DISK STRIPING

Kenneth Salem
Hector Garcia-Molina

EECS-TR-332-84

(Revised)

December 1984

DISK STRIPING

Kenneth Salem
Hector Garcia-Molina

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

ABSTRACT

Just like parallel processing elements can substantially speed up computationally intensive tasks, concurrent transfer of data in and out of memory can speed up data intensive tasks. In this paper we study one general purpose facility for achieving parallel data motion: disk striping. A group of disks is striped if each data block is multiplexed across all the disks. Since each subblock is in a different device, input and output can proceed in parallel. With the help of an analytical model, we investigate the advantages and limitations of striping in four representative applications: simple block fetch, b-tree searching, file processing and merge sorting. We also explore four possible enhancements to striping: immediate reading, ordered blocks, matched disks, and subblock screening.

index terms: disks, input/output, striping, searching, sorting, parallelism, performance

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.

DISK STRIPING

Kenneth Salem
Hector Garcia-Molina

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

1. Introduction

A group of disk units is *striped* if each data block is stored, not on one of the disks, but across all of the disks. That is, if there are n disks, each block is split into n subblocks and a subblock is placed on each disk.

The goal of striping is to improve IO bandwidth. Ideally, if a block is spread across n disks, then it can be read (or written) in parallel, cutting access time by a factor of $1/n$. Unfortunately, this beneficial effect may be counterbalanced by several factors, including higher CPU overhead and larger rotational and arm movement delays. In this paper we study these tradeoffs in detail, and characterize the applications and circumstances that make striping advantageous. We also study variations and enhancements to the basic striping idea, and attempt to determine their usefulness.

In the current quest for very high performance computers, the idea of striping has been causing considerable interest. Both the Cray Operating System[†] and Convex Unix now provide support for striped disks [MASO84][CONV86]. Reportedly, it improves performance on some applications significantly [WALL86].

Striping, however, is by no means a new idea. Although we have been unable to find references in the published literature, it appears that some early versions of Unix gave users the option of striping their files [HONE84]. This option disappeared from the newer version of Unix, it seems, because it was not heavily used. Striping can be implemented in a device as well as in an operating system. On some parallel-readout disk models, striping has been done for many years within the disk drive itself. These units have multiple read/write heads mounted on a single arm, and parallel data transfers to and from the heads are possible. However, as disk densities increase, it becomes more difficult to align the different heads

† As far as we know, this system was the first to use the term "striping" that we have adopted here

simultaneously during I/O. This makes parallel-readout disks much more expensive and limited to relatively small number of striping channels. The alternative, striping from separate disk drives, is becoming more effective, and is thus the only one we consider in this paper.

Striping has also been proposed for multi-processors and database machines. For example, the PASM multi-processor [SIEG79] includes the specification of multiple disk drives for parallel loading of the primary memories. This scheme has the added advantage that each disk has an independent path to its corresponding memory module, avoiding the inherent contention of a system-wide memory bus. It is also interesting to note that Boral and DeWitt [BORA83] point to striping as one important technique that may improve performance in database machines.

With rapidly increasing memory densities, there has recently been interest in memory resident database systems [GRAY83, DEWI84, GARC84]. Here again, striping has been suggested as a means to improve bandwidth to disks. For example, striping can reduce the time it takes to load the database into memory after a crash, or it can reduce delays in writing the log.

Finally, we suspect that some type of striping is used in many high performance commercial applications, for instance, in sorting packages. However, in this paper we are interested in viewing striping as a *general purpose facility* that can be used in a transparent way. In this facility, applications would simply define the degree of striping (i.e., n) and would read and write full blocks. The manipulation of the subblocks would be invisible.

In spite of the wide range of applications of striping, very little is known about its performance and its usefulness as a system facility. Is it only useful for loading up databases and initializing large memories, or can it be used for other important computations? What variations of the basic striping idea work best? What are the limits in the performance improvements possible through the use of striped disks? Is striping simply a "hack", or should it be incorporated into operating systems? In order to answer these questions, we embarked on the present study.

Most performance evaluations make numerous simplifying assumptions, but since we are dealing here with non-linear mechanical devices (i.e., the disks) we make more than our share. These assumptions will be described in detail in the following sections. In general terms, we believe they are justified because we are only looking for general *trends* in striping performance. The present study must be viewed as an exploratory one, to be followed up by an experimental one that actually verifies the the promising ideas that we uncover here.

There are a number of ways to approach a study of disk striping. We have used a model in which the striped disks are part of a batch-type, back-end system. High-performance computers are frequently shared among a set of "client" machines in this fashion. Other models are certainly possible and important. One example is the model used in [KIM84] which considers synchronized, striped disks in a multiprogrammed system. Models such as ours have the virtue of simplicity; they allow us to understand striping by viewing it in a less complicated environment. In addition, back-end batch systems are in common use and can benefit tremendously from better IO performance, so our model is an important one that deserves consideration.

The remainder of the paper is divided into five sections. In Section 2 we present our model for disks, including the CPU overhead, rotational and latency delays. We first model a single subblock transfer, and then the transfer of n subblocks. Since all n subblock transfers must complete before a block transfer terminates, the expected rotational and seek delays for the full block will be larger. This, as we will see, will be one of the most important factors that limits the efficiency of striping.

In Section 3 we describe four striping enhancements that may improve performance. In Section 4 we discuss four different applications: simple block fetch, b-tree searching, file processing, and merge sorting. For each application, we compare the completion time of a task that exploits striping to one that does not, varying the enhancements that are used. Examining these specific tasks introduces additional parameters and assumptions, but in return we can evaluate striping in a set of realistic and representative environments. Section 5 is a discussion of the reliability aspects of striping. Since a detailed comparison of the reliabilities of striped and unstriped disks could fill another whole paper, we have merely described the important issues and some strategies for boosting availability. Finally, in Section 6 we make some concluding remarks.

2. The Model of Disk Response Time

A simple mathematical model is used to determine the disk IO response time in a multiple-disk system. Response time includes all of the elapsed time between the initialization of the IO request at the CPU and the storage of the data at its destination (disk or memory). It consists of instruction execution time and the mechanical and electronic delays in the disk drive and channel. For the purposes of the model, response time is split into two components, CPU time and disk time. Each of these is discussed in one of the following sections.

2.1. CPU time

CPU time is spent initializing requests when it becomes necessary to transfer data to or from the disks. This includes any calculations and lookups necessary to determine the location of the desired data, management of in-core buffers, and time to make this information available to each disk in the system (i.e. place it on a bus). CPU time also includes time to handle "IO completed" interrupts from the various disks after each transfer.

To reflect the CPU time, the model incorporates an initialization time $t_{ioc} + nt_i$. The first term represents that part of the CPU time that does not vary with the number of disks in the striped group. The second term represents processing which must be done for *each* disk in the n-disk system. For example, the locations of target disk sectors must be determined for each of the n disks in the striped group for each IO operation.

2.2. Disk time

Disk time can in turn be broken into three major components. *Seek time* is time required to position the read/write heads of the disk over the track containing the data. *Rotational latency* refers to the wait for the spinning platter to carry the desired sector or sectors of the disk track under the read/write heads. *Transfer time* is the time taken for the actual transfer of data from the disk to a memory buffer (or vice versa). Our models for these delays are discussed more fully in the next two sections.

2.2.1. Seek and Rotation Delays

In the general case, we assume that pieces of the same block need not be stored at the same location on each of the disks in a striped system. As a result, the seek and rotational delays involved in the same transfer will be different for each disk. In the model, the delays at each disk are random variables. Individual rotational delays are assumed to be uniformly distributed on $[0,t_{ml}]$ where $t_{ml}=\omega_{disk}^{-1}$, the maximum rotational latency of the disk. The seek time distribution is slightly more complicated. We assume that the next subblock to be retrieved from a disk is equally likely to be found on any cylinder in the task sub-partition (described in section 2.2.2) of that disk. This is a slightly pessimistic assumption in that it does not attribute any special locality to the disk reference pattern. The resulting distribution is shown in Figure 1.1. A derivation of this function can be found in Appendix C. Other seek time distributions are certainly conceivable. We also experimented with a uniform distribution on $[0,t_{ms}]$ but the results were not significantly different.

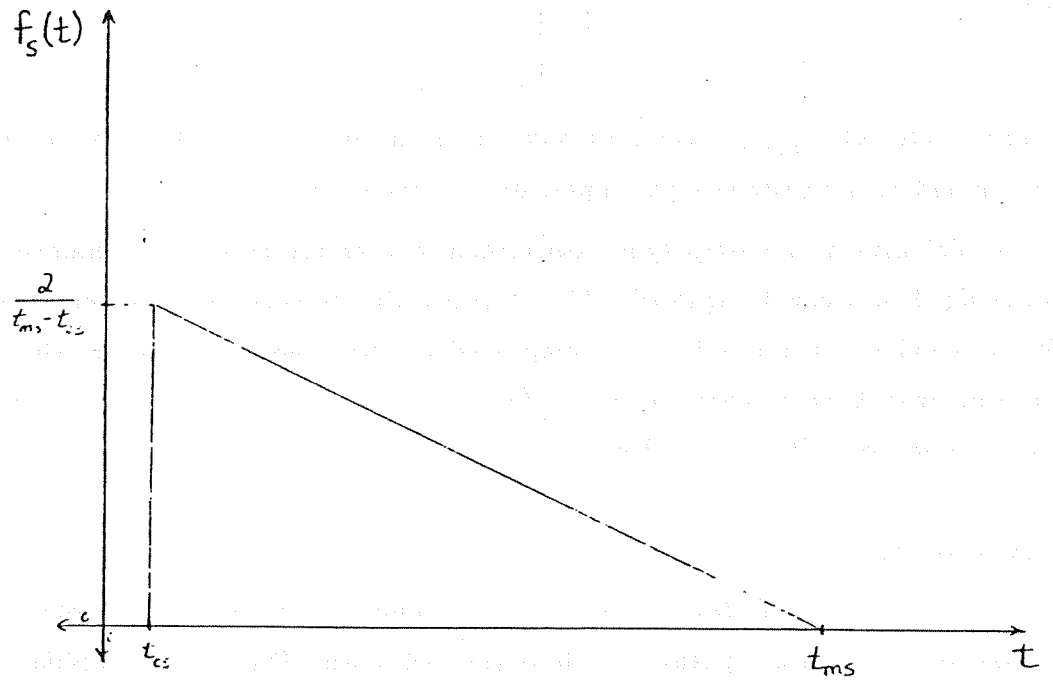


Figure 1.1 - Distribution of Seek Time for a Single Disk

Once the seek and rotational distributions, $f_s^1(t)$ and $f_r^1(t)$ are known, the distribution of their sum can be determined (for a single disk) by convolving:

$$f_{(s+r)}^1(t) = \int_{-\infty}^{\infty} f_s^1(t-x) f_r^1(x) dx$$

The seek and rotational delay for the n-disk system will be the longest of the delays of the component disks. Letting $f_{(s+r)_i}^1(t)$ be the distribution for the ith disk and assuming that the disks are independent, the distribution $f_{(s+r)}^n(t)$ of the delay for the n-disk system can be determined. Specifically, if we let $F_{(s+r)_i}^1(t)$ represent the cumulative distribution function corresponding to $f_{(s+r)_i}^1(t)$ and $F_{(s+r)}^n(t)$ be the cumulative distribution function corresponding to $f_{(s+r)}^n(t)$, then

$$F_{(s+r)}^n(t) = \prod_{i=1}^{n} F_{(s+r)_i}^1(t)$$

since we must wait for *all* of the n disks to respond. If all of the distributions $f_{(s+r)_i}^1(t)$ are the same, we can write

$$f_{(s+r)}^n(t) = \frac{d}{dt} F_{(s+r)}^n(t) = \frac{d}{dt} \left[\left[F_{(s+r)}^1(t) \right]^n \right] = n \left(F_{(s+r)}^1(t) \right)^{n-1} f_{(s+r)}^1(t)$$

The expected value of $f^n_{(s+r)}(t)$, the seek and rotational delay distribution for the n-disk system, is computed numerically using a trapezoidal integration scheme.

The model makes one simplifying assumption concerning seek times, namely that any head-switching delays can be ignored. This is reasonable because head-switching and movement to a new cylinder can usually be accomplished simultaneously. Situations where the head switching time may have an effect, such as when no seek is necessary at all, occur infrequently enough under our model that their effect would be small.

2.2.2. Transfer time

Transferring data from disks to memory is a complicated operation involving disk and memory controllers and data paths as well as the disk itself. Our model simplifies the measurement of transfer time by assuming that the slowest link in this process is the rotational speed of the disk. In other words, it is assumed that the rest of the system hardware and busses are fast enough to transfer data as quickly as it comes off of the disk. If the disks have separate data paths to primary memory, this is a reasonable assumption. Many current disk drives list peak transfer rates on the order of 2 Mbytes/second, well within the grasp of a system in which bus contention is not a problem. Even without separate data paths, the assumption will be valid for small n. However, when the number of devices sharing a data path becomes too large, bandwidth limitations lead to a more complicated expression of transfer time than the one we present. Our data and conclusions should be considered with this assumption in mind.

Transfer time depends on how data is stored on the disks, and the application models discussed later use some common assumptions regarding storage and transfer. The space available for storing data on disks can be broken up into cylinders, tracks and sectors. Our models make little or no distinction between tracks on the same cylinder since head-switching times are ignored. We also assume that the *minimum* amount of data that can be transferred from a single disk at one time is one sector, and that the *maximum* transfer is one complete cylinder. This latter requirement eliminates the problem of head seeks during data transfer. Furthermore, only whole sectors are transferred.

Each of the applications (except Block Fetch) would involve the use of a data set. The data may or may not be broken up into records, depending on the application. In either case, the total amount of disk space which would be occupied by the data set is called the task *partition*. When the data are spread across several disks, the space occupied on each disk is called the task *sub-partition* of that disk. The size of a sub-partition is determined by packing data

into sectors as tightly as is possible, with the provision that records are never split across sectors unless the record size is greater than the sector size. For the Block Fetch task, each sub-partition is simply taken to be $1/n$ of a disk's surface in an n-disk system.

Since we are studying a back-end, single-user computer, we assume that a running task is the only task making IO requests to the disk. In other words, once a disk's heads are positioned in its sub-partition they will not moved out by an IO request from any other task. Also, each sub-partition is assumed to be of single extent. This means that the data would occupy a set of physically contiguous cylinders on each disk.

Other important quantities involved in data transfer are m, the memory buffer size, and b, the block size. The block size is the the *total* amount of data sent or received by all of the disks in a single transfer. A block can be broken down into subblocks of size b_i in a multiple-disk system, where b_i is the size of the subblock from the ith disk in the group. In our model all disks in a striped group have subblocks of the same size although this need not be a general requirement.†

Because of the requirement that only whole sectors be transferred from each disk (b_i must be evenly divisible by d_s, the number of bytes per disk sector), it may be the case that

$$b < \sum_{i=1}^{n} b_i$$

Such extraneous transferred information is assumed to be ignored at the destination.

A second consequence of this requirement is that transfer time, t_x, is not a smooth function of b since $b_i = b/n$. We can write the following expression for the time to transfer a block to or from an n-disk striped system:

$$t_x = \left\lceil \frac{b_i/d_s}{s} \right\rceil \frac{1}{\omega_{disk}} = \frac{b_i}{s d_s \omega_{disk}}$$

Here s represents the number of sectors per track on the disk and b_i/d_s is always a whole number as discussed above.

---

† In the COS implementation of striping [MASO84] the subblock sizes b_i may not be the same for all disks in the striped group. IO requests are broken down into pieces of a fixed size but these pieces may be distributed unevenly across the disks, depending on the total size of the request.

3. Performance Enhancements

Early data from the model confirmed that seek delays and rotational latency were the major contributors towards the response times of striped disk groups. With an eye towards reducing these times, several enhancements to the original striped disks were developed. Each of the enhancements acts to reduce either the seek delays or rotational latency of the disks. They can be included in the model in any combination, so that the effect of each on system performance can be determined. These enhancements are discussed in the following four sections.

Note that some enhancements may be easier to implement than others. We return to these implementation questions after Section 4, once we understand the performance gains that can be achieved with each enhancement.

3.1. Immediate Reading

The original model assumes that any transfer of a block of data to or from a disk must begin at the position of the beginning of the block on the disk. Therefore, if the disk head arrives at the appropriate track just after the beginning of the desired block has spun past, data transfer must wait until the disk has made almost a complete revolution. If instead we assume that the capability exists to begin transferring data from any point within the block, we can reduce the expected rotational latency of the disk. This new assumption is called the *immediate reading* enhancement.

Allowing for immediate reads changes the probability distribution of the rotational delay from that which was used in the standard case (a uniform distribution). With immediate reads, the distribution varies with the ratio of subblock size to disk track size. In fact, if we match the block size to the size of a track on the disk, we reduce rotational latency to zero since we never have to wait for the correct part of the track to spin under the read/write head. An equation for the new distribution of rotational latency is developed in Appendix D.

Note that immediate reading could also be used to enhance the performance of a single disk. In a striped group, however, rotational delays are more critical, since we must always wait for the slowest of the n disks in the group. Therefore we expect that enhancing a striped disk group in this manner will produce a more dramatic speedup than would the enhancement of a single unit.

3.2. Ordered Blocks

In this scenario, data is stored on the disk in the order in which it is to be retrieved. Frequently, there is some natural order to the data in an application. For example, a b-tree can be stored on disk one level at a time so that the disk head need never "backtrack" over previously visited cylinders to get the next block. Sometimes it doesn't matter in what order the data is read, for example if we are trying to compute the average salary found in a database of employee records. In these cases we can merely specify that the data is to be read in physically sequential order.

Typically, the price that is paid for this speedup is that adding and deleting data becomes more difficult. Going back to the b-tree example, we must either include extra blocks to handle "spill-over" data or rearrange the b-tree on the disk (presumably an expensive operation) when additions and deletions occur. However, ordered blocks may still be useful if the percentage of operations that cause modifications is small [WIED77].

An important consequence of ordering disk blocks is that seek times can no longer be considered probabilistic in the same sense as before. With ordered blocks there is more information as to the location of the next block to be read. In some cases the location may be completely specified. Where ordered blocks are applicable as an enhancement to applications considered in later sections, they are handled as a special case.

Ordered blocks can also be used to enhance the performance of single disks. However, as was the case for the immediate reading enhancement, we expect that a striped group employing ordered blocks will show a more dramatic performance improvement.

3.3. Matched Disks

Here we consider the possibility of requiring that the pieces of a block be stored at the same location on each of the disks in a multiple-disk system. This way, the read/write heads of all of the disks will move together. Matching the disks means that the contribution of seek time to $f^{n}_{(s+r)}(t)$, the distribution of seek and rotational delay for an n-disk group, is simply $f^{1}_{s}(t)$ regardless of the value of n. This implies that the expected seek time will remain approximately constant as the number of disks in the system increases, since all of the disks are doing the same seek.

It would be convenient to assume a similar matching of rotational delays. Unfortunately, without synchronization rotational velocities are expected to vary from disk to disk (typically a 5% to 10% tolerance is specified). Matched disk striped systems give rise therefore to a seek

and rotational delay (with distribution $f^{n}_{(s+r)}(t)$) whose expected value increases with the number of disks in the system (because of the varying rotational delays), but not as quickly as it would have in an unmatched system.

3.4. Subblock Screening

Sometimes, when reading blocks of data from a striped disk group, we are not interested in all of the data in the block. Instead, we may be interested in only a single datum located in one of the subblocks. For example, in the b-tree task discussed in section 4.2, each block is a b-tree node consisting of an ordered set of pointer/key pairs. We are interested in locating one such pair and following its pointer to the next node.

If it were possible quickly to check each subblock when it arrived in memory and eliminate those which could *not* hold the data of interest, we could decrease both IO and processing time for the task. We call this the *subblock screening* enhancement. For the b-tree task, where the data in each subblock is ordered, we can check by determining whether our test key falls between the smallest (first) and largest (last) keys in the subblock. This would be a relatively fast operation since it involves checking only two of the values from each subblock. For simplicity, we assume the checking time is negligible.

We should note here that an implementation of this enhancement would be application specific and therefore is note quite in line with our original goal of a transparent striping system. However, the potential benefits here seemed significant and worth investigating. Our aim is to shed some light on the magnitude of the speedup obtainable through such a tradeoff.

Subblock screening saves CPU time because only the subblock containing the necessary datum need be (fully) processed. Interestingly enough, screening also reduces the expected seek and rotation delay for the striped group. If the screening can be accomplished very rapidly, it is as if we have a disk group which can automatically "guess" which disk holds the desired subblock. It appears that there is only one, rather than n, disks in the system when seek and rotational delays are determined. As a result, the expected delay from disk seeks and rotation remains approximately constant as n increases.

The effect of the subblock screening enhancement is explored in section 4.2, which presents the b-tree task. Unfortunately, this enhancement is not applicable to the other tasks (file processing and merge sort) since they require that all subblocks be present for processing.

4. Applications and Results

To fully evaluate striping we must look at an entire application, not just at an isolated disk to memory data transfer. Therefore we selected several tasks representing a variety of data access and computational patterns. Each was analyzed using the model presented in the last section to determine the effectiveness of striping in that particular situation. Most of these tasks involved multiple disk IO operations. All but one, the block fetch task, included some measure of processing time once the data was present in memory.

The effectiveness of striping for any task is the percentage of speedup obtainable by striping the task's data set across n disks rather than storing it on a single disk. This is measured by comparing the task completion time in the striped system to the task completion time in the other. For example, consider a situation in which a task can be completed by a single-disk system in 2.5 seconds, while the same task is finished in 1.8 seconds by a system with a 3-disk striped group. We compute the effectiveness of the striped group by:

$$effectiveness = \left[\frac{2.5 - 1.8}{2.5}\right] 100 = 28\%$$

Under this definition, the maximum possible speedup is 100%. Negative effectiveness indicates that the task completion time for the striped group is greater than that for the single-disk system.

For the sake of comparison, all of the data presented in the following sections was collected using a common set of values for most of the model parameters. Of these parameters, those directly concerned with the disk were assigned values corresponding roughly to actual values specified for DEC's RA81 disk drive [DEC82]. Other common parameters take into account CPU time spent initializing disk transfers. All are listed in Table 4.1. Any variations in these values are noted in the sections describing the various tasks. Data resulting from parameter values corresponding to other disk drives besides the RA81 are presented for comparison after the tasks are discussed, in section 4.5.

4.1. Block Fetch

This task involves the transfer of a data block of fixed size from disk to a memory buffer. This is the basic task in that no processing time is included except that necessary to initialize the disk transfer. We can write an expression for t_n, the completion time for this task using an n-disk striped group, as follows:

$$t_n = t_{ioc} + nt_i + E[f^n_{(s+r)}(t)] + t_x$$

symbol	description	value	units
t_{ms}	maximum seek time	50	milliseconds
t_{os}	seek time overhead (see Appendix C)	7	milliseconds
ω_{disk}	disk rotational velocity	3600	rpm
h	# of read/write heads	14	
c	# of cylinders	1258	
s	# of sectors per track	52	
d_s	bytes per sector	512	bytes/sector
t_{ioc}	CPU overhead time	10	microseconds
t_i	CPU per disk overhead time	100	microseconds

Table 4.1

We use $E[f(x)]$ to represent the expected value of probability density function $f(x)$. In the above expression, the first two terms represent time spent initiating the transfer at the CPU. The remaining terms represent seek and rotational delays and transfer time, respectively.

Figure 4.1a shows the effectiveness of striping for the block fetching task with block sizes ranging from 1K to 256K bytes. It is clear that striping is more effective for larger block sizes, although significant savings can be obtained for block sizes as small as 1K bytes. That large block sizes provide an ideal environment for striping is to be expected since transfer time is exactly the component of the task completion time that striping works to reduce.

Figure 4.1b represents the effect of varying several of the parameters of the system, one at a time, by as much as $\pm 40\%$. The CPU processing constants t_{ioc} and t_i, the disk rotational velocity ω_{disk}, and the maximum single-disk seek time t_{ms} are varied. Effectiveness for a striped group with four disks and a block size of 16K bytes is shown on the graph.

The curves for ω_{disk} and t_{ms} are the most interesting here, for they represent results that would have been difficult to predict using simple intuition. For example, as ω_{disk} increases, the transfer time decreases (diminishing the effectiveness of striping) and at the same time rotational delays decrease (improving the effectiveness of striping). Figure 4.1b indicates that the latter effect is more significant.

Also surprising is the fact that striping actually becomes *more* effective as t_{ms} increases. This is a reflection of the seek time reductions that come from spreading the data set across more and more disks (reducing the size of the sub-partitions). These reductions overcompensate for the increases in expected seek time that come from striping across more disks.

4.2. B-tree

The b-tree task is to locate a record in a large database using a b-tree and a key (to identify the target record). Each b-tree node corresponds to a disk block. The depth d of the b-tree is given by

$$d = \left\lceil \log_{k_b}(r_d) \right\rceil = \left\lceil \frac{log(r_d)}{log(k_b)} \right\rceil$$

where r_d is the number of records in the database and k_b is the number of pointer/key pairs per b-tree node.

Once a block is in memory, a binary search can locate the appropriate pointer/key pair in time $c_b \log_2(k_b)$. The proportionality constant c_b is a model parameter. Completion time for the task can now be expressed as

$$t_n = d\ (t_{ioc} + nt_i + E[f^n_{(s+r)}(t)] + t_x + c_b \log_2(k_b))$$

since we must retrieve and search through d b-tree nodes to locate the target record.

Values of parameters specific to the b-tree task are given in Table 4.2.

symbol	description	value	units
r_d	# of records in the database	3000000	records
s_k	size of pointer/key pair	10	bytes
c_b	binary search time constant	50	microseconds

Table 4.2

The parameter s_k, along with the block size b, determines k_b. The block size is not specified as a parameter. Rather, we compute completion time for an n-disk system using a variety of block sizes. The smallest of these times is taken as the task completion time for that system. In this sense, completion times are computed using an *optimal* block size for each system.

The b-tree task was chosen for a number of reasons. By not fixing the block size, we have created a tradeoff between minimizing the number of blocks transferred (by increasing the block size) and minimizing the amount of unnecessary data transferred (by decreasing the block size). In addition, with the b-tree task it is not known which node must be retrieved next until processing on the current node is completed. One possible method of sidestepping this problem is introduced in [GHOS69]. It involves the use of values in the b-tree nodes and the

value of the search key to make an educated guess at the next block. For simplicity, we have assumed that the b-tree is traversed without the aid of such a scheme.

Some results from the b-tree task are described in Figures 4.2a and 4.2b. Figure 4.2a shows completion times as the file size is varied. The performance gains due to striping are modest and much less than we had expected for this application. (The effectiveness values for Figure 4.2a are in most cases less than 10%, significantly lower than those produced by the file processing and merge sort tasks.) The most important reasons for this difference lie in the nature of the b-tree task itself, particularly in the block size tradeoff. Blocks which produce optimal completion times are not necessarily large enough to allow significant improvement through striping.

The jagged nature of the lines of Figure 4.2a is a result of two factors. First, block sizes are only optimized over a relatively small set of possible values. If s is sectors per track and c is tracks per cylinder, we consider subblocks of 2^i sectors where $2^i \leq s$ and $i \geq 0$ and subblocks of si sectors where $si \leq c$ and $i \geq 1$. Second, the depth of the b-tree, which directly affects the task completion time, is a discontinuous function of the block size. It is possible to reduce the non-monotonic behavior of the curves by increasing the size of the set of possible block sizes, however they have been left in this form. The curves provide a graphic reminder of the importance of the choice of block size when working with b-trees.

For all file sizes, the graphs show a general trend towards decreased task completion times as the size of the striped group increases. Effectiveness values for these curves would be as high as 20% in the best case, which is significantly lower than those produced by the file processing and merge sort tasks. The most important reasons for this difference lie in the nature of the b-tree task itself, particularly in the block size tradeoff. Blocks which produce optimal completion times are not necessarily large enough to allow significant improvement through striping.

Figure 4.2b shows the how the various enhancements alter the effectiveness of striping. Note that the effectiveness measurements are made with the enhancements applied to both the striped and the unstriped systems. Screening subblocks appears to be by far the most effective of the enhancements for this task. On the other hand, striping and ordered disk blocks do not seem to work well together. In this particular instance, striping the disks actually results in worse performance. Again, the reasons are peculiar to the b-tree task in combination with the ordering enhancement. In this case, the retrieval time did not benefit from a reduction in the b-tree depth caused by striping because retrieval of the additional block in the non-striped version was already very cheap.

All of this does not mean that ordered b-trees are searched more slowly than unordered ones, but only that striping will probably be more helpful in the latter case. It is interesting to note that if the size of the b-tree is increased striping becomes beneficial even in the ordered case because of additional speedups from reductions in the size of sub-partitions as the b-tree is spread across several disks. Also, in the file processing task described in following section we find that striping is beneficial even when blocks are read in order. In this case the difference lies in the optimality of larger block sizes for that task.

4.3. File Processing

The file processing task is to read and perform some type of operation on the records in a file. Depending on which of two variations of the task was chosen, the processor then may or may not be required to write the records back out to the disks. For all of the data presented in this section the latter variation is to be assumed unless otherwise specified

The file processing task requires several unique input parameters. These are listed in Table 4.3.

symbol	description	value	units
r_d	size of the database	10^8	bytes
m	size of memory buffer	32K	bytes
p	processing time per byte	1	microsecond

Table 4.3

Processing of the data is simulated by the specification of a processing constant p, the amount of time necessary for the processing of each byte. The number of bytes of memory available for the task is specified by the parameter m. As in the b-tree task, b, the block size, is not specified as a parameter. Instead, computations are done for a variety of block sizes and the smallest time is used as the completion time for the task. Block size is restricted to be no more than one-half of the memory size so that we have room for double buffering.

It is assumed for this task that disk requests can be queued. The processor initially requests enough blocks to fill the memory buffer, then begins processing the first block. After each block is processed, a request to write the used block back to disk is queued, followed by a request to read in a new block if any remain. Processing of the next block is then begun. Letting k represent the number of blocks that can fit in the memory buffer, we summarize this protocol with the simple program shown in Program 4.3.

```
initial phase:  REPEAT k TIMES {
                    -IF (unrequested blocks remain) THEN
                        queue a block read request
                }

main phase:     WHILE (not all records processed) DO {
                    -process a block in the buffer
                    -queue a block write request for the processed block
                        (if necessary)
                    -IF (unrequested blocks remain) THEN
                        queue a block read request
                }
```

Program 4.3

No scheduling optimization is modeled for the disk request queue; requests are assumed to be handled on a first-come-first-serve basis.

Analysis of the file processing task is more complicated than earlier analyses since the processing of data and disk IO are simultaneous. An estimation as to whether the completion time will be IO-bound or processor-bound is used in determining the completion time. A detailed look at how this determination is made can be found in Appendix A.

Results from the file processing model are shown in Figures 4.3a-d. The first Figure (4.3a) shows the effectiveness of striping for two different file sizes. Even in this simple case, with no enhancements, effectiveness is about twice that found for the b-tree task. This is because the in file processing there is no penalty for large block sizes as there is with the b-tree. The system is free to read the biggest blocks possible as long as the buffer size constraint is not violated.

Like most of the effectiveness curves in this paper, those for the file processing task tend to flatten out as n increases. Two factors that contribute to this. First, the expected value of the sum of seek and rotation time in a striped system approaches a fixed maximum value, namely $t_{ms} + t_{ml}$, as n increases. Second, as the number of disks increases, the subblocks become so small that transfer time becomes negligible. Where these effects become noticeable depends on the characteristics of the system. After that point, the addition of additional disks to the striped group causes little change in its effectiveness.

Figure 4.3b shows the important and somewhat unexpected effect of memory size on effectiveness. When only an 8K buffer is available, blocks are limited to this size, making striping definitively not worthwhile. With 32K available, striping becomes effective. However, if memory is increased beyond this value, effectiveness decreases once again! This is because the

extra memory lets us decrease IO times to the point where IO is no longer a bottleneck. This is an important property of any system in which IO and processing are occurring simultaneously: once the task is no longer IO-bound, adding additional disks to the striped group will not change its effectiveness. With 64K bytes of memory, the disks begin to outrace the CPU when $n = 3$. With 128K, even when $n = 1$, the CPU cannot process the data as quickly as the disk can supply it.

Figure 4.3c shows that the same limiting effect arises if we vary p, the processing time per byte. As p is increased from 1 to 5 microseconds, the task changes from IO-bound to CPU-bound. In the case where $p = 1.5$ microseconds, the system becomes CPU-bound after the number of disks becomes greater than two.

Finally, Figure 4.3d shows the effect of the enhancements on task completion time. In this case ordering the blocks can result in a significant reduction in completion time. Matching disks has little effect in this example because seek times are already small in comparison to other factors (i.e. rotational latency). With larger file sizes or smaller disks, matching becomes a more effective enhancement.

4.4. Merge Sort

The final task is a balanced 2-way merge sort [KNUT73]. We assume that we begin with an unsorted database on an n-disk striped group. The task is to sort the database and store the sorted version on the same striped disks.

Parameters for the merge sort task are described in Table 4.4.

symbol	description	value	units
r_d	# of records in the database	1500000	records
s_r	size of a record	100	bytes
m	size of memory buffer	32K	bytes
c_s	sort time constant per record	50	microseconds
c_m	merge time constant per record	10	microseconds

Table 4.4

The merge sort task uses two processing constants. The initial phase of the merge sort involves sorting k-record buffers which takes time $c_s k\, log(k)$ per buffer. The constant c_s is a model parameter. The newly sorted segments are called *runs*. During the second phase of the task, pairs of runs are combined (merged) into longer runs of twice the original length. The processing time involved here is c_m per record merged, where c_m is also a model parameter.

The block size, b, is a function of the memory buffer size m. Ideally, we would have $b = m$, however it may be the case that $b < m$ since records will not always pack evenly into the memory buffer. Actually, the processor would need more than m bytes of memory to accomplish the merge sort efficiently. In fact during the merge phase a total of $2m$ bytes might be in use simultaneously. We assume that this additional scratch space is available.

The merge sort task involves multiple reads of the complete database. Specifically, if the initial sorted runs are k records long, we must make

$$\left\lceil \log_2 \left(\frac{r_d}{k} \right) \right\rceil$$

passes through the database since the number of runs is halved with each pass. There is also one additional pass to create the initial runs. A detailed description of the merge sort algorithm can be found in Appendix B.

Figures 4.4a-d describe some results from the merge sort model, showing the effects of variations in certain parameters. The first Figure (4.4a) shows the effect of changing the size of the data file. The effectiveness seen in each of these cases is comparable to that found for the file processing task.

The next two Figures (4.4b-c) show the effects of changing memory buffer size and the processing constants (c_s and c_m), respectively. Increasing the buffer size allows the use of larger disk blocks. As was seen in previous tasks, larger block sizes provide a better environment for striping and result in higher effectiveness. The processing constants do not have as much of an effect as the block sizes. Figure 4.4c shows the effect of doubling or halving the original values of both c_s and c_m. Longer processing times result in decreased effectiveness since in those cases the IO time represents a smaller fraction of the total task completion time, t_n. Striping, which affects IO time, will therefore have relatively less effect on the total time for the task. This situation is in sharp contrast to the file processing task because processing and disk IO are not overlapped. Thus we do not have the notions of "CPU-bound" or "IO-bound" when studying the merge sort.

Figure 4.4d compares the merge sort with the immediate reading enhancement to merge sort without. Immediate reading is the only enhancement which can be applied to the merge sort task since we assume some ordering of the blocks in our analysis of the task (see Appendix B) to keep completion times from becoming outrageous.†

† Actually, the matching enhancement can also be applied to this task. In this case seek times were small

4.5. Other Disk Drives

This section is included to show the effect of the choice of disk drives on the performance of a striped group. Two additional drives, the RA80 and the RP06 are compared to the RA81 drive, which was used throughout the previous sections [DEC81a, DEC81b, DEC82].

Values of parameters for all three drives are listed in Table 4.5 for comparison.

symbol	description	RA81	RA80	RP06	units
t_{ms}	maximum seek time	50	50	60	milliseconds
t_{os}	seek time overhead (see Appendix C)	7	6	10	milliseconds
ω_{disk}	disk rotational velocity	3600	3600	3600	rpm
h	# of read/write heads	14	14	19	
c	# of cylinders	1258	561	815	
s	# of sectors per track	52	31	22	
d_s	bytes per sector	512	512	512	bytes/sector

Table 4.5

The task used for comparison was Block Fetch, since it is the simplest. Each disk is asked to retrieve a single 32K byte block. Striping's effectiveness is shown in Figure 4.5.

An examination of the characteristics of the disk drives reveals why some are more advantageous to striping than the others. Simply stated, if their rotational speeds are constant then striping works better for drives with lower bit-per-track densities. This is so because, for a fixed block size (i.e. 32K bytes), a lower bit-per-track density means a longer transfer time t_x. This is the situation for which striping is the most effective, since it acts to reduce transfer time.

The same statement could also be made for disks with slower rotational velocities, since this again would translate to longer transfer times. However, the effect is partially counterbalanced in this case because slow rotational velocity also means longer expected rotational latency. This leads to less effectiveness since striping tends to bring out the worst in rotational delays.

---

enough that completion times with and without matching were nearly identical. Certain parameter changes, such as a larger data file, might have altered this situation.

5. Reliability

As mentioned in the introduction, there is a *potential* problem with disk striping: loss of a single disk can make an entire striped group unavailable. Although the main focus of this paper is the performance of striping, it is still important to understand the implications of the reliability problem.

In this section we will argue that (1) in many cases of interest striping does not imply lower reliability, and (2) in those cases where there is a problem, reasonable steps can be taken to improve reliability. Our goal is not to present detailed models and analyses of striping reliability; it is only to convince the reader of the above two points.

Our comparisons will deal with the reliability of a non-striped file and striped one. To make these comparisons fair, we will assume that both files are stored on a system with identical hardware. Specifically, even if the non-striped file fits on a single disk, we will assume that the system has the same number of disks as the striped system. If we do not make this assumption, then we are not only studying the effects of striping but also of the different hardware, and this can confuse the issues. (A system with more disks has more components that can fail, but also more storage capacity. These factors are independent of whether striping is used or not.)

Striping a file does not necessarily imply lower reliability. (As we will see in a moment, it can even mean higher reliability.) For one, if the non-striped file is large, it may also occupy several disks. The data would be stored in a different order, but the large file would still be unavailable when any of its disks were unavailable. For instance, in a memory resident database system, the backup database stored on disk is usually considered a single file that occupies the same number of disks, whether it is striped or not. If any disk unit is broken, it will not be possible to re-load the database after a crash.

If the file is important, many applications would have backup copies and an activity log for the file, regardless of whether the file was striped or not. (The log records changes to the file as they occur. The log is stored on a device where no files are kept, e.g., a dedicated tape unit.) In this case again, striping would usually have no adverse effect on reliability. After a disk failure, the disk, striped or not, is loaded from the backup and then brought up to date with the log. System down time would be the same in either case.

Even if a file is not backed up, striping could yield good availability because it speeds up tasks. To illustrate, suppose that we want to execute a task that takes T seconds and uses a file during this time. (Say the file is discarded after the task.) Assume that with probability $p = 0.99$, a disk unit will not fail during the T seconds. If the file is not striped and fits on a

single disk, the task will successfully complete with probability $P = p = 0.99$. If the file is striped over 4 disks, then $P = p^4 = 0.96$. So striping appears to have lower reliability.

However, a more detailed analysis shows that this may not be so. With striped disks, the task completes faster, say in $T/2$ seconds. (See figure 4.4b, $n = 4$, for example). The probability that a disk does not fail in this interval is approximately $p' = 0.995$ (the probability that it fails is roughly half of the previous amount). Hence, the striped task will be successful with probability $P = (p')^4 = 0.98$.

The reliability is still lower than for the non-striped case. However, we have ignored the CPU. Processors fail more often than disks, so it is reasonable to sy that the CPU does not fail in T seconds with probability $q = 0.97$, in $T/2$ seconds with probability $q' = 0.985$. Now, the non-striped task is successful with probability $P = qp = 0.960$, the striped one with probability $P = q'(p')^4 = 0.965$. Of course, this is just a simple example, but it does illustrate that striping does not necessarily lower availability.

Finally, note that in cases where an n-striped does yield lower reliability, the loss may be insignificant if n is small. In our previous example, even if the task takes the full T seconds with striping, the successful completion probabilities are $P = p^2 = 0.98$ for $n = 2$ and $P = p^3 = 0.97$ for $n = 3$. These values are not much lower than the $P = 0.99$ of a non-striped task, and may easily be tolerable in return for the higher performance of striping. Furthermore, we have seen that striping gains are largest with small n, so it is precisely these small numbers that will be the most common in practice.

There are of course, cases where striping can cause a significant loss in reliability. For instance, suppose that files are backed up only at midnight. Let us say that a new file has just been created. If the file is not striped and stored on a single disk, the probability that the file makes it intact until midnight may be $p = 0.99$. If the file is striped on 4 disks, the probability is $p^4 = 0.96$. (This comparison must be made with care, however, since in a given time interval more work will be accomplished by the faster striped system.)

In such cases, one can add redundancy to the striped file to improve its reliability. For example, suppose we want to cope with the failure of a single disk of the striped group. We can use an extra disk with parity bits. For each stored block, there will be a corresponding sub-block in the extra disk. The first bit in the sub-block will be the parity bit for the set of bits that constitute the first bit of the other sub-blocks. The second bit will be the parity for the second set of bits, and so on. This is illustrated in the diagram below, for 4-way striping:

Disk 1	Disk2	Disk3	Disk4	Parity Disk
a_1	b_1	c_1	d_1	$a_1 \oplus b_1 \oplus c_1 \oplus d_1$
a_2	b_2	c_2	d_2	$a_2 \oplus b_2 \oplus c_2 \oplus d_2$
a_3	b_3	c_3	d_3	$a_3 \oplus b_3 \oplus c_3 \oplus d_3$
a_4	b_4	c_4	d_4	$a_4 \oplus b_4 \oplus c_4 \oplus d_4$
⋮	⋮	⋮	⋮	⋮

When a failure occurs, the error detection codes internal to each disk will be used to identify the unit and sub-block that has been lost. Then the parity sub-block can be used to reconstruct the missing sub-block. If having delays in the correction phase is tolerable, then the parity block does not have to be read until an error is detected. To correct more than a single failure, this example can be generalized using well known erasure correcting codes [BERL68].

This strategy obviously has a cost: an extra disk is required, $1/n$ extra storage is used, and writes are more expensive (the parity sub-blocks have to be computed and written). However, not only have we improved reliability, we have probably surpassed the no-striping case. In our file loss example, the non-striped, single disk file is safe with probability $p = 0.990$. With redundant striping, the file is safe as long as 4 out of 5 disks do not fail, i.e., with probability $p^5 + 5\,(1-p)\,p^4 = 0.999$.

It is interesting to note that as n, the number of striped disks, increases, the redundant storage and the availability decrease. With $p = 0.99$ and $n = 15$, only 6% extra storage is needed and the reliability is 0.990, the same as that of the non-striped file. Beyond $n = 15$ (an unlikely case), we would require a 2 or more error correction strategy to make reliability higher.

If an extra disk is not available for storing the parity information, it could also be stored on the striped disks. The following diagram illustrates this for $n = 4$[†]:

Disk 1	Disk 2	Disk 3	Disk 4
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2

† In an actual implementation, parity bits would be stored at the end of each disk block instead of immediately following their associated data. They are shown immediately following their data in the diagram to make the error correction concept clear.

a_3	b_3	c_3	d_3
$b_3 \oplus c_3 \oplus d_3$	$c_2 \oplus d_2 \oplus a_3$	$d_1 \oplus a_2 \oplus b_2$	$a_1 \oplus b_1 \oplus c_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

The parity bit for a group of $n-1$ data bits is stored on the remaining disk. This way, the parity bit is always available in the case of a failure. (The strategy can also be generalized for more than 1 error detection).

This strategy is slightly more expensive in terms of redundant storage (it is $1/(n-1)$ as opposed to $1/n$) and write overhead (writes of parity information tie up the disks where the data is stored). However, there is no need for an extra disk unit, and the reliability is improved even more. In our running example, with this strategy the file will be safe with probability $p^4 + 4(1-p)p^3 = 0.9994$. The extra disk strategy had reliability 0.9990 and the no-striping strategy 0.9900.

In summary, striping can be evaluated along two dimensions: performance and reliability. We have seen that in many cases, the reliability of striping is comparable (or even higher) to that of no striping, so that the performance comparisons given in Section 4 are fair.

In cases where "plain" striping does not give adequate reliability, making concrete performance comparisons, of any type, would not be entirely fair. The results of Section 4 can be considered "optimistic" in this case because of the lower reliability. However, if one adds th overhead of redundant storage to the performance model, the results would be "pessimistic" for striping because it then increases reliability beyond the no-striping case. In light of these complications, and because of space limitations, we do not discuss this further. However, it should be fairly clear to see how our performance model can be extended to cover the overhead of redundant storage.

6. Conclusions

Our results on striping seem to be mixed. On the one hand, the performance improvements observed for the "typical" tasks are in 10% to 70% range (speedups of up to about 3 to 1), modest in comparison to the orders of magnitude improvements sought by supercomputer efforts. On the other hand, the striping improvements have been achieved at a very low cost. Many existing computers already have multiple disks and controllers, so the hardware cost of striping may be zero. The software development cost may also be small, especially if a general

purpose striping facility is implemented and its cost amortized over many applications.

Striping also promises to become much more effective in the near future. Processors are becoming faster, memories larger, but disk speeds are not changing very much. At the same time, data requirements are growing in many applications. All of these factors make striping more effective. For example, in Figure 5 we plot the effectiveness of striping not for a typical file processing task as was done in Section 4.3, but for a task that is likely to be encountered in 5 years. We have set the processor speed at one nanosecond per byte (achievable with parallel processing), the file size at 2 gigabytes ,and the memory buffer at half a megabyte. (The curve begins at $n = 5$ since we need five of our disks to hold the 2 gigabyte file.) We have also replotted the curve for a 200 megabyte file with a 32K buffer, as seen in Figure 4.3a, for comparison. In the new scenario, striping speedups are 3 to 1 or more, and clearly striping pays off.

Our results show that striping has limitations and must be used with care. Specifically, striping does not seem to be useful in applications where we must search through a large disk data structure (e.g. a b-tree) as opposed to ones where we read and process the entire structure. In all cases, the speedup obtainable by striping tends towards a constant value as the number of disks increases. Finally, striping is very sensitive to parameters such as available memory buffer space and the amount of processing that must be performed on the data. This means that selecting the optimum number of disks, n, to stripe for an application is difficult, especially if the same data file is going to be used in different ways.

The striping enhancements can significantly improve performance and at least some of them should be included in a general purpose operating system facility. The immediate reading enhancement should probably be included. It provides significant improvements, and should be relatively easy to implement. The improvements of the matched disk enhancement are not as significant, but this enhancement would probably also be included because it could actually simplify system implementation (each subblock would have the same address on the disk). Ordering the blocks can improve performance, but in this case the application must inform the system where blocks should be placed. The block screening strategy does not seem to be a good candidate: it only improves performance in searching tasks, and would be more difficult to implement.

For many applications, striped disks may actually provide greater reliability than unstriped disks. If striping can produce a significant reduction in the completion time for a task, this shorter "failure window" may counterbalance the negative aspect of striping disks reliably: that the failure of a single disk can make the entire group unavailable. Comparing the reliability of striped and unstriped disk groups also depends on the application programs

that will use them. Large (greater than the size of a disk), monolithic data sets will not be any more reliable on unstriped disks than striped ones; any disk failure makes the data unavailable.

If reliable striping is a problem for an application, reliability can be boosted through the use of error correcting codes stored across the disks as well as within each disk. Extra disks can be used to store the error correcting information, or it can be stored on the same disks that hold the data. Availability can be brought to whatever level is desired by increasing the amount of error correcting information that is stored.

Bibliography

[BERL68] Berlekamp, E. *"Algebraic Coding Theory,"* McGraw-Hill, 1968, pp. 229-231.

[BORA83] Boral, H.,DeWitt, D.J.,"Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Database Machines,* Leilich,H.-O,Missikoff, M., ed., Springer-Verlag, 1983, pp. 166-187.

[CONV86] *"Convex System Managers Guide,"* Doc. #710-000330-000, Rev. 3.0, Convex Computer Corp, 1986.

[DEC81a] *"RA80 Disk Drive User Guide,"* Digital Equipment Corporation, 1981.

[DEC81b] *"Peripherals Handbook,"* Digital Equipment Corporation, 1981.

[DEC82] *"RA81 Disk Drive User Guide,"* Digital Equipment Corporation, 1982.

[DEWI84] Dewitt,D.J.,Katz,R.H.,Olken,F.,Shapiro,L.D.,Stonebraker,M.R.,Wood,D., "Implementation Techniques for Main Memory Database Systems," *SIGMOD Record,* Vol.14,#2,1984.

[GARC84] Garcia-Molina, H.,Lipton, R.J.,Valdes, J.,"A Massive Memory Machine," *IEEE Transactions on Computers,* Vol. C33, No. 5, May 1984, pp. 391-399.

[GHOS69] Ghosh,S.P.,Senko,M.E.,"File Organization: On the Selection of Random Access Index Points for Sequential Files," *JACM,* Vol. 16, No. 4, October 1969, pp. 569-579.

[GRAY83] Gray, J.,"What Difficulties are Left in Implementing Database Systems," Invited Talk at *SIGMOD Conference,* San Jose, CA.,May 1983.

[HONE84] Honeyman, P., personal communication.

[KIM84] Kim, M.Y.,"Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving," *Proc. of the Fourth Int'l. Workshop on Database Machines,* March, 1985, pp. 299-329.

[KNUT73] Knuth,D.E., *"The Art of Computer Programming,"* Vol. 3, Addison-Wesley, 1973, pp.361-371,471-479.

[MASO84] Mason, D., Cray Research Inc., personal communication.

[SIEG79] Siegel,H.J.,Kemmerer,F.,Washburn,M.,"Parallel Memory System for a Partitionable SIMD/MIMD Machine," *Proc. 1979 Int'l. Conference on Parallel Processing,* pp. 212-221.

[WALL86] Wallach, S., Vice-President, Technology, Convex Computer Corp., personal communication.

[WIED77] Wiederhold, G., *"Database Design,"* McGraw-Hill, 1977, Chaps. 2,3.

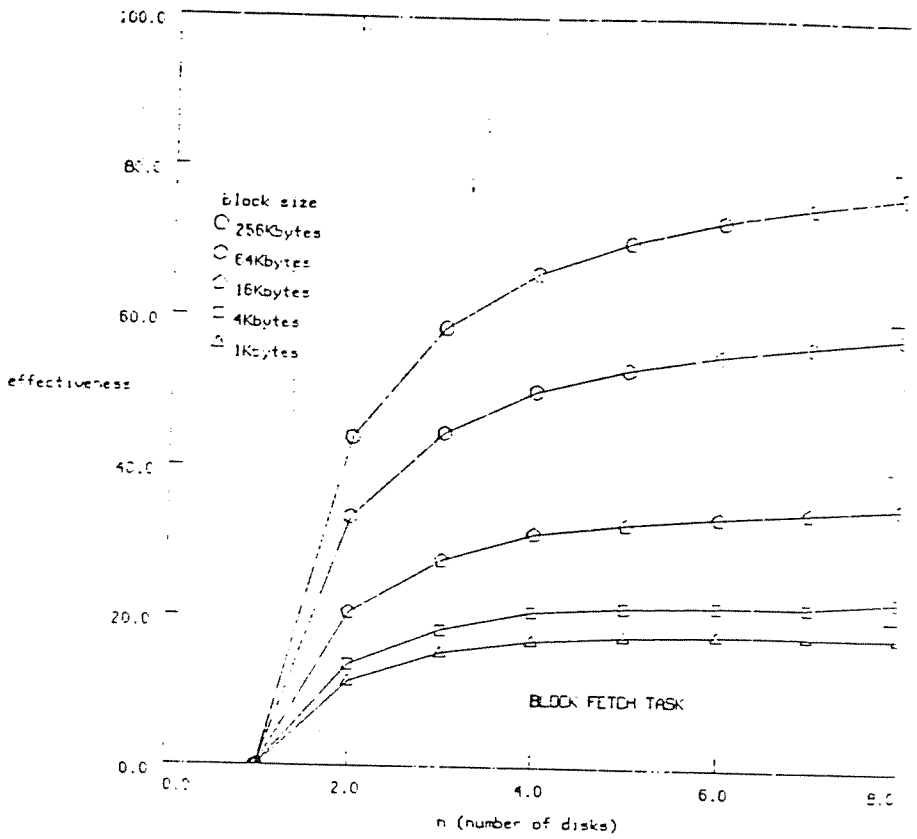


FIGURE 4.1a

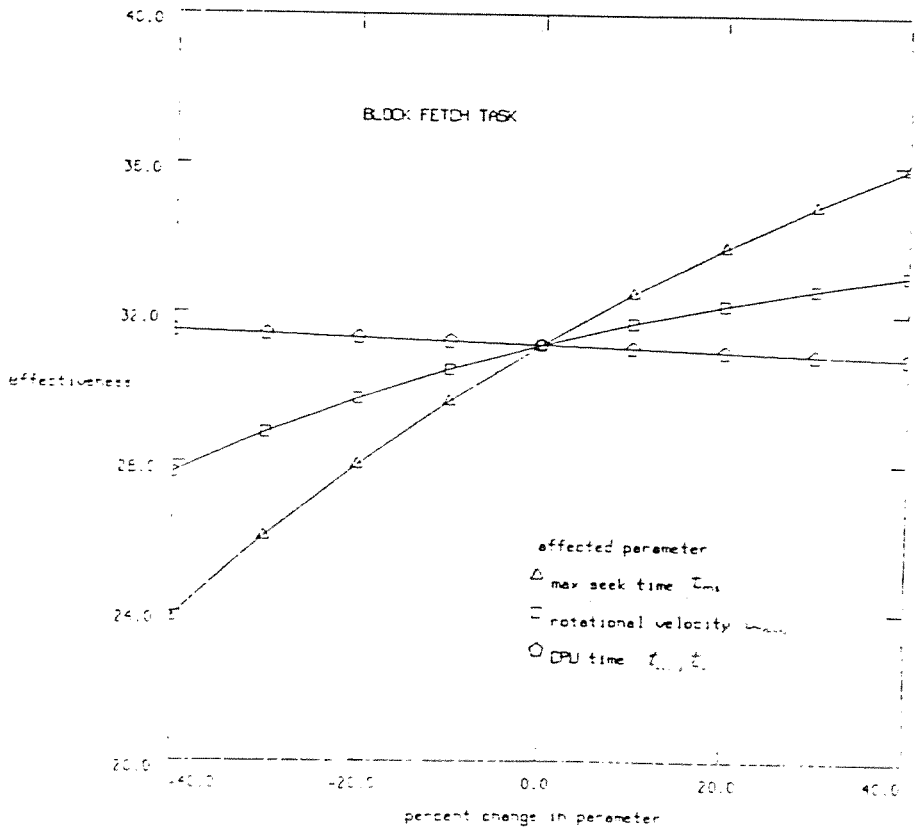


FIGURE 4.1b

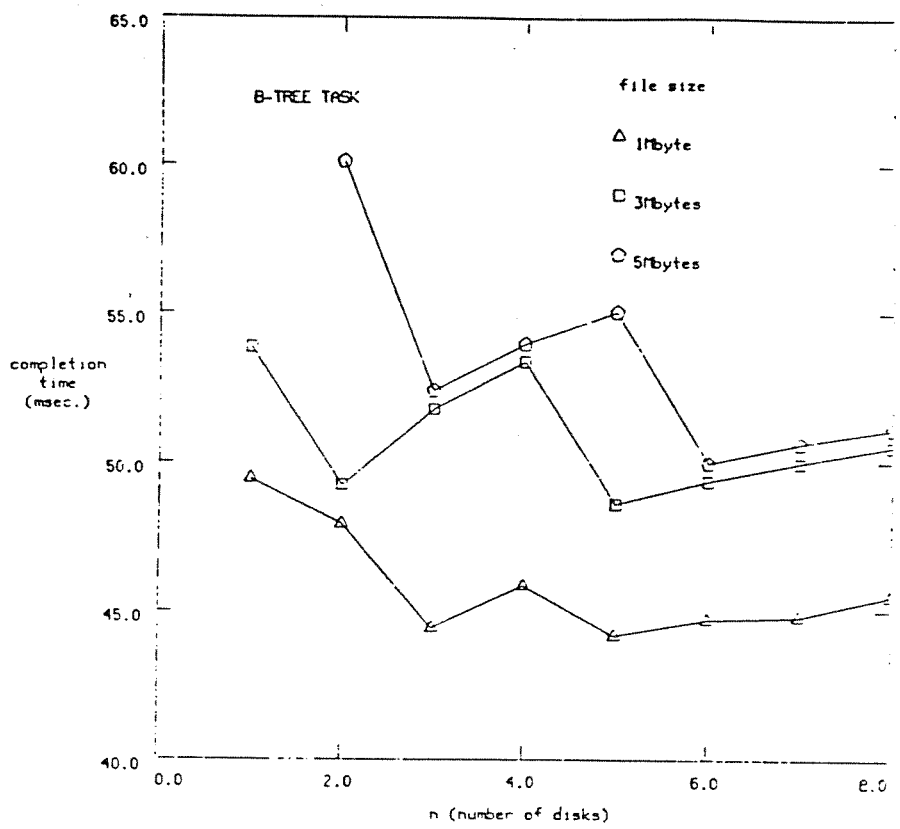


FIGURE 4.2a

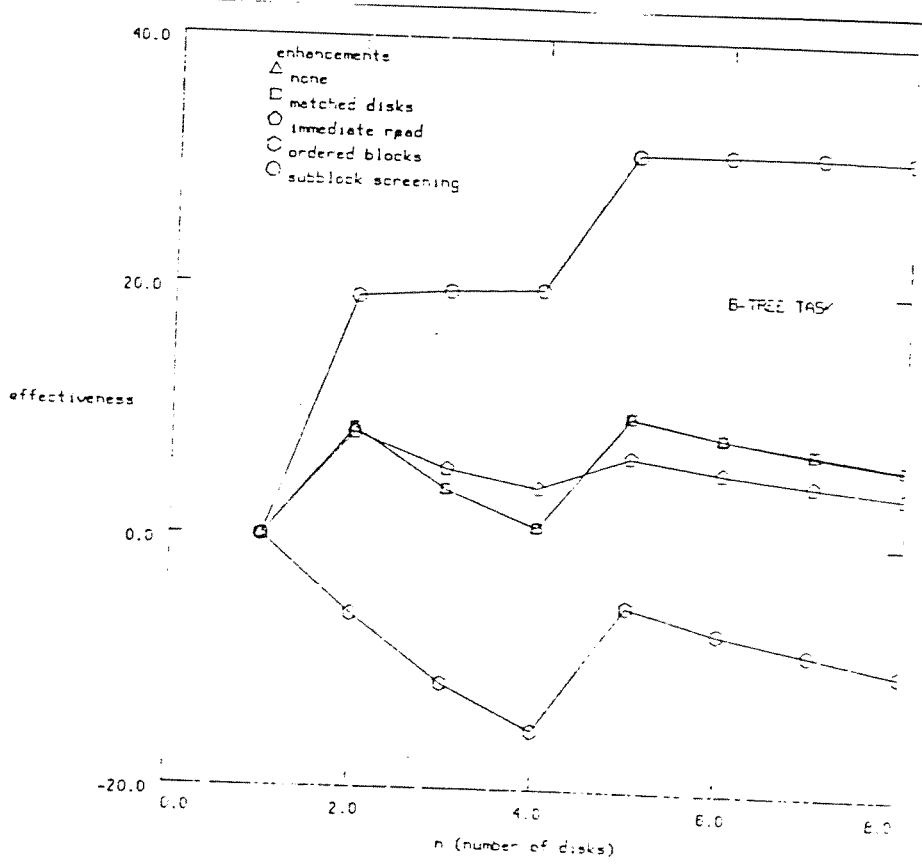


FIGURE 4.2b

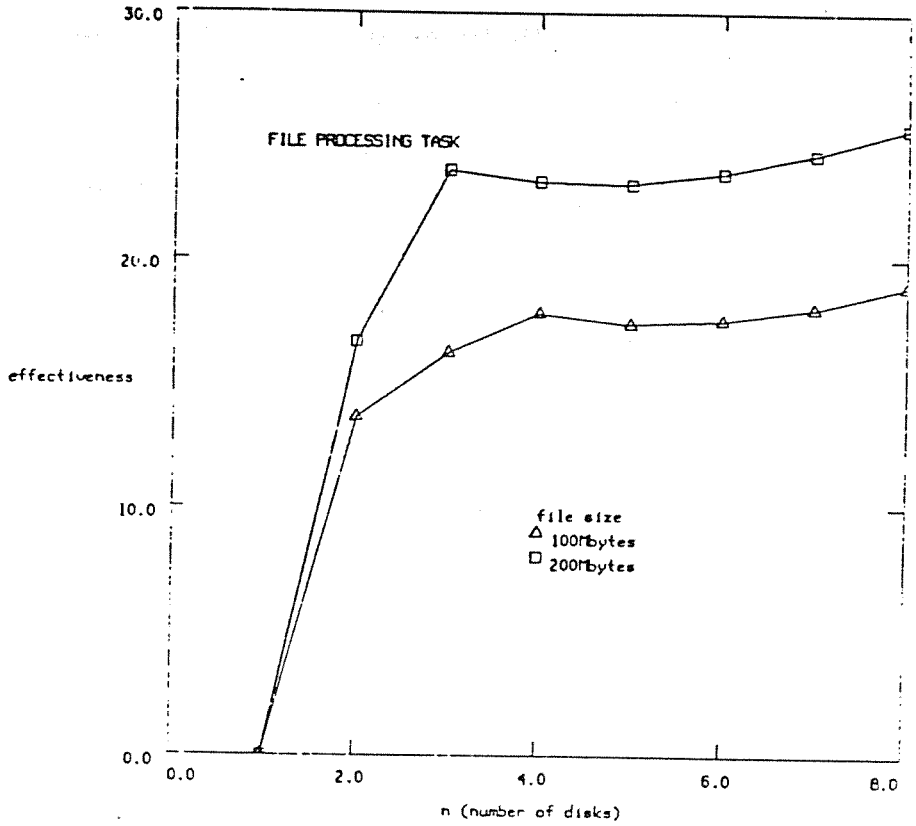


FIGURE 4.3a

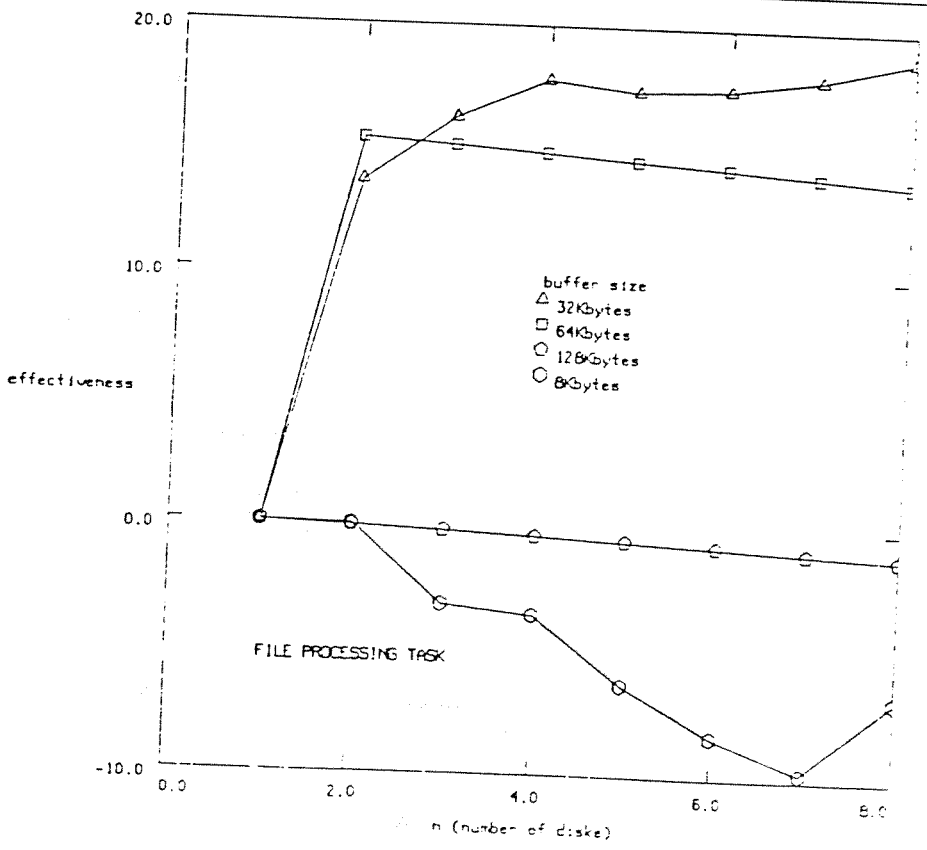


FIGURE 4.3b

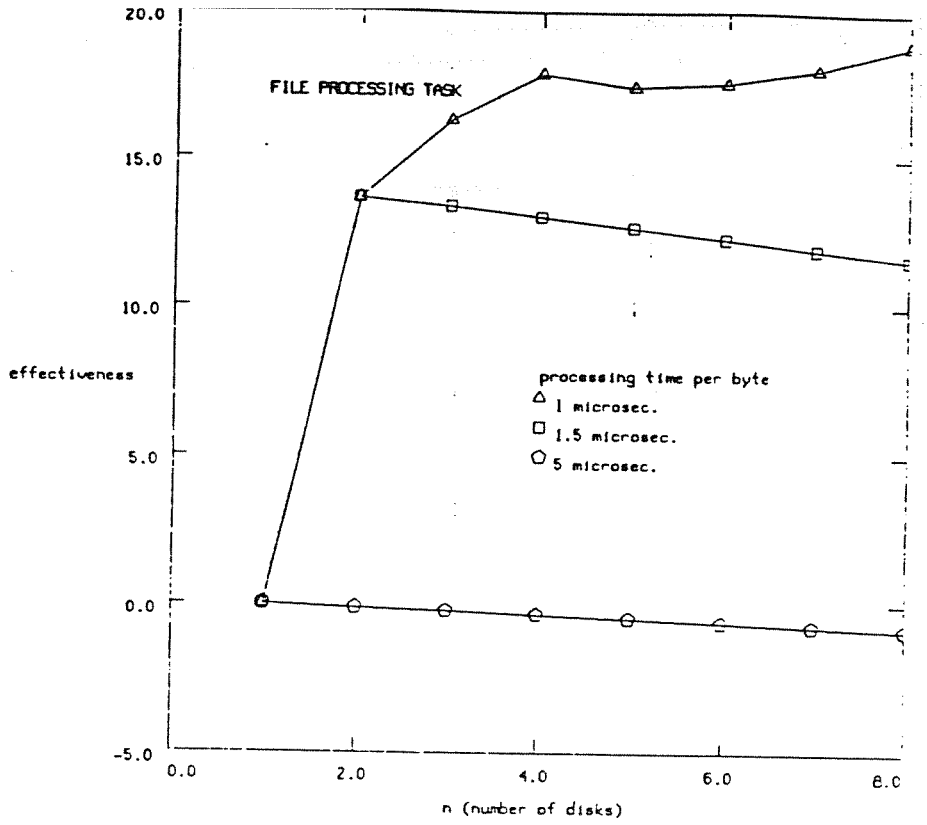


FIGURE 4.3c

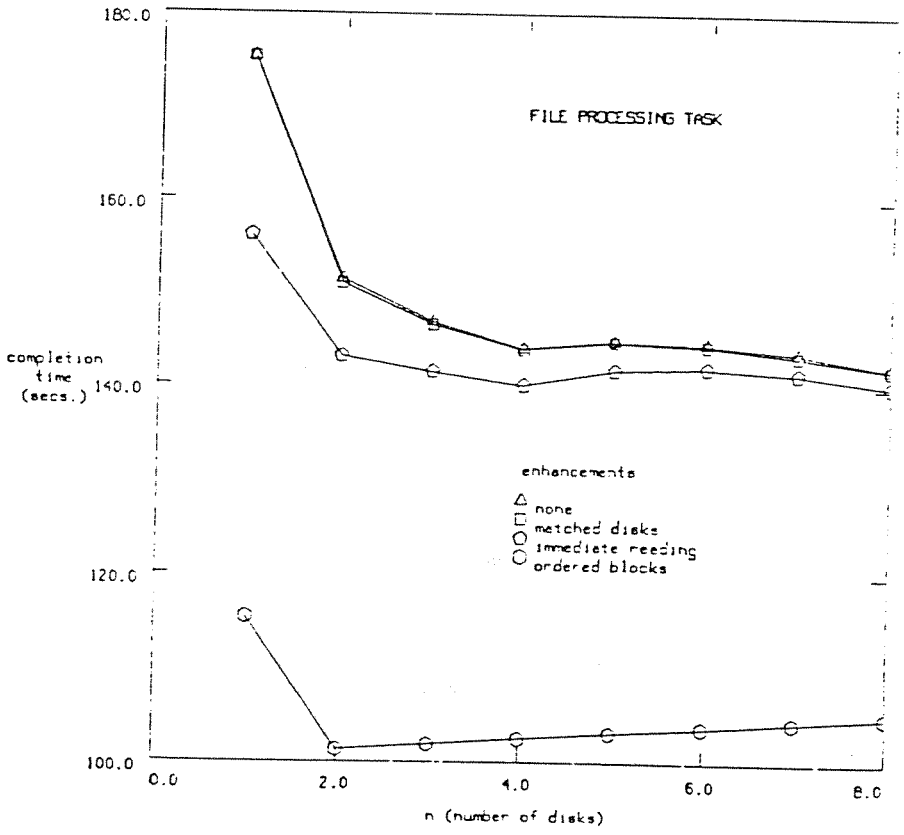


FIGURE 4.3d

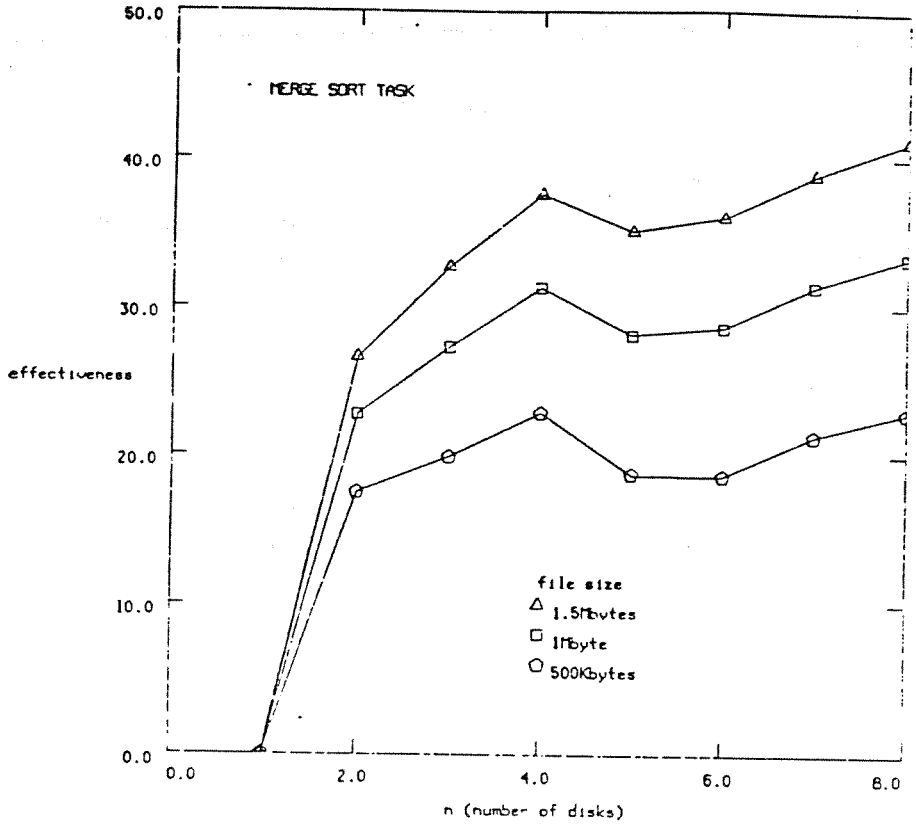


FIGURE 4.4a

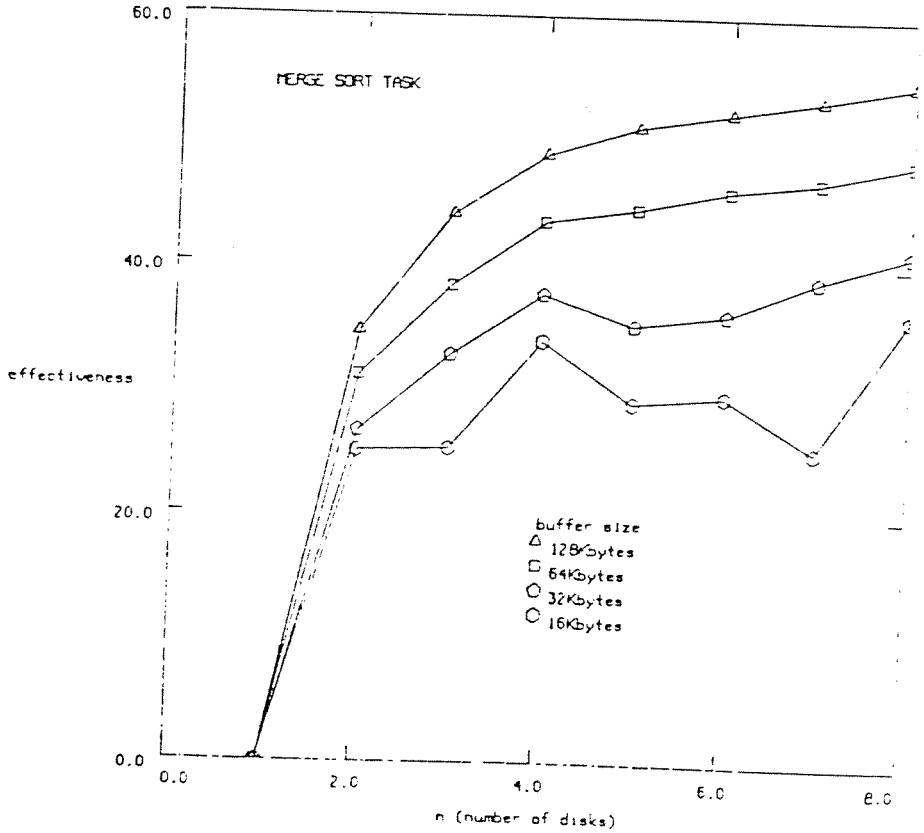


FIGURE 4.4b

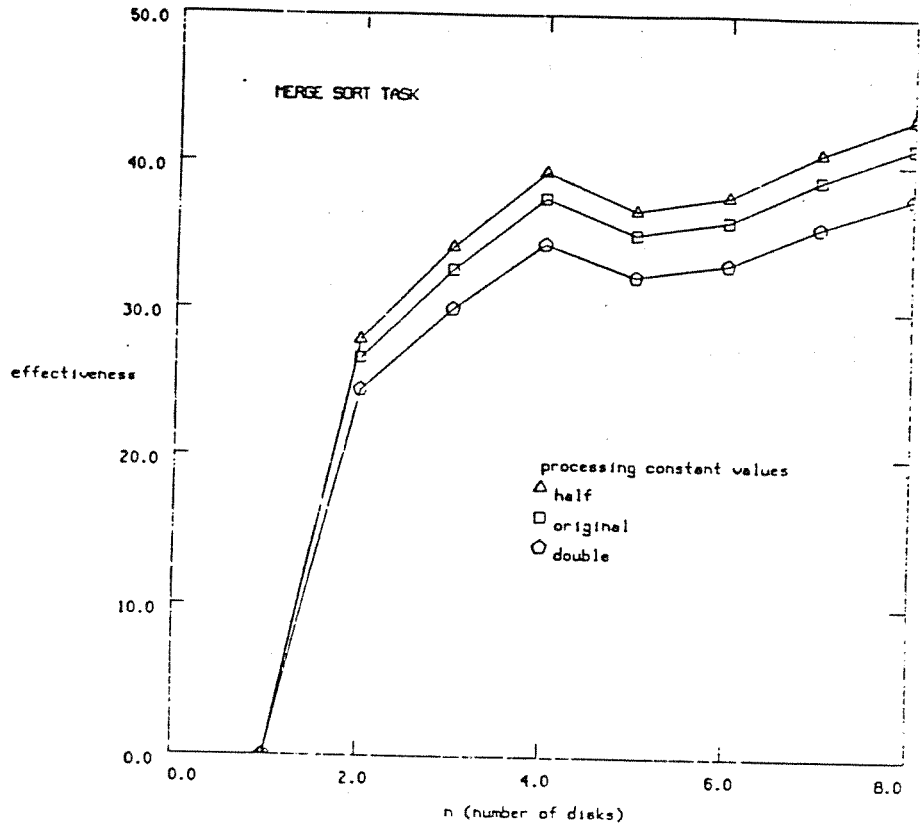


FIGURE 4.4c

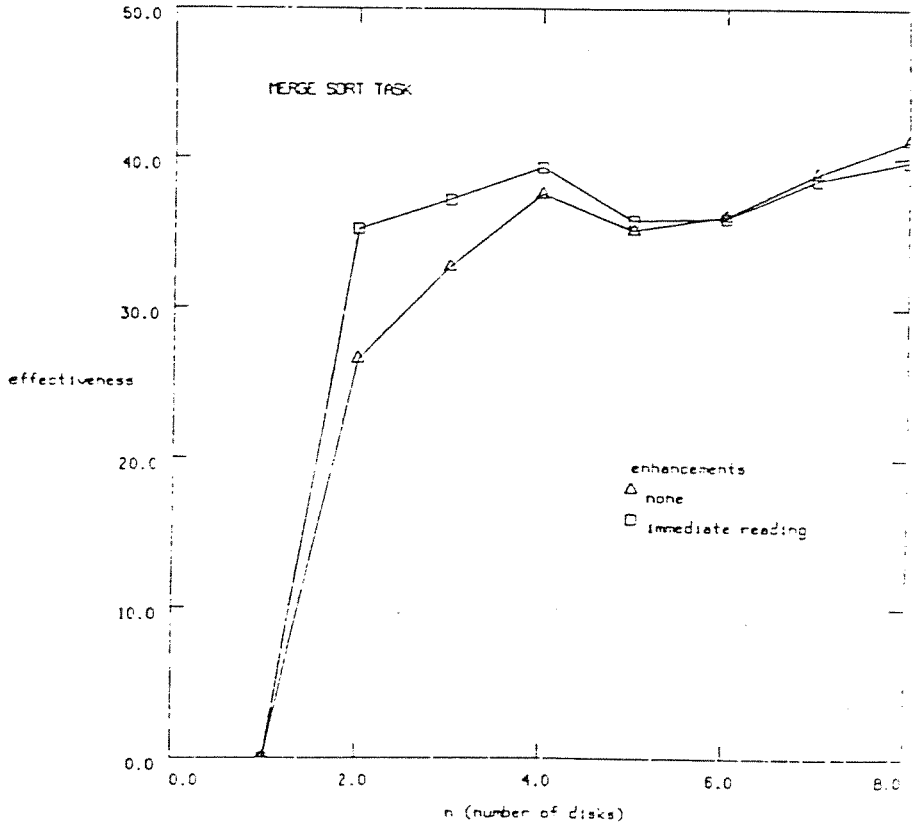


FIGURE 4.4d

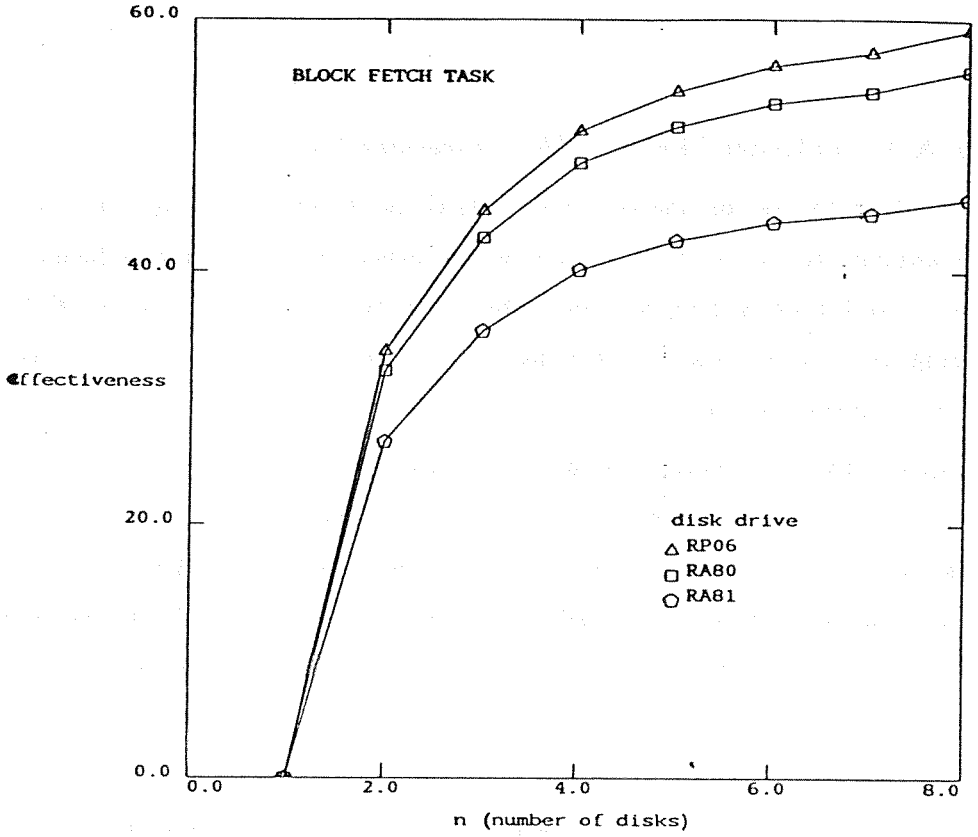


FIGURE 4.5

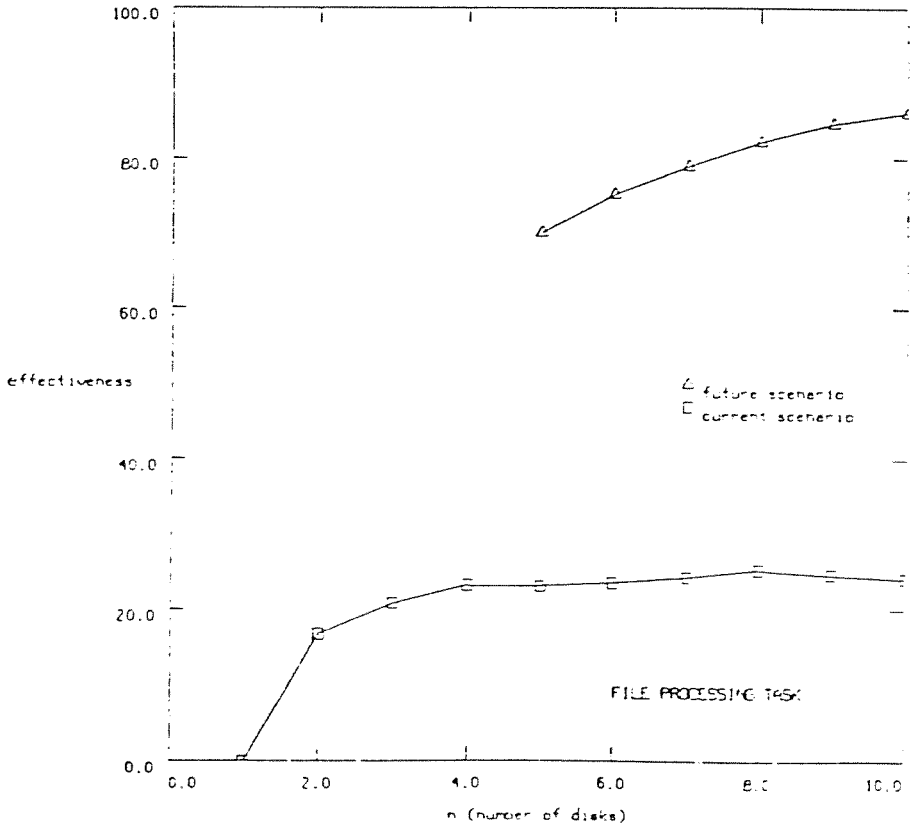


FIGURE 5

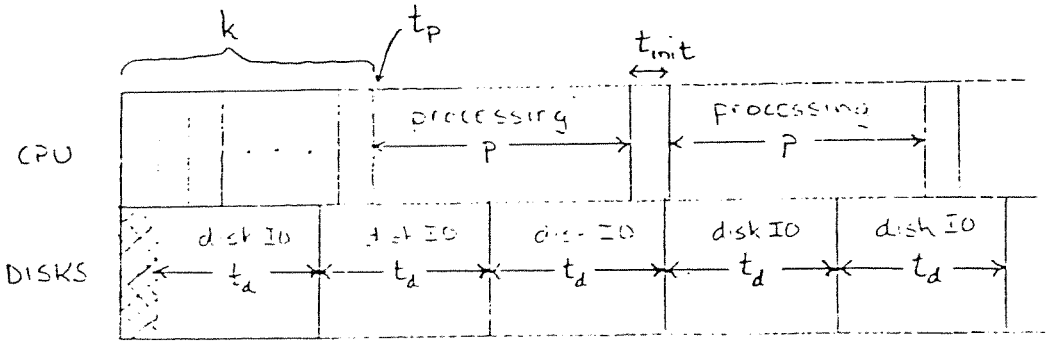
Appendix A: Completion Time for File Processing Task

This analysis focuses on the read-only version of the task. The modifications necessary to adapt this analysis to the read/write version are briefly discussed in conclusion. For the read-only version, the following diagrams describe the activity over time of the CPU and the disk drives during the initial phase of the task. (The initial and main phases of the task are described in Program 4.3 of the text.)

We suppose that the memory buffer is capable of holding k blocks of data. According to the task description, processing of the first block does begin until k blocks have been requested and the first of these blocks has arrived in the primary memory. The first diagram describes the case when the first block arrives before all k initial blocks have been requested. The second diagram describes the other possibility, that the initial requests are completed before the first block arrives.

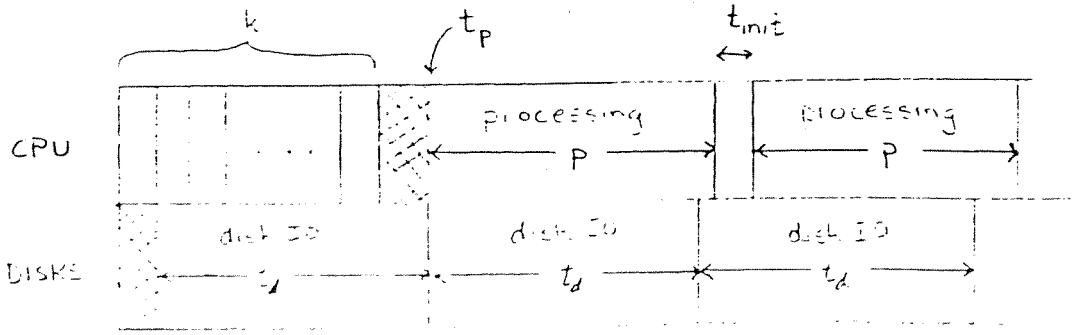
In both diagrams, shaded areas represent idle time. CPU overhead time for initializing block requests is represented by t_{init}, where $t_{init} = t_{ioc} + nt_i$. Processing of the first block begins at time t_p.

DIAGRAM 1



$t_p = k t_{init}$

DIAGRAM 2



$t_p = t_{init} + t_d$

Since disk IO and processing are overlapped for this task, we need to determine whether the task completion time is bound by the CPU or by the disks. To estimate which of these bounds applies, we examine the rates of change of the sizes of the disk queue and the list of unprocessed blocks available in memory and at the initial sizes of these lists at time t_p, when the main phase of the algorithm begins. The task is considered CPU-bound if the disk queue would empty before the blocks list and disk-bound if the opposite is true. To make this determination we consider the following variables:

r_{io} = net rate of change of size of disk queue
r_b = net rate of change of size of blocks list
a_{io} = size of disk queue at time t_p (initial condition)
a_b = size of blocks list at time t_p (initial condition)
z_{io} = time at which size of disk queue will be zero
z_b = time at which size of blocks list will be zero

The disk queue grows every time processing is completed on a block and shrinks every time a disk block transfer is completed. Similarly, the block list is shrinks with every processing completion and grows with disk block transfers. Using this information and the diagrams above, we can write:

$$t_p = max(kt_{init}, t_d + t_{init})$$

If $t_p = kt_{init}$ then

$$a_b = \frac{t_p - t_{init}}{t_d}$$

$$a_{io} = k - a_b$$

else $t_p = t_d + t_{init}$ and

$$a_b = 1.0$$

$$a_{io} = k - 1.0$$

Rates of change for the list and the queue are given by

$$r_b = \frac{1.0}{(t_d)} - \frac{1.0}{pb + t_{init}}$$

$$r_{io} = -r_b$$

and zero times are

$$z_b = \frac{-a_b}{r_b}$$

$$z_{io} = \frac{-a_{io}}{r_{io}}$$

In these equations p and b represent the processing constant and the block size, respectively. Also, $t_{init} = t_{ioc} + nt_i$ and $t_d = E[f^n_{(s+r)}(t)] + t_x$. These terms represent CPU time and disk time as described in the text.

To determine whether the task is disk-bound or CPU-bound we use the values of z_b and z_{io}. By inspection of the above equations, we see that z_b and z_{io} have opposite signs. A negative value implies that the corresponding queue will never be empty. If z_b is positive, we say the task is disk-bound since the blocks list empties first (the disks cannot keep up with the CPU's demands). Similarly, the task is CPU-bound if z_{io} is positive. Once the bound is known, the task completion time can be determined. If the task is disk-bound:

$$t_n = t_{init} + \frac{r_d}{b} t_d$$

otherwise

$$t_n = (t_{init} + pb)\frac{r_d}{b}$$

where r_d is the size of the database.

Completion time calculations proceed in a similar fashion for the read/write version of the task. The principal difference is that the rates of change of the two queues might both be negative since the CPU generates two IO requests for each block processed. In this case, both z_b and z_{io} will be positive. We choose the smallest of the two and make the bound determination as before.

Appendix B: Completion time for Merge Sort Task

The analysis of the merge sort task uses some basic assumptions about the location of records on the disks. As a result, seeks in the merge sort task cannot be considered random in

the same sense as before.

Two copies of the data file are used for the merge sort. Each alternately acts as the *read* copy or the *write* copy, with the alternation occurring between merging passes in the main phase of the algorithm. Each copy is assumed to occupy a contiguous set of cylinders, called its task data space, on each disk. These twin spaces are assumed to be physically adjacent.

During the first phase of the algorithm, the read copy of the data file is broken into block-sized pieces. Each of these pieces is sorted and written back to its original position in the task data space. Assuming that a block holds k records, this involves $2(r_d/k)$ disk transfers. However, these transfers can be accomplished with a single sweep of the disk heads across the read task data space. All seeks will be to an adjacent cylinder, and we need only do as many seeks as there are cylinders in the data space (call this c_{tds}). However, every transfer will involve rotational latency, transfer time, and CPU initialization. We can therefore express the total time for the first phase of the merge sort by:

$$t_{first} = c_{tds}t_{os} + 2\frac{r_d}{k}(E[f^n_{(r)}(t)] + t_x + t_{ioc} + nt_i + c_s k log(k))$$

Following the notation used in the paper, $E[f^n_{(r)}(t)]$ represents the expected rotational latency in an n-disk system. The remaining terms within the parentheses represent transfer time, CPU initialization time (two terms), and sorting time, respectively.

Once this initial sorting has been accomplished, the main phase of the algorithm involves repeatedly merging two runs from the read task data space into a single run of twice the original length, and writing the new run into the write task data space. This process is iterated until the entire database consists of a single run.

Each disk transfer in the main phase involves $k/2$ rather than k records since we must have pieces of two runs in memory to merge them. The locations of disk blocks to be transferred during the main phase are partially specified by the algorithm. We know which cylinders are occupied by the runs being merged, however the actual order in which blocks will be retrieved from and written to these cylinders depends on the data.

To get an approximation of the completion time for the second half of the task, seeks are broken down into two types. Seeks which involve a movement from one task data space to the other are called *long* seeks. Those in which the head remains in the same data space are *short*.

Short seeks are treated as random seeks over a number of cylinders only as large as the number of cylinders (call this c_r) occupied by the current runs being merged. We approximate

a long seek by making the disk head travel the distance between the read and write copies of the current runs, plus $c_r/2$ cylinders, plus the distance of a short seek. This is by no means exact determination of the distance covered during a long seek but it provides a reasonable estimate, especially when the runs are relatively small.

Merging two runs of y records each involves $8(y/k)$ disk transfers, since we transfer in blocks of $k/2$ records and each block is both read and written. These transfers alternate between reads and writes except for the first two which are both reads and the last two which are both writes. Therefore, in merging two runs of length y we get $8(y/k)-2$ long seeks and 2 short seeks.

With all of this information in hand we can compute the completion time for the main portion of the merge sort task. As was discussed in the text,

$$i_{\max} = \log_2\left(\frac{r_d}{k}\right)$$

passes are required to reduce the database to a single sorted run. During the ith pass, pairs of runs of length $2^{i-1}k$ will be merged to form runs of length $2^i k$. We let t^i_{long} and t^i_{short} represent the expected times for long and short seeks, respectively, during the ith pass.

For a single pair of runs being merged during the ith pass, the total seek time involved is given by

$$t_{seek}(i) = \left(8\left(\frac{2^{i-1}k}{k}\right)-2\right)t^i_{long} + 2t^i_{short} = (2^{i+2}-2)t^i_{long} + 2t^i_{short}$$

We can write the total rotational latency and transfer time for these runs as

$$t_{rot}(i) = 8\left(\frac{2^{i-1}k}{k}\right)\left(E[f^n_{(r)}(t)] + t_x(2^{i-1}k/2)\right)$$

which can be rewritten as

$$t_{rot}(i) = 2^{i+2}\left(E[f^n_{(r)}(t)] + t_x(2^{i-2}k)\right)$$

Finally, CPU time is spent initializing the transfers and merging the records once they are in memory. This time is represented by

$$t_{CPU}(i) = 2^{i+2}(t_{ioc} + nt_i + c_m 2^i k)$$

Combining the above equations we arrive at an expression for the total completion time for the

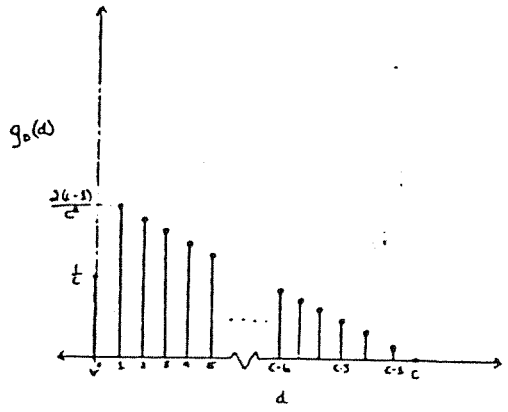


FIGURE C1

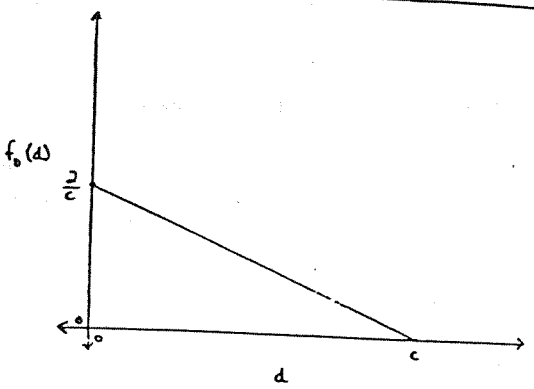


FIGURE C2

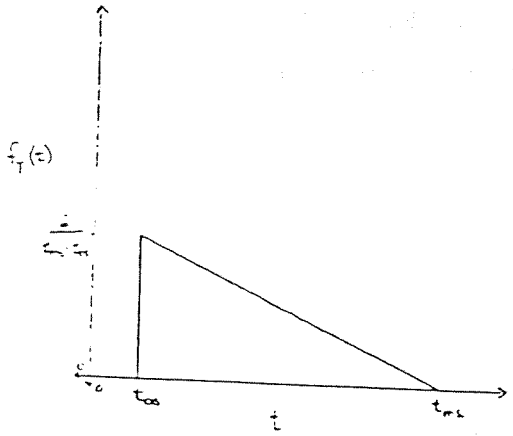


FIGURE C3

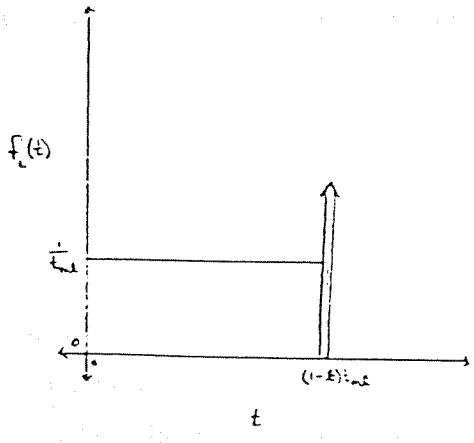


FIGURE D1

main phase of the merge sort

$$t_{main} = \sum_{i=1}^{i_{max}} \frac{r_d}{2^i k} \left(t_{seek}(i) + t_{rot}(i) + t_{CPU}(i) \right)$$

The completion time, t_n, for the entire task is then

$$t_n = t_{main} + t_{first}$$

the sum of the completion times for each phase.

Appendix C: Seek Time Probablility Density

The key to the developement of this density function for seek time is an assumption that given a sequence of blocks to be retrieved consecutively from a disk, each of the blocks has a uniform probability of residing on any cylinder. With this assumption, we develope a density function for a random variable D, the length (in terms of the number of cylinders traversed) of a head seek. Seek time is then expressed as a function of D.

Wiederhold, in [WIED77], developes a similar distribution for D. His distribution expresses the probabilty of traversing d cylinders *given that the disk head is going to move.* The resulting distribution, for a disk with c cylinders, is

$$g_D(d) = 2\left(\frac{c-d}{c(c-1)} \right)$$

for $1 \leq d \leq (c-1)$.

The principal difference between this expression and our own is that we wished to include the possibilty that the disk head might not have to move at all. Such a situation would occur if two consecutively read subblocks resided on the same cylinder. Our distribution can then be thought of as expressing the probability of having to traverse d cylinders before accessing the next subblock, given our initial assumption. The derivation of our density function proceeds as does Wiederhold's, except that there are c^2 rather than $c(c-1)$ possible pairs of cylinders we might have to travel between. We therefore arrive at the expression

$$g_D(d) = 2\left(\frac{c-d}{c^2} \right)$$

for $1 \leq d \leq (c-1)$. We also have that

$$g_D(0) = \frac{1}{c}$$

The function $g_D(d)$ is graphed in Figure C1 for $c = 5$.

To simplify calculations, we approximate function g_D by a continuous density function f_D where

$$f_D(d) = \frac{-2}{c^2} d + \frac{2}{c}$$

for $0 \leq d \leq (c-1)$. $f_D(d)$ is shown in Figure C2 for $c = 5$.

Next, we express seek time, T, in terms of our random variable D. Again following Wiederhold, we appoximate the time vs. distance characteristic of a disk drive by

$$T = \alpha D + \beta$$

where the constants α and β are determined from a disk's specification sheet. Specifically, if t_{os} represents the *single cylinder* seek time (time to move the disk head to an adjacent cylinder) and t_{ms} represents the maximum seek time (time to move the disk head across c cylinders) then we set

$$\alpha = \frac{t_{ms} - t_{os}}{c}$$

and

$$\beta = t_{os}$$

Finally, we can get the probability density function for T using

$$f_T(t) = \frac{d}{dt}\left[F_T(t)\right] = \frac{d}{dt} F_D\left(\frac{ct - ct_{os}}{t_{ms} - t_{os}}\right) = \frac{c}{t_{ms} - t_{os}} f_D\left(\frac{ct - ct_{os}}{t_{ms} - t_{os}}\right)$$

where functions F_T and F_D represent the cumulative distribution functions of T and D, respectively. Substituting, we arrive at

$$f_T(t) = \frac{-2}{(t_{ms} - t_{os})^2} t + \frac{2}{(t_{ms} - t_{os})^2} t_{os} + \frac{2}{t_{ms} - t_{os}}$$

for $t_{os} \leq t \leq t_{ms}$, which is shown graphically in Figure C3.

Appendix D: Rotational Latency with Immediate Reading

To facilitate this discussion, we define rotational latency as all of the *wasted* time between the completion of a head seek (if one is necessary) and the completion of the data transfer from the disk. Wasted time is time during which no data transfer is taking place. Without immediate reading, this corresponds to the time we wait for the beginning of the subblock to appear under the disk head since no data transfer happens until this event occurs. We assume that this time is uniformly distributed on $(0,t_{ml})$ since we do not know the angular postion of the disk at the time of the head's arrival.

We assume the the fraction of a disk track occupied by a subblock is given by b, where $0 \leq b \leq 1$ and consider immediate reading. To determine rotational latency we need to look at two possible situations. When the disk head arrives at the target cylinder, either the subblock is underneath it or it is not. If the subblock is underneath, that part of the data which has not passed under the head can be read immediately. We then must waste time $b(1 - t_{ml})$ before the beginning of the block appears and the rest of the block can be read. If the subblock is not underneath, we must wait for the beginning of the subblock to appear, at which point the data can be read exactly as if the disk did not have immediate reading capability.

Assuming that the disk head has equal probability of appearing at any angular position, the probability that it will appear over the subblock is just b, the relative size of the subblock. We can therefore write $f_L(t)$, the probability density function of rotational latency L, as the sum of two functions, each representing one of the two situations described above.

$$f_L(t) = f_{L1}(t) + f_{L2}(t)$$

$$f_{L1}(t) = \frac{1}{t_{ml}} \qquad 0 \leq t \leq (1-b)t_{ml}$$

$$f_{L2}(t) = b\,\delta((1-b)t_{ml})$$

Here $\delta(x)$ is a delta function of unit area at x. The function f_{L1} represents the situation in which the disk head does not arrive over the subblock while f_{L2} represents the other. Note that if the head arrives over the subblock (with probability b) we always have a rotational latency of $(1-b)t_{ml}$. The distribution $f_L(t)$ is shown in Figure D1.

Using the immediate read distribution, the expected value of rotational latency (for one disk) is given by

$$E[L] = \left(\frac{1}{2} - \frac{b^2}{2}\right)t_{ml}$$

which compares favorably with

$$E[L] = \frac{t_{ml}}{2}$$

the expected value of rotational latency without immediate reading.