Heuristics for Disk Drive Positioning in 4.3BSD

W. Richard Stevens

Health Systems International

ABSTRACT

The throughput of a disk subsystem is critical to the performance of a computer system. The 4.3BSD Unix operating system provides a good example for looking at the heuristics that are used to optimize the positioning of the read–write heads of a disk drive. In this paper we investigate the heuristics used in the BSD *hp* disk driver. These heuristics are driven by three parameters that can be changed by the system manager. We investigate the interaction of these parameters with the BSD fast filesystem. This provides a way to see the effects of the BSD fast filesystem. Finally we present the results of monitoring three active filesystems during day-to-day use, to see what effect these heuristics can have.

1. A Scenario

A site is running 4.3BSD on a VAX with one of the popular disk configurations: an Emulex Massbus controller with Fujitsu Eagle disk drives. They upgrade to the 4.3BSD Tahoe release and after a few days notice sluggish performance. Measuring the throughput of the disks, they find it has gone from 750 Mbytes/sec to 320 Mbytes/sec. What's happened? To answer this question they might do a *diff* of the *hp* driver between the two releases, but this generates about 1,500 lines of output, as there were some fundamental changes made to all the disk drivers in the Tahoe release with the addition of disk labels. Another option is to go back through the Usenet newsgroups (assuming they've archived the appropriate newsgroups) where they would find an official Berkeley posting that describes how to tune three parameters for this disk driver that can greatly affect performance. (This posting is included as Appendix A.)

This paper is a detailed analysis of exactly what these three parameters do and how to measure them for a system. This analysis provides an interesting look at the heuristics used by some of the BSD disk drivers in positioning the read–write heads of a disk. We also find an interaction between these parameters and the BSD fast filesystem. Finally we look at some statistics collected during a day's worth of typical timesharing usage to see what effect these heuristics can have.

2. Introduction

There are two types of disk controllers typically used today:

- Controllers such as DEC's UDA50, which are termed "smart controllers." This type of controller takes requests from the operating system and automatically optimizes them. Typical optimizations include the efficient use of multiple drives, and the sorting of requests for a single drive to minimize access times.
- Controllers such as the *hp* controllers found on many DEC systems. These are termed "dumb controllers" as they require the operating system to handle any desired optimizations.

Section 8.6 of [Leffler et al. 1989] gives a description of some of the heuristics used in the *up* disk driver. This driver is for a "dumb" controller that is similar to the *hp* controller used for the tests in this paper. We are specifically interested in the heuristics used to position the read–write heads of a disk. We show that these heuristics are dependent on the processor speed, the disk characteristics (rotational speed and number of sectors per track), and the layout policies of the filesystem.

Consider the following points:

- Disk drives are slow compared to processors. Typical rotational speeds are 3600 revolutions per minute, which gives an average latency time (one-half a revolution) of 8.33 milliseconds.

- Many disk controllers support both a search command and a seek command. A *search* command positions the read–write head to a specific sector on a specific cylinder. As soon as the specified cylinder and sector are reached, the driver is notified by an interrupt. A *seek* command positions the read–write head to a particular cylinder. The driver is notified by an interrupt as soon as the specified cylinder is reached. The sector position after a seek is random.
- Some controllers support the seek command but not the search command. If a search command is supported, it is preferable to the seek, since the search notifies the driver only when the desired sector is ready to be read or written. The BSD disk drivers contain a variable, called *sc_doseek* in the *hp* driver, that can be set if the controller does not support a search command.
- Many controllers allow the driver to have a seek or a search in progress on more than one disk at the same time. This is called “overlapping seeks.” Since the positioning of the read–write heads typically accounts for most of the time spent by a disk, being able to overlap this operation on multiple disks can reduce the overall time required to access the disks. For example, assume the driver has a request for one disk that requires a seek of 5 ms followed by a read of 2 ms, and a request for a second disk that requires a seek of 10 ms followed by a read of 2 ms. If the seeks are not overlapped, a total time of 19 ms is required. Overlapping the seeks reduces this total time to 12 ms. When more than one disk is seeking or searching, the first that arrives at its desired location interrupts the processor.
- Many controllers allow the driver to obtain the current position of the read–write heads. In the *hp* controller this is called the “lookahead register.” This capability is required if the driver wants to optimize positioning on a particular cylinder.
- Most controllers only allow a single disk to be reading or writing at a time. Other disks can be seeking or searching while one disk is executing a read or write command. Also, once a read or write command is initiated, the controller cannot start a seek or search on another disk until the read or write completes. Therefore, before starting a read or a write, the heads should be positioned as close as possible to the starting sector. For example, if a read is initiated when the starting sector is still three-quarters of a revolution away on that cylinder, during that 12.5 milliseconds, plus the time required to actually read the desired sectors, the controller can’t initiate a read, write, seek, or search on another disk.
- Interrupts require processor time to handle. If every data transfer is preceded by a seek or search, then it takes two interrupts for every read or write. If the driver never performed seeks or searches, only a single

operation and a single interrupt would be required. Clearly there is a tradeoff between tying up the controller with a read or write command, versus the additional interrupt overhead associated with a seek or search.

3. BSD Block Allocation

Chapter 7 of [Leffler et al. 1989] describes the allocation of blocks to a new file that is being written. For our purposes we are interested in the following features:

- The BSD block allocation routines try to place sequential blocks on the same cylinder. This implies that the disk drivers often find the read–write heads already positioned on the correct cylinder.
- In addition to trying to place consecutive blocks on the same cylinder, the block allocation routines also try to position sequential blocks so they are in rotationally optimal locations. When reading a file sequentially, after one block has been read the driver has to process the interrupt for that operation and then set things up to read the next block. Sequential blocks should be separated by as many sectors as required so that the driver has time to start the read of the next block. This is called the ‘‘rotational delay’’ of the filesystem and we discuss it in Section 5. This parameter depends on the amount of CPU time required by the driver, so it is dependent on the processor.
- Almost every Unix system detects sequential reading of a file by a process, and automatically initiates the read of the next data block. This means that most *read* system calls, after the first one, return quickly, since the requested block is already in a disk buffer. Section 5.2 of [Bach 1986] describes this read-ahead policy. To a disk driver this means that it often finds a request for block *N* next on its queue, when it is processing the read-completion interrupt for block *N–1*.
- In the following examples we assume a block size of 8192 bytes and a fragment size of 1024 bytes. That is, all blocks of the file are 8192 bytes in length, except possibly the final block.

4. The BSD hp Disk Driver

The *hp* driver supports three parameters for each type of disk drive: *mindist*, *maxdist*, and *sdist*. These three parameters control the heuristics that are used in positioning the read–write heads. With 4.3BSD these three values are initialized in the driver source code. To change them requires modifying the driver and recompiling the kernel, or using a debugger to patch the kernel. The 4.3BSD Tahoe release moved these parameters to the file */etc/disktab*, which is used by the *disklabel* program to write these three values, along with other disk-dependent parameters, to the label on the disk drive. The Tahoe release also refers to these three parameters as *d1*, *d2*, and *d3*. The parameter *d1* corresponds to *mindist*, *d2* is *maxdist*, and *d3* is *sdist*. We’ll use the 4.3BSD names since they are more descriptive.

Any time a search operation is executed, the driver leads the search by the number of sectors specified by *sdist*. For example, if *sdist* is 5 and the desired sector is 32, then the driver issues its search command for sector 27. This parameter is required because it takes the driver some time to process the interrupt that occurs when the search is complete, before it can issue a read or write command. This parameter is affected by both the speed of the processor and the speed of the disk. If this parameter is too small, then once the interrupt from the search command is received and the read or write command issued, the desired sector has already passed by. This is called ‘‘losing a revolution’’ and the controller is tied up for a complete revolution while waiting for the desired sector to pass by again. If this parameter is too large, then the read or write command is issued too early, tying up the controller when it might be possible to be servicing another disk.

The *mindist* and *maxdist* parameters are used by the driver to determine if it should issue a search command or if it should immediately issue the read or write command. These parameters are used only when the disk is already positioned on the correct cylinder. If the disk is on the desired cylinder, the driver reads the disk's lookahead register to determine the sector at which the disk is currently positioned. If the distance from the current position to the desired position is within the ranges specified by *mindist* and *maxdist*, then the driver issues the read or write command immediately. But if the distance is less than *mindist* or greater than *maxdist*, the driver issues a search command instead.

The *mindist* parameter specifies the minimum amount of time required by the processor to set up a read or write command. If the distance to the desired sector is less than this value, a read or write should not be initiated. By the time the command is issued the desired sector will have passed by the read–write heads. Instead, the driver should issue a search command to avoid tying up the controller for an entire revolution.

The *maxdist* parameter is used to avoid starting read or write operations for sectors that are still a long distance away on the current cylinder. If the distance is greater than *maxdist* it is better to issue a search command, rather than tying up the controller with a read or write.

These three parameters should obey the following inequality:

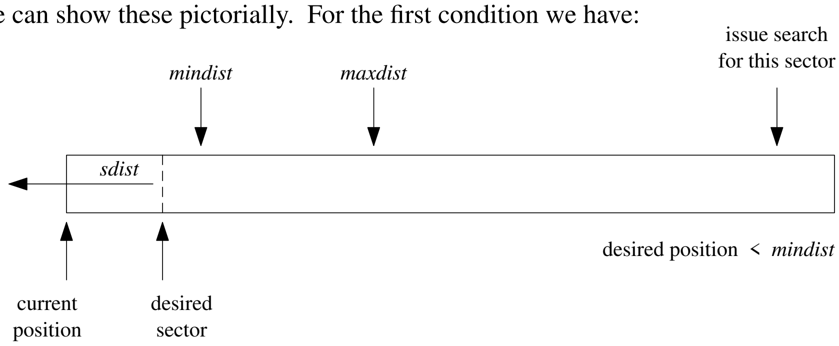
$$mindist < sdist < maxdist \tag{1}$$

Our reasoning is as follows. First, the *mindist* parameter is examined while in the disk driver. If the driver decides to issue a read or write, no additional interrupt processing is required. All it has to do is set up the appropriate device registers and start the operation. The *sdist* parameter, however, has associated with it the processing of another interrupt and then the initiation of a read or write. The difference between *sdist* and *mindist* is the time required to process this additional interrupt. To prove the second relationship of inequality (1), assume that *maxdist* is less than or equal to *sdist*. For example, assume *maxdist* is 6, *sdist* is 10, and the distance to the desired sector is 8. This distance is greater than *maxdist* so a search command is issued. But the search command is offset by *sdist*, so the search is for sector −2. (All the sector arithmetic is done modulo the number of sectors per track, since sector numbers are nonnegative. Hence, if there are 48 sectors per track, then the sector that is −2 away from the current position corresponds to the sector that is 46 sectors away.) This means the search command is issued for the sector that is behind the current position by 2 sectors—almost an entire revolution is lost. Therefore, since any search for a sector that is greater than *maxdist* sectors away is always offset by *sdist*, *maxdist* must be greater than *sdist* to avoid losing a revolution.

There are three different conditions that the driver must handle.

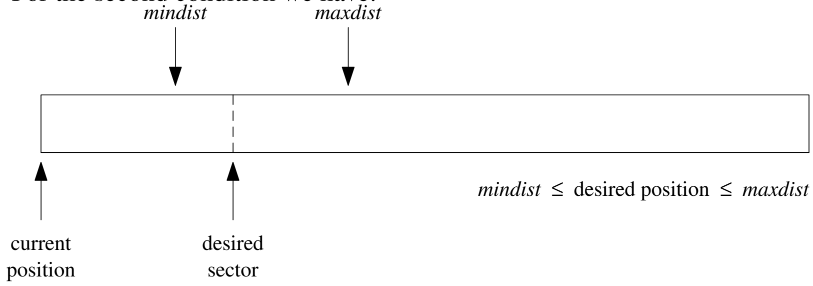
1. desired position < *mindist*.
2. *mindist* ≤ desired position ≤ *maxdist*.
3. *maxdist* < desired position.

We can show these pictorially. For the first condition we have:



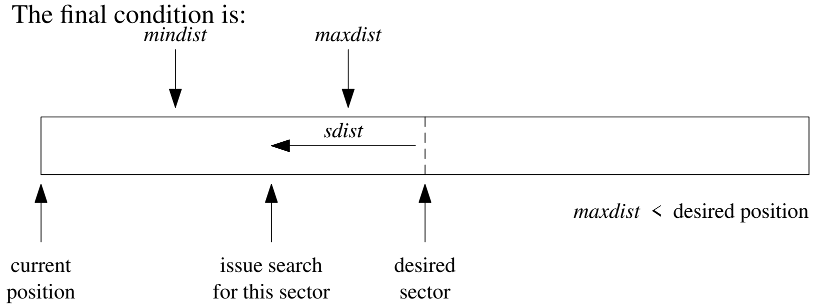
Here we show a single track of the disk as an elongated box. Realize that the left end of the box is actually joined to the right end, forming a circle. Since the distance to the desired sector is less than *mindist*, and since *sdist* is greater than *mindist* by inequality (1), the search is for a position prior to the current position.

For the second condition we have:



In this case the driver issues the read or write command immediately.

The final condition is:



The implementation of the *hp* disk driver in the original 4.3BSD distribution contains some errors that are worth investigating. First, the test against *mindist* and *maxdist* is backwards:

```
if (on desired cylinder) {
        /* calculate dist = distance to desired sector */
        if (dist > maxdist || dist < mindist)
                /* start read or write */
        else
                /* do a search */
}
```

The condition should be

```
        if (dist <= maxdist && dist >= mindist)
                /* start read or write */
        else
                /* do a search */
```

This is fixed in the 4.3BSD Tahoe release. Another bug that is fixed in the Tahoe release is that the original 4.3BSD release incorrectly set the *sc_doseek* true for some controllers (all Emulex controllers). This prevented any search commands from being executed for these controllers. Instead, the driver executed only seek commands and then issued the read or write anywhere on the desired cylinder.

The final bug, which is still in the Tahoe release, is that the distributed values for the three parameters, *mindist*, *maxdist*, and *sdist* are incorrect. The entries for every disk type have a *maxdist* value that is less than *sdist*. As shown above, this can cause a revolution to be lost whenever the actual distance is between the two values. These incorrect values can drop the performance of certain disks by a factor of three when switching from the original 4.3BSD release to the Tahoe release. Indeed, this performance loss was noticed and a Usenet message posted by Berkeley on how to adjust these three values. This posting is included as Appendix A. The techniques given in this message are not optimal, although the values derived are better than the distributed values.

5. Measuring the sdist Parameter

Tests were run on two different systems, a VAX 11-785 and a VAX 8650. On the 785 was an Emulex SC780 disk controller with Fujitsu Eagle (M2351) disk drives. The 8650 had an Emulex SC7003 disk controller with CDC 9720-850 disk drives. Both Emulex controllers are Massbus controllers. Some relevant parameters for these two disks are

	Fujitsu Eagle	CDC 9720-850
revolutions per minute	3961	3600
sectors per track	48	68
bytes per sector	512	512
microseconds per sector	316	245
track-to-track seek, milliseconds	5	5

Table 1: Disk drive parameters for disks used in tests

The microseconds per sector value is calculated from the first two values above it in the table.

The first thing to measure is the *sdist* value. This value shows the amount of time required by the processor to handle an interrupt from the controller and issue a read or write. We expect that as we continually decrease this value we will hit a point where there isn't enough time for the driver to respond to an interrupt and start the read, causing a revolution to be missed. Indeed, Figure 1 shows this.

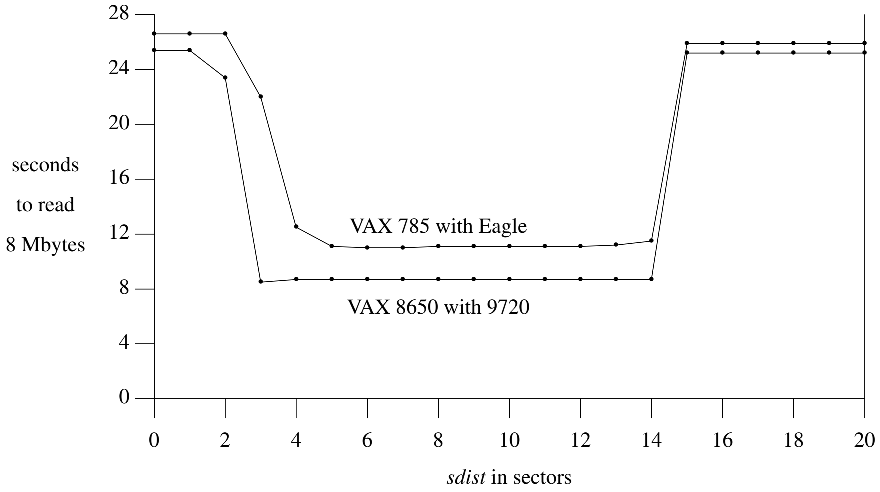


Figure 1: Effect of *sdist* on a sequential read

The timing tests were done by writing an 8 megabyte file and then reading the file using different values for *sdist*. The reading and writing were done with 1024 reads or writes of 8 Kbytes at a time. Multiple tests were done for each value, to make certain there were no anomalies with any specific test. The systems were in single-user mode while the timing tests were conducted. The important value to measure is the total elapsed time—the wall clock time. Also, to accurately measure the *sdist* effect, we want every read or write to be preceded by a search command, even if the disk is already positioned on the correct cylinder. To do this we set *mindist* to 1 and *maxdist* to 0. This causes the driver's test that we showed earlier to always be false, causing a search command to be issued before every read or write. Then we continually decreased the value of *sdist* to see how far ahead of the desired cylinder we had to be interrupted. For the 785 the minimum value for *sdist* is 5 sectors and for the 8650 it is 3 sectors. In actual use, however, one should be conservative with these values since there is typically more system activity than in the single-user conditions used for these tests. For example, we actually use *sdist* values of 8 for the 785 and 5 for the 8650 to allow for this variability.

Note, however, that the graph not only shows an increase in the time required to read the file when the *sdist* value is too small, but it also shows an increase at larger values. This is an interaction with the BSD fast filesystem that is instructive to look into.

First, as mentioned earlier, the BSD filesystem tries to place consecutive blocks of a file separated by a certain spacing (in sectors). This is termed the "rotational delay." With 4.3BSD it defaults to 4 milliseconds when the filesystem is created and it can be changed at a later time using the *tunefs* command. With the Tahoe release the rotational delay is an optional argument to the *newfs* command and it can also be changed with the *tunefs* command. This rotational delay is to ensure that the system has enough time after receiving the read-complete interrupt to process the interrupt and start the read of the next block of the file. This optimization of the next sector location is done when the file is being written. The actual algorithm to calculate this distance is in the routine *blkpref*, and is based on the following calculation:

$$dist = \left\lfloor \frac{rotational\ delay\ in\ milliseconds \times \dfrac{revolutions}{second} \times \dfrac{sectors}{revolution}}{\dfrac{sectors}{fragment} \times \dfrac{1000\ milliseconds}{second}} \right\rfloor \qquad (2)$$

The value of this equation is the number of disk fragments (e.g., 1024-byte fragments, if the filesystem uses 8 Kbyte blocks with 1 Kbyte fragments) that should separate consecutive data blocks for a given file. The integer value *dist* is then rounded up to the next multiple of the block size (e.g., 8). For both the Eagle and 9720 disks, this calculation generates a spacing of 8 fragments (16 sectors), if the rotational delay is 4 milliseconds. Since the value is always rounded up to the next multiple of the block size, a smaller value of the delay in milliseconds still generates the minimum spacing of 16 sectors. It is possible, however, to generate a larger spacing by specifying a delay greater than 4 milliseconds.

Now we can see why Figure 1 has a large increase around an *sdist* value of 14 sectors for both disks. When the driver processes the interrupt for the completion of the read, the next block to read is usually 16 sectors away. By the time it issues its search command, if the value of *sdist* is greater than 14, it generates a search command for a sector prior to its current position, so a complete revolution is lost. The spacing of consecutive data blocks around the cylinder by the BSD fast filesystem is what generates the increase in Figure 1 around the *sdist* value of 14 sectors.

We now compute an estimated value for the maximum reading rate that we can expect from these two disks. To read an 8 Mbyte file in 8 Kbyte chunks, the following two steps are executed 1024 times:

1. Read 8192 bytes (one block, 16 sectors).
2. Let the disk move another block (16 sectors) until the next data block is reached.

For the 9720 disk this gives a calculated minimum time of 8.03 seconds, and for the Eagle disk the value is 10.35 seconds. These values are close to the measured values of 8.7 and 11.1. Note that we have ignored the track-to-track seek times that must also occur, since an 8 Mbyte file requires at least 18 cylinders on both disks. We have also ignored the opening of the file, which also requires a small amount of time.

If we miss a revolution, however, the steps become:

1. Read 8192 bytes (one block, 16 sectors).
2. Let the disk move another complete revolution plus one block (16 sectors) to reach the next data block.

Adding up the times for these two operations gives calculated values of 25.1 seconds for the 9720 and 25.9 seconds for the Eagle. The measured values were 25.2 and 25.9, respectively.

We can now understand why the faulty 4.3BSD routines generated optimal results, despite their bugs. Since the driver's *sc_doseek* flag was always true for an Emulex controller, whenever the disk was already positioned on the correct cylinder, a read was started, regardless how far away (or how close) the desired sector was. Since the next block was typically 16 sectors away, the read was started and a revolution was not lost.

Fixing the coding bugs, however, without setting the three parameters to reasonable values, provides a performance drop of a factor of three. For example, setting the driver's *sc_doseek* flag false (which is correct) and correcting the test involving *mindist* and *maxdist* generates the following scenario for the Eagle. The default value of *sdist* is 15 and the default value of *maxdist* is 8. When the driver finishes reading one data block the next one is about 14 sectors away. This is greater than 8 so the driver issues a search, but the search is offset by 15 sectors (e.g., one sector before the current position) so a revolution is lost.

Performing the timing tests described in this section not only shows the maximum value for *sdist* that should be used, it also shows the spacing being used to separate sequential blocks of a file. To see what effect the "rotational delay" term in equation (2) has on the performance, we changed a filesystem on a CDC 9720-850 disk from a 4 ms delay to a 5 ms delay, using the *tunefs* command. Equation (2) now gives a *dist* value of 10 fragments, which is then rounded up to 16. This corresponds to 32 sectors. Rewriting the 8 Mbyte file and then reading the file using different values of *sdist* gives the following results.

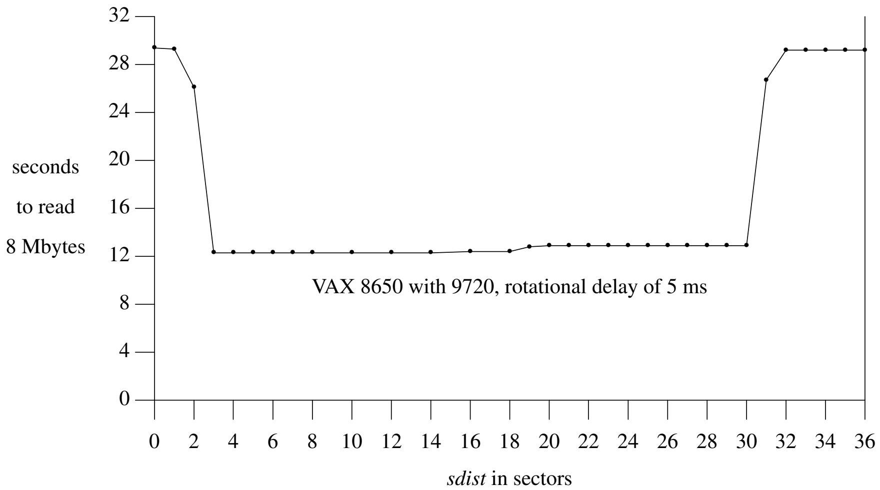


Figure 2: Effect of *sdist* and rotational delay on a sequential read

This effect of the rotational delay is increased for disks with large numbers of sectors per track since the number of sectors per track is in the numerator of equation (2).

The drop in Figure 2 between a distance of 1 sector and a distance of 3 sectors is identical to the drop in Figure 1 for this processor. We expect this, since the amount of processor time required to handle the search-complete interrupt and start the next operation is dependent only on the processor and the sequence of instructions that must be executed in the device driver. This means that if we measure the *sdist* value for a particular processor, device driver, and disk drive, we can calculate the value for any other disk drive that uses the same processor and device driver. For example, both Figures 1 and 2 show that the *hp* driver on a VAX 8650 requires about 735 microseconds to handle the search-complete interrupt and start another read. This is because both Figures 1 and 2 flatten out at a distance of 3 sectors, and the time per sector is 245 microseconds from Table 1. Given the first four numbers in Table 1 for any other disk drive on this system, one can calculate the number of sectors corresponding to 735 microseconds.

6. Additional Driver Measurements

Another empirical test that we use to verify what is happening in the *sdist* measurements is to modify the disk driver to keep a counter of what the actual sector distances are. We have modified the *hp* driver to keep the following counts, for each disk:

1. For every disk operation, how many times the disk is already positioned on the correct cylinder.
2. If the disk is already on the correct cylinder, we maintain a set of counters for each sector distance.

The first counter indicates how often the *mindist* and *maxdist* values are actually used. The second set of counters allows us to build a histogram of the sector distances.

These counters were enabled on the VAX 8650 while doing the sequential read of the 8 Mbyte file from the previous section. The results of the first counter show that 98% of the time the disk is already positioned on the correct cylinder and only 2% of the time is a search command to a new cylinder required. We estimated earlier that it required a minimum of 18 cylinders on either the Eagle disk or the 9720 disk to store an 8 Mbyte file. The actual counter showed that 20 searches had to be executed to move to another cylinder.

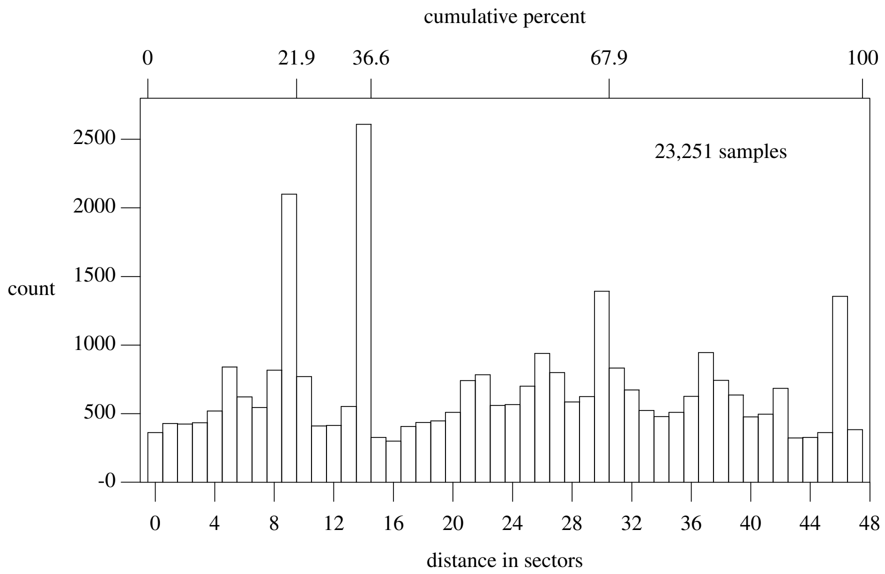
The second set of counters shows that 98% of the time the desired sector is 14 sectors away from the current position. The desired sector is 15 sectors away 1% of the time, and the remaining 1% of the time the distance is other than 14 or 15. This correlates with the values in Figure 1 and our discussion in the previous section.

7. Selection of mindist and maxdist

The tests in Sections 6 and 7, however, are for a special case: the sequential reading of a large file with no other system activity. Section 6 showed that the value of *mindist* is insignificant 99% of the time. The value of *maxdist* only determines whether the driver starts the read 14 sectors away from the desired location, or whether it relinquishes the controller. To provide fast reading, *maxdist* should be greater than or equal to 15, allowing the disk that just completed the read to start another read. Setting *maxdist* less than 14 allows another disk to either issue a search or start a read or write. If another disk is ready to start a read or write, it will probably tie up the controller so that the next block will be missed.

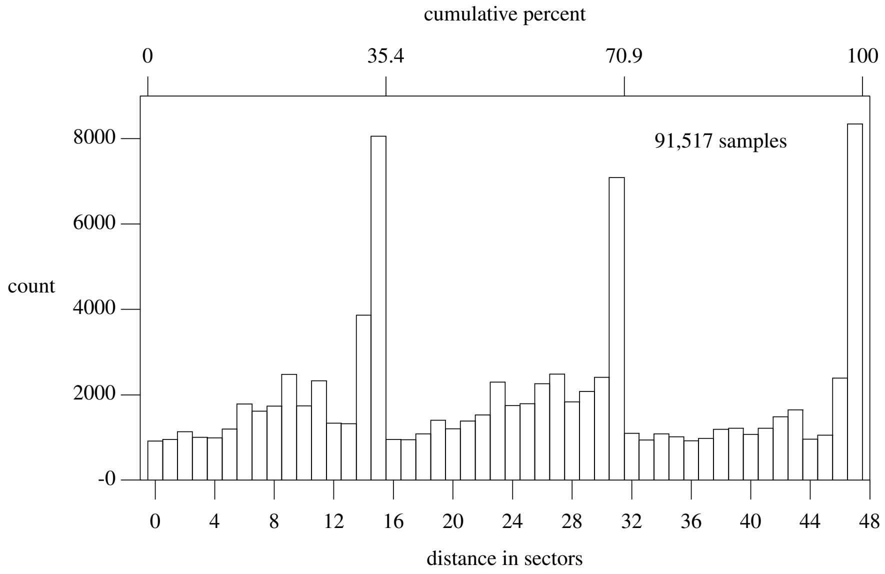
What should drive the selection of *mindist* and *maxdist* is the distribution of the sector distances for a typical load. This distribution was measured on both the VAX-11/785 and the VAX 8650 by enabling the counters described in the previous section for 15 hours during a typical work day. The 785 usually had around 55 users and the most frequent commands were the shell, *ls*, *vi*, *more*, and *troff*. The 8650 was used by 30 programmers for program development and testing. The most frequent commands were *ls*, *vi*, *make*, and the multiple steps of the C compiler. There were four Eagle disks on the 785 and two 9720 disks on the 8650. Since there is no ‘‘typical’’ disk usage in a multiuser environment, three different filesystems were monitored.

The first filesystem we show is an Eagle disk that is the primary user filesystem on the 785. This is where the home directories for all users are located and where most users keep the files they edit. User filesystems are characterized by many small files. The first counters show that 30.4% of the time the disk is already on the correct cylinder, while 69.6% of the time it is not. The following figure shows the histogram of the sector distances when the disk is already positioned on the correct cylinder.

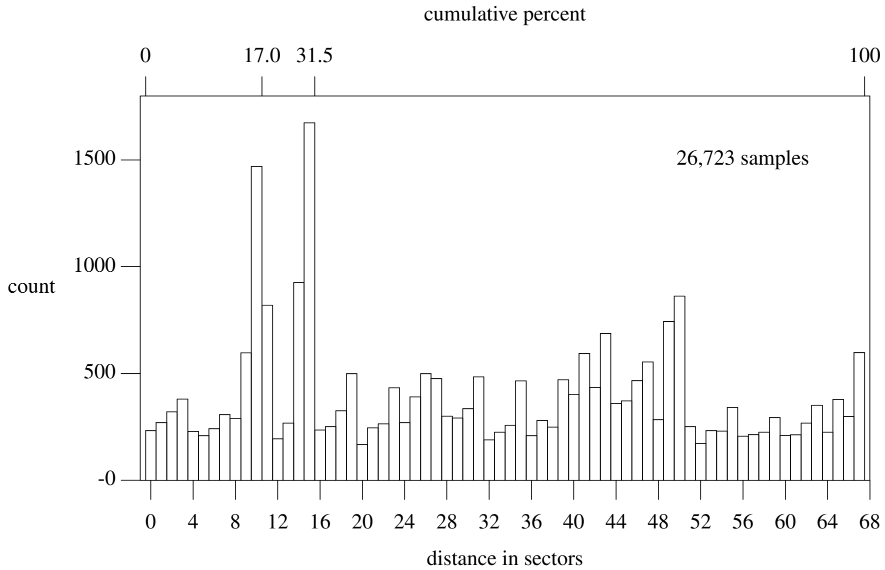


The top axis shows the cumulative percent, up to and including that distance in sectors. For example, 21.9% of all distances are between 0 and 9 (inclusive); 36.6% of all distances are between 0 and 14 (inclusive). The peak around 14 is probably caused by the reading of blocks sequentially from files. The slight increase at 30 is for reads that are separated by two 8 Kbyte filesystem blocks. The increase at 46 is for blocks that are a complete revolution away. This happens when a read or write completes, and the next request on the queue is for the next contiguous sector, somewhere on the current cylinder. But by the time the driver has processed the interrupt for the operation that just completed, the desired sector has just passed by. This is an example where the driver should not issue the read or write, as it would tie up the controller for about 16 milliseconds before the data transfer can even begin. In this amount of time another disk can, for example, do a track-to-track seek, skip over 8 sectors, and then read 16 sectors.

The next figure is for the root filesystem on the 785. Additionally, this disk contains three other filesystem partitions: */usr*, */usr/local*, and a swap partition. This is a heavily used disk drive. Most executable files are read from its different partitions, */bin*, */usr/bin*, */usr/ucb*, and */usr/local/bin*. The root partition also contains the */etc* directory, which contains frequently accessed files. The swap partition is not used heavily, as the system has enough memory (24 Mbytes) to prevent most paging and swapping. Only 26.0% of the requests found the disk already positioned on the correct cylinder. This low value is probably because there are four different partitions in use on the disk. The distribution of these requests for the different sector distances is given below.



The final histogram we show is for a 9720 disk that only contains the */tmp* filesystem for the 8650. This filesystem is characterized by the writing and then reading of temporary files. For 48.5% of the requests the disk was already positioned on the correct cylinder. For these cases the histogram of the sector distances is given below.



Having some actual histograms of the sector distances, we can now make reasonable choices for the *mindist* and *maxdist* values. We choose a value of *mindist* of 2 for the 8650 and 4 for the 785. We want to make certain that the driver has enough time to issue the next read or write command, before that sector has gone by. We choose a larger value on the slower system (the 785), since the same sequence of instructions has to be executed on both systems.

For both the 785 and the 8650 we choose a *maxdist* value of 16. From the three histograms above, this guarantees that we immediately read or write the blocks that cause the histograms to peak between distances of 9 and 15. In all three histograms, distances of less than 16 sectors account for about one-third of the cases when the disk is already positioned on the correct cylinder. To account for a higher percentage of the cases, *maxdist* must be at least 32. But for the Eagle this ties up the controller for up to 10 milliseconds, which is time that can be spent servicing another disk.

8. Summary

There are two factors that a system manager has control over that affect the throughput of a disk controller such as the *hp* controller used for the tests described in this paper.

1. The values of the three parameters *mindist*, *maxdist*, and *sdist*.
2. The spacing that results from the rotational delay between sequential blocks of a file.

We can provide the following recommendations for users of either the 4.3BSD or the 4.3BSD Tahoe systems:

1. With the 4.3BSD release (not the Tahoe release) one should fix the driver bugs having to do with the incorrect setting of the *sc_doseek* flag and the “if” statement that compares the distance to the desired sector against

mindist and *maxdist*.

2. Measure the *sdist* parameter for the processor being used. This is critical to good performance. This measurement also verifies the rotational delay between sequential blocks in a file. If the rotational delay is too large, use the *tunefs* command to reduce it. The spacing shown by these measurements should correspond with the value calculated by equation (2).

3. Choose reasonable values for *mindist* and *maxdist* as described in Section 7. Be sure to maintain the relationships of inequality (1).

Appendix A:
The Original Tuning Method

The following is the Usenet posting from September 1988 on how to tune the *d1*, *d2*, and *d3* parameters.

Subject: How to tune d1, d2, d3 params in /etc/disktab
Index: etc/disktab 4.3BSD-tahoe

For those of you that are interested in maximizing the performance of your disks, here is an empirical procedure developed by Tom Ferrin (tef@cgl.ucsf.edu) to tune the drive dependent parameters for disk controllers that have search capabilities. CSRG would be interested in getting experimental results for other drives in /etc/disktab and also in hearing of refinements to this procedure.

Here is the procedure to empirically determine the correct drive-dependent parameters for "HP" type disks. This technique improved both read and write times for an NEC2362 drive by more than 2x! Since the "doubleeagle" params were used as a starting point, throughput to that drive could probably be improved considerably as well since the physical characteristics of the two drives are very similar. The throughput is now nearly 1MB/sec on writes (995 KB/s to be exact) and 842 KB/s on reads.

Here is the method:

0. format the drive and build the badblock table.
1. set d3 (sdist) to something large, say 20.
2. set d1 (mindist) to 0.
3. set d2 (maxdist) to #sectors/track - 1 (63 for the NEC).
4. write out these parameters using disklabel.
5. newfs a filesystem and mount it.

This essentially disables the "search" capability of the driver so than whenever the drive is positioned on the proper cylinder the driver immediately issues a data xfer command. This will minimize the data transfer time at the expense of tying up the controller for a longer period (possibly for a complete revolution [~17ms]). Now use Mike's "write_8192" program (was in ~karels/tests on the distribution tape) to write a 8MB file out to the filesystem.

Optimize maxdist as follows:

6. use the read_8192 program to determine the real time required to read the data file. (do 4 - 6 runs to make sure there are no abnormalities on a particular run).
7. reduce maxdist (d2) by 10 in disktab, write it out with disklabel, and do step #6 again.

The real time required to read the file should remain relatively constant until maxdist gets too small, then the time will suddenly jump up as you begin missing revs on the xfer. Decrease the step size from 10 to 5 and iterate again, finally choosing a value a a little larger than the "cut off" value. (It is much better to be conservative.)

Optimize mindist by increasing it upwards from 0 using a step size of 1 or 2, again using read_8192 as the benchmark. You will see that time required to read the file will initially be small, but then will start creeping upwards as the driver begins issuing "search" commands when it could be starting a read directly. Back off 1 or 2 sectors from the point where the file read time first starting increasing. (Being conservative is less critical here.)

Mindist and maxdist determine the window (measured in sectors) from which the driver determines to whether to begin a data xfer immediately or to do a "search" command first (possibly freeing the controller for a data xfer on another drive). Values of 2 and 15 worked best for mindist and maxdist on a NEC D2362, using an Emulex SC7003 controller on a VAX8650.

Lastly optimize sdist in the same way you optimized maxdist. A value of 5 is reasonable (far different than the 15 originally found in disktab). Again it is better to error on the conservative side, since being too liberal causes you to miss a rev, while being too conservative just ties up the controller longer while the data xfer is queued up and the controller is waiting for the heads to get over the proper sector.

A final note: bus I/O architecture and CPU speed play a factor in all of this, since how fast it takes the processor to field a device interrupt and how fast the code in the driver executes affect timing. Ideally someone should use the same drive and controller setup on a slow CPU (eg VAX750) and see how different the numbers are.

References

M. J. Bach, *The Design of the UNIX Operating System,* Prentice Hall (1986).

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