than seek distance. It is a direct analogue of SSTF.

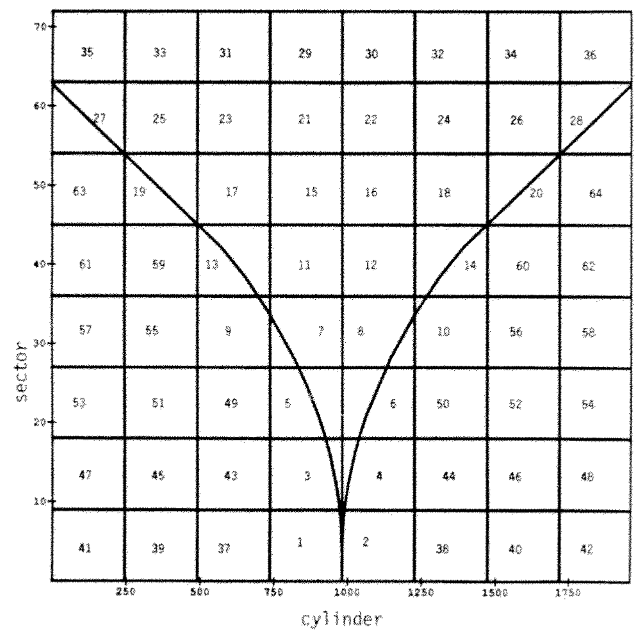


Figure 4. The disk divided into cells with an example ordering.

The implementation of SATF follows closely that of SSTF: the queue of pending requests is scanned, and the access time T_A calculated for each request given the known head position (if the disk is idle), or the calculated head position at the end of the current request.[6] This is easily implemented with the assistance of a small static array holding the seek times for each possible seek distance.

We can reduce the number of requests that get scanned by dividing the disk up into a number of bins (see figure 4), each of which encompasses a connected region of the access-time graph shown in figure 3. When a request arrives, it is put into the bin associated with the target portion of the disk. When the time comes to calculate a schedule, the bins are searched in an order that depends on the current head position. This order can be pre-computed and stored in a small array indexed by head position. Figure 4 shows the bin search order for when the head is at sector zero on the middle cylinder.

Some cells lie on both sides of the head trajectory (e.g. some requests can be accessed in one revolution, some need two). To cope with this, a threshold value $T_{worst}(cell)$ is calculated for each cell that indicates the worst possible time for a "good" request from this cell. If a request in a split cell is met that is below this limit, it is the best one to do next; otherwise the search continues.

These split cells could be numbered twice: once for their "good" requests, once for their "bad" ones. We can avoid the second scan by keeping track of the best request seen so far (best)—even if it came from the "bad" side of a split cell. If a better request is seen, it replaces best. The search terminates as soon as a value or cell is encountered such that $T_A(best) \leq T_{worst}(cell)$, which will

[6] We assume for simplicity that the disk itself does not do command queueing, or, worse, reordering. We also ignore the effects of spared sectors and tracks.

always happen by the last cell of the scan. The list of bin orderings and worst values is quite small, and can be held in 8KB for a 64-element grid.

The algorithm in figure 5 illustrates this in more detail (Taccess(a->b) is the time to access request b given that the head is at position a).

5 Reducing starvation

Starvation happens because new requests continue to arrive and be chosen for servicing before some of the ones already in the queue. One way to avoid this is to refuse temporarily to admit new requests to the queue: instead, they are accumulated off on the side until some future time at which they are be considered for scheduling. Unfortunately, as the pool of requests to be scheduled decreases in size, the likelihood that the disk will be kept optimally utilized declines: the population of candidates becomes too small to allow selection of requests that fit neatly into the “holes” in the schedule. Maintaining high disk efficiency requires that the pool of candidates for scheduling be as large as possible; but avoiding starvation requires that new requests not be allowed to indefinitely delay old ones, even if this reduces the efficient use of the disk. Result: some compromise is required.

As with SSTF, SATF is subject to starvation. The conditions that engender it are somewhat more complex, because “nearness” is now a dynamic function of rotational position, but the effect is the same. This section introduces a number of variations on SATF to address the starvation problem by limiting the time any request can remain unserved. This will both reduce the variance of the service time and eliminate starvation at some expense in maximum throughput.

The first approach presented here *responds* to starvation by forcibly scheduling the oldest request. Variations on this theme mollify the tendency of this approach to degenerate to FCFS under high-load conditions: the very ones in which starvation is likely.

The second approach *avoids* starvation by batching requests, and preventing new ones from being considered until the first batch has been processed. Variations on this theme allow some new requests to be scheduled, as long as they do not delay the execution of the current batch.

```
bestsofar := nil;
h := current head position;
for each cell in the order given by the current head position do
    for each request r in this cell do
        if Taccess(h->r) <= Taccess(h->bestsofar) then
            bestsofar := r;
        fi;
    endfor;
    if Taccess(h->bestsofar) <= Tworst[this_cell] then
        exit returning bestsofar;
    fi;
endfor;
exit returning bestsofar;
```

Figure 5. Efficient SATF algorithm

```
requiredreq := nil;
h := current head position;
if oldest is too old then
        z := the oldest request;
else
        z := nil
fi;
bestsofar := z;
for each cell in the order given by the current head position do
        for each request r in this cell do
                if ((z=nil) or (Taccess(h->r) + length(r) + Taccess(r->z) <= Taccess(h->z)))
                        —make sure we can still reach z without losing a revolution
                then if (Taccess(h->r) <= Taccess(h->bestsofar)) then
                        bestsofar := r;
                fi;
                fi;
        endfor;
        if T(bestsofar) <= worsttime[this_cell] then
                exit returning bestsofar;
        fi;
endfor;
exit returning bestsofar;
```

Figure 6. SATFUF algorithm

Finally, the third approach also *avoids* starvation by modifying the criteria used to select the "most deserving" request to be processed by adding an age-related amount to the calculated access time in selecting whether it should be selected or not.

5.1 Forcibly scheduling the oldest request

The *Shortest Access Time First with Urgent Forcing* (SATFUF) algorithm, shown in figure 6 forcibly schedules the oldest request to be executed immediately once it has aged too much. To minimize the effect on disk throughput any other requests that can be scheduled in front of it are so scheduled—but only if this doesn't delay the oldest request. All symbols are as before, with the addition that length(r) is the length of request r in sectors.

Simulations showed that SATFUF is not a good algorithm: not very many requests actually get spliced in front of the oldest request, and once the starvation behavior is triggered, the performance approximates that of FCFS. Typically, this happens once the arrival rate has exceeded the capacity of FCFS to clear the backlog, and the response time never recovers. As a result, we dropped pure SATFUF from the set of simulated algorithms.

One modification to SATFUF is to forcibly select N, rather than one, of the oldest requests, and schedule them—again, slotting in additional requests if these can be accommodated without delaying the schedule. This is called *Shortest Access Time First with Urgent Forcing*(N), or SATFUF(N). With N=1, this is SATFUF; with N=∞, this is one of the family of batch algorithms discussed in section 5.2.

A second approach to improving SATFUF starts out as before, by selecting the oldest request once it has aged too much, but then inserts a few (e.g. 5) whole-rotations worth of slack before it has to be executed. As many requests as possible are then scheduled in the intervening time gap. Again,

```
static deadline initially nil;
z := oldest request;
t := current time in sectors;
h := current head position;
if deadline = nil then
        deadline := t + Taccess(h->z) + S*D;
fi;
bestsofar := z;
for each request r do
        if t + Taccess(h->r) + length(r) + Taccess(r->z) <= deadline
                -- make sure we can still reach oldest by deadline
        and H(h,r,z,deadline-t) > H(h,bestsofar,z,deadline-t)
        then
                bestsofar := r;
        fi;
endfor;
if bestsofar = z then
        deadline := nil;
fi;
exit returning bestsofar;
```

Figure 7. SATFGDF algorithm

the number of requests that is swallowed up can be parameterized; this algorithm is called *Shortest Access Time First with Delayed Forcing(N)*, or SATFDF(N).

There are several possible variations on how to schedule the "inserted" requests (the ones that aren't being forcibly scheduled). For example:

1. Use perfect optimization. Unfortunately, with even two rotation times of delay, the entire request queue is eligible for consideration, and the factorial behavior renders this approach computationally intractable.
2. Approximate the optimal assignment with the greedy algorithm. Because the greedy algorithm only uses the deadline (when the forced request is to begin) to determine its cutoff point, it has no knowledge of where it is trying to end up. Ideal would be for the head to be positioned close (in access time) to the start of the forcible request; in practice, a uniformly-distributed random repositioning delay is likely to be introduced.
3. Use two greedy algorithms: one working forward from the current head position; the other backwards from the known end point. Once both have been computed, the first portion of the forward algorithm is spliced together with the last portion of the backwards one at a point that minimizes the resulting "dead time".
4. Try to "coax" the head toward the forced request, by using a heuristic to guide the greedy algorithm in its selection. This algorithm is called *Shortest Access Time First with Guided Delayed Forcing*, or SATFGDF.

Note that all but variation 2 require examining the entire queue at every scheduling step.

In the pseudocode in figure 7 for SATFGDF, the "static" variable deadline survives from one request to another. D is the number of revolutions the oldest is scheduled into the future (typically 4 or 5). H is the hueristic function, to be described below. The larger the value it returns, the more desirable the choice is.

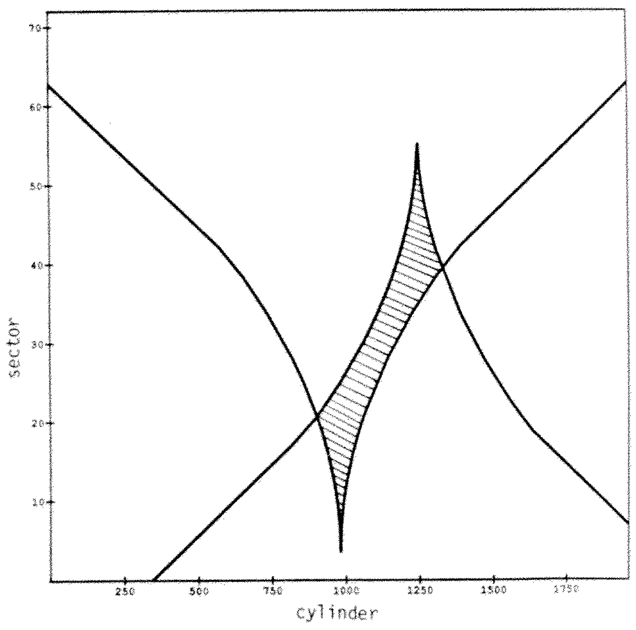


Figure 8. The “area” of accessible disk space

There are several possibilities for the heuristic function (a function which when maximized is likely to yield a good schedule). One is the number of additional requests that could be reached from the end of request r but still reaching z in the remaining time. Unfortunately this would make the complexity of selection of the next request quadratic in the number remaining requests, and linear is complex enough! Simply returning Taccess(h->r) is virtually identical to SATFDF and does nothing to coax the head toward the oldest request. One heuristic that looks appealing is to use the “area” of the disk that can be reached in the remaining time. Figure 8 illustrates the idea.

A plot of the “area” heuristic curves by distance to the deadline, plotted against the starting cylinder number, looks rather like a family of parabolas. Rather than using the area metric value directly (since it is moderately expensive to compute or store), an approximation was developed that ignored the effect of being near the edge of the disk, and used a quadratic function. For each distance the area heuristic was computed, and a quadratic function fit through it (figure 9). The coefficients were stored in arrays indexed by the remaining time in sectors. (The linear term is always zero, since these quadratics are centered about 0.)

5.2 Batch algorithms

Batch algorithms are ones that *prevent* starvation by temporarily preventing new requests from joining the queue and thereby delaying old ones indefinitely.

The batch algorithms described here can be used continuously, or in a two-mode fashion: invoked only when starvation has been observed to bring it into check and prevent further occurrences. The two-mode behavior attempts to benefit from the high throughput of SATF, while limiting the damage caused by starvation by use of the batch technique.

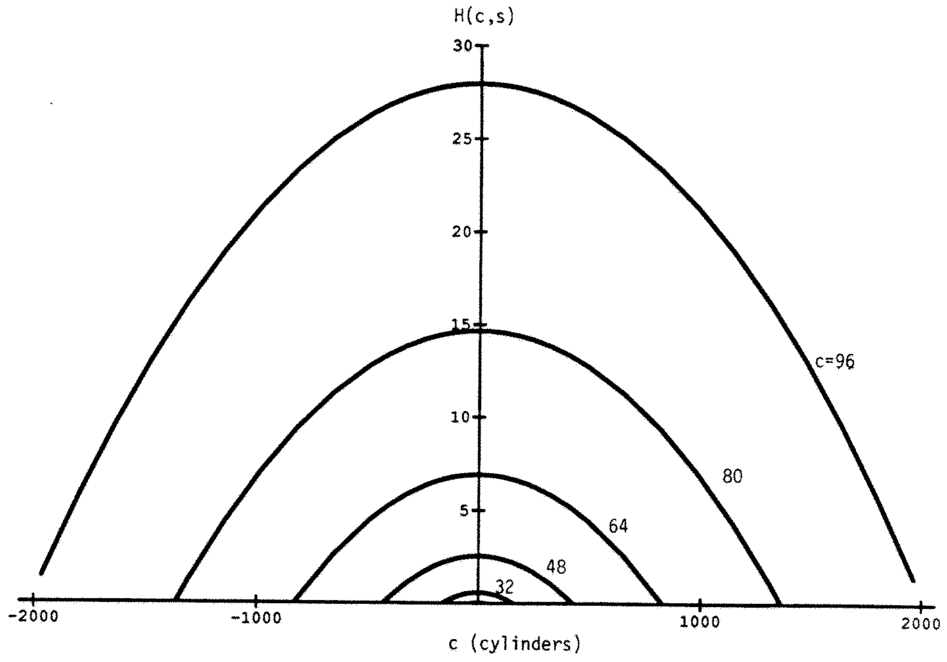


Figure 9. Approximations to the “area” heuristic for various times until the “deadline”.

The simplest algorithm is *Batched Shortest Access Time First* (BSATF). It operates by processing all the requests that are currently on the queue to completion before admitting any more. It uses a greedy algorithm for the optimization.

The second variant relaxes the “no new requests” rule. It is called *Leaky Batched Shortest Access Time First*, or LBSATF. As with BSATF, a batch is acquired and scheduled using a greedy algorithm, and the projected end time is remembered as a deadline. If a new request arrives, it is added to the batch if a schedule can be found that will complete it as well as all the existing requests before the existing deadline. If not, the request is put aside until the next batch is taken. Like BSATF, forward progress is guaranteed, but some new requests may also be accommodated.

It can be seen that SATFUF(N) is a variety of two-moded batch algorithm.

5.3 Modified selection criteria

To avoid starvation, all that is needed is for the oldest requests to be removed at some strictly positive rate from the queue. The choice of rate influences how long a request will be able to sit on the queue: the higher the rate, the shorter the time that a request will be stuck. One way to do this is to weight the scheduling algorithm’s choice of “most deserving” request to take account of their age.

The *Aged Shortest Access Time First(w)* or ASATF(*w*) algorithm calculates the merit *M* of a request in the queue as $M = wT_{age} - T_A/\tau$. The calculated access time is modified by a value proportional to the time, T_{age}, the request has been waiting in the queue. (The units of w are sectors per second.)

The weighting factor w can be varied from zero (pure SATF) to nearly infinity (pure FCFS). In this sense it is similar to the *V(R)* algorithms of [Geist87], which covers a spectrum between SSTF and SCAN .

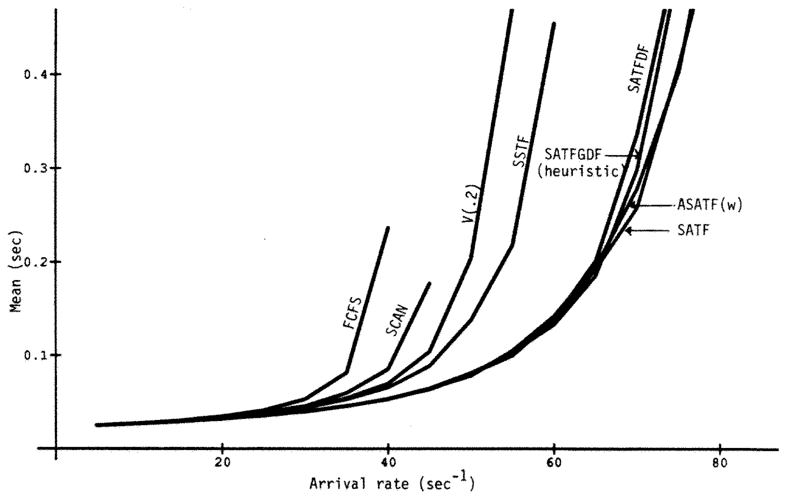


Figure 11. Service time distributions for ASATF(30) over a range of arrival rates, showing 95% confidence intervals.

6 Performance

A simulator was constructed (in C++) to model the behavior of the HP 97560 disk, including seek time, rotational latency, and transfer time. For each run, first 1000 "warmup requests" were generated, then 2000 "real" requests, and then an indefinite number of "keep warm" requests. The "real" requests were tracked and measured. The simulation was terminated when all 2000 "real" requests had completed. Each data point on the response time curves corresponds to the mean of the results from twenty such experiments.

The requests were uniformly distributed over the disk, with exponentially distributed interarrival times. Requests were a uniform 8KB in size, as this transfer size is common in UNIX file systems. The simulation was carried out for a range of mean arrival rates from very low to above the capacity of the scheduling algorithm to handle. For each arrival rate the mean service time and the 95th percentile of the service time distribution were collected. (A 200-bin histogram of response times was collected. The 95th percentile was arrived at by interpolation from this histogram.)

Simulations of the FCFS, SCAN, V(.2), SSTF, SATF, and SATFUF, SATFDF, SATFGDF with the quadratic heuristic, and ASATF(30) have been conducted. The results are shown in Figure 10.

The method of replications [Pawlikowski90] was used to estimate confidence intervals. In all cases, they were narrow. Figure 11 shows this graphically by plotting error bars representing 95% confidence intervals for the response time against arrival rate curve for the ASATF(30) algorithm.

6.1 Results

- All algorithms perform equally well at low arrival rates, when the queue lengths are very short.
- FCFS is always the least good algorithm.
- The algorithms that use seek time as their basis are distinctly better than FCFS. Ranked according to the following metrics, their performances are:[7]

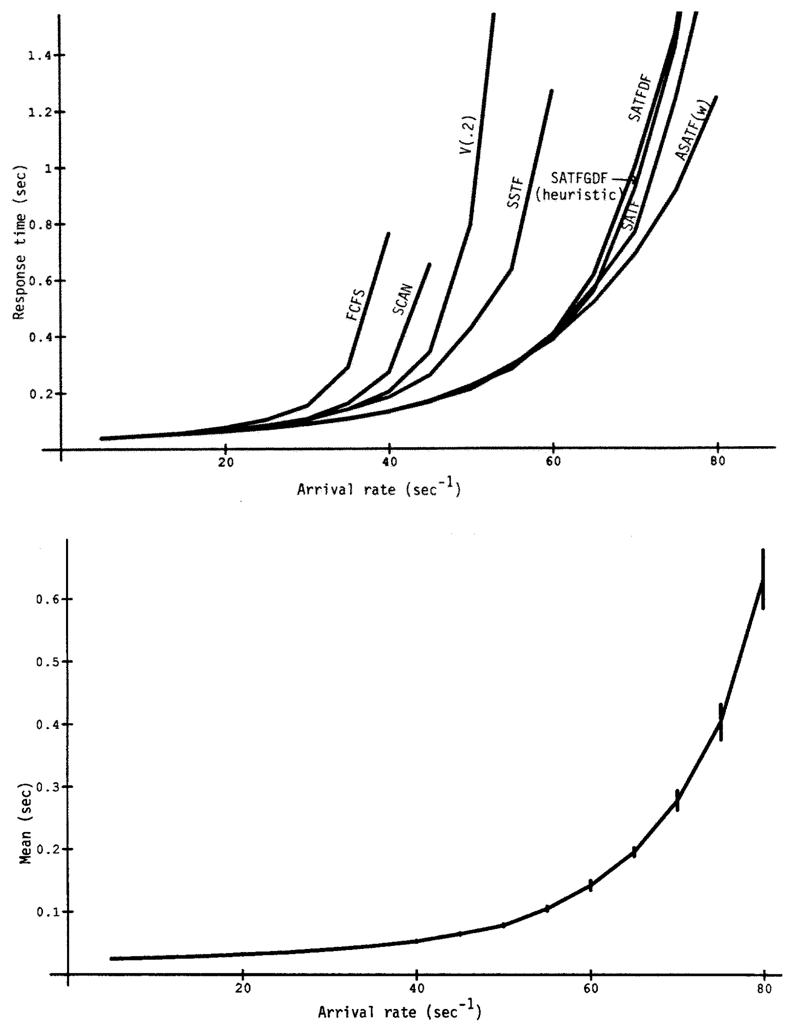


Figure 10. Mean (upper) and 95th percentile (lower) service times for different scheduling algorithms over a range of arrival rates.

- throughput only: SCAN < V(.2) < SSTF
- starvation resistance: SCAN < V(.2) < SSTF

- All the rotational-sensitive algorithms performed better than the seek-time based algorithms. In increasing order of goodness, with the metrics shown, the algorithms performed as follows:
  - throughput only: SATFDF < SATFGDF < ASATF(30) ≤ SATF
  - starvation resistance: SATFDF < SATFGDF < SATF < ASATF(30)

ASATF(30) shows the best overall performance. We chose this value of *w* by exploring the performance of the ASATF(*w*) algorithm as a function of *w* for a high-load case (a mean arrival rate of 70 requests per second). Figure 12 plots the mean, standard deviation and 95th percentile

[7]. The rankings are obtained by comparing the request rates sustainable at fixed (high) response times: 0.3s for the means, 1.0s for the 95th percentiles.

of the response time against a range of values of w. It can be seen that all these performance measures have good values over the range of $30 \leq w \leq 50$; we conservatively choose the value 30.

7 Conclusion

Several new disk scheduling algorithms based on overall access time have been proposed. The more promising ones were studied, along with several well-known algorithms, using simulation.

All the algorithms perform the same at very low arrival rates. At higher arrival rates, all the access-time algorithms match or outperform all the seek-time ones we studied.

Our best algorithm ASATF(30), can sustain 18% higher throughput than SSTF at 0.1s mean response time, 21% at 0.2s, and 25% at 0.3s. In terms of request rates for 95th percentile response times, ASATF(30) outperforms SSTF by 15% at 0.2s, 17% at 0.4s, 25% at 0.6s, and 32% at 1.0s. The request rate for ASATF(30) was never more than 2% worse than SATF for a given mean response time, and was 5% better at a 95th percentile response time of 1.0s. Finally, ASATF(30) could sustain a 44% higher throughput rate than SCAN while maintaining a 95th percentile response time under 0.4 seconds. For a 95th percentile response time of 0.5 seconds, ASATF(30) could sustain a throughput 50% higher than SCAN.

Heretofore the goal of disk scheduling algorithms has been to minimize disk arm motion. This work shows that a new goal of minimizing overall access time, taking into account rotational position, will improve maximum disk throughput by about 50% with modern disks.

The simulation results reported here were based on a load with exponential interarrival times and a uniform distribution of requests over the disk; in the next couple of months we plan to test the algorithms against traces of real system activity.

References

[Coffman72] E. G. Coffman, L. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing*, **1**(3):269–79, September 1972.

[Coffman82] E. G. Coffman, Jr and M. Hofri. On the expected performance of scanning disks. *SIAM Journal on Computing*, **11**(1):60–70, February 1982.

[Denning67] P. J. Denning. Effects of scheduling on file memory operations. *Proceedings of AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.

[Frank69] H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *Journal of the ACM*, **16**(4):602–20, October 1969.

[Garey79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H Freeman, 1979.

[Geist87] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, **5**(1):77–92, February 1987.

[Gotlieb73] C. C. Gotlieb and G. H. MacEwen. Performance of movable-head disk storage systems. *Journal of the ACM*, **20**(4):604–23, October 1973.

[Hofri80] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, **23**(11):645–53, November 1980.

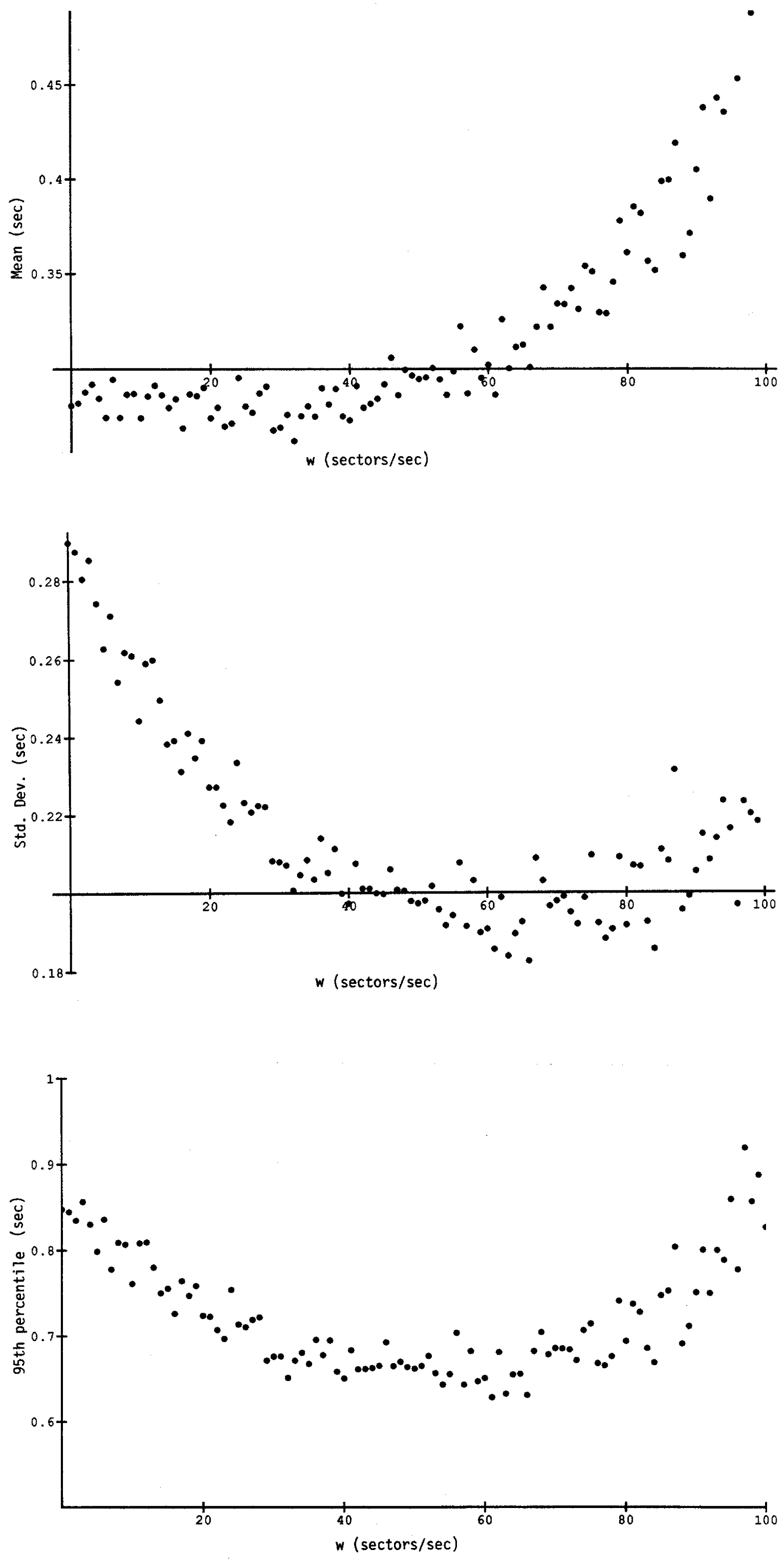


Figure 12. Mean, standard deviation, and 95th percentile of response time versus *w* for ASATF(*w*) at 70 arrivals per second.

[Leffler89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Oney75] W. Oney. Queueing analysis of the scan policy for moving-head disks. *Journal of the ACM*, **22**(3):397–412, July 1975.

[Pawlikowski90] K. Pawlikowski. Steady-state simulation of queueing processes: a survey of problems and solutions. *Computing Surveys*, **22**(2):123–70, June 1990.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of the Winter 1990 USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323.

[Wilhelm76] N. C. Wilhelm. An anomaly in disk scheduling: A comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Communications of the ACM*, **19**(1):13–17, January 1976.

[Wong83] C. K. Wong. *Algorithmic studies in mass storage systems*. Computer Science Press, 11 Taft Court, Rockville, MD 20850, 1983.