Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis

Vinay S. Pai *
Peter J. Varman†
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251

Abstract

With the increase in the size of main memory in computer systems, multiple disks and aggressive prefetching can be employed to significantly reduce I/O time. Two prefetching strategies - intra-run and inter-run - for external merging using multiple disks are studied. Their performance is evaluated using simulation, and simple analytical expressions are derived to explain their asymptotic behavior. The results indicate that a combination of the strategies can result in a significant reduction in I/O time.

1 Introduction

Advances in processor architecture have resulted in steady increases in processor speeds over the past several years. The performance of I/O subsystems, in contrast, has generally not kept pace with these improvements in processor performance. The mismatch between the speeds of the two subsystems is exacerbated by parallel processing technology that allows orders-of-magnitude increase in processing speeds by coupling several processors together to work on parts of a single problem. The data rates possible from single disks are limited by physical considerations such as the speed of disk rotation and the rate of head movement, and are unlikely to increase dramatically. As a consequence, there have been a number of recent proposals for the use of multiple disks to form high-performance I/O subsystems [9] [17] [19]. Performance evaluation of different multiple-disk systems, and associated management strategies have been studied in [20] [14] [18] [6] [7] [12], for example.

External sorting of large files is important for a number of applications in database systems, and has been extensively studied for several decades [11]. Mergesort is one well known algorithm that is used for external sorting [5]. In this algorithm, a number of sorted runs are first created by individually sorting one memory-load of data at a time, and writing each run out to external disk storage. These sorted runs are then merged together in a small number of merge passes, using a multiway merge algorithm. Due to the extensive use of secondary storage for storing temporary runs, the performance of external mergesort greatly depends on the management of the I/O.

This paper presents a simulation study of multiple disk systems to improve the I/O performance of multiway merging. With the growing amounts of main memory in computer systems, multiple disks and aggressive prefetching can be employed to significantly reduce the I/O time. We study two different *prefetching* strategies for external merging using simulation, and present simple analytical models to analyze the asymptotic speedups in the I/O time that can be realized. The results show that the introduction of concurrency in disk operations by prefetching, combined with a reduction in the average seek time and rotational latency, can result in superlinear speedups over straightforward single-disk I/O time.

A number of papers in recent years have dealt specifically with the I/O performance of the merge phase of mergesort [5], [13], [21], [1], [8]. These are summarized below, to place the study reported in this paper in perspective. The models in [13], and [21] assume that all runs that are input to the merge reside on a *single* disk. In each run the data are stored in fixed-size *blocks*; a block is the basic unit of transfer between the disk and main memory. Kwan and Baer [13] assume a random model of block depletion, in which a block from any of the runs is depleted with equal probability. They derive the average I/O time for accessing a block in this model and use that in estimating the total I/O cost of the merge. The costs they consider are seek time, rotational latency, and transfer time. The model in [21] assumes that the memory capacity is sufficient to fetch large-sized blocks on every access, so that the seek and rotational latency are dominated by the transfer time. In such a model, the optimality of mergesort as an external sorting procedure is established. Both these papers concentrate on the input data stored on a single disk, and do not consider the speedup potentially available with multiple, concurrently accessible disks for the input.

Aggarwal and Vitter [1] show that mergesort is an optimal external sorting method (up to a constant factor) in the total *number* of I/O's required. They also consider the possibility of using D disks to obtain con-

*Partially supported by an NSF Graduate Research Fellowship.

†Partially supported by NSF and DARPA under Grant CCR 9006300.

currency in the I/O, and describe a method to predict which D blocks to prefetch on an I/O operation. They do not, however, consider the possibility of contention for the disks, since their model assumes that *any* set of D blocks can be fetched in one parallel operation. The model in [8] is similar in that it assumes multiple disks for the input runs, and that the contention for the disks is negligible. Like [21] they assume that the I/O time for a block is dominated by the transfer time. All of these works, as well as the study described in this paper assume a separate set of disk(s) for writing the sorted output of the merge, thereby eliminating contention between the read and write traffic.

For other works on the I/O performance of different database algorithms like transitive closure the reader is referred to [2] and [22]. In studying the performance of multiple disks for general workloads, Salem and García–Molina [20] proposed declustered arrangements that interleave data at the sector level and showed that response times were improved for large block transfers. Livny, Khoshafian, and Boral [14] compare the performance of clustered and declustered file arrangements for uniform and nonuniform workloads characterized by normally-distributed file access patterns. Reddy and Bannerjee [18] consider hybrid systems composed of several declustered units in which each unit is possibly a synchronized disk array, and use simulation to evaluate their performance for characteristic transactional and scientific workloads. Kotz and Ellis [12] have investigated prefetching in a parallel environment by categorizing typical access patterns and by experimentally studying the effect on execution time.

The rest of the paper is organized as follows. In section 2 we describe the system model, and describe the different prefetching strategies. In section 2.1 we detail the parameters used in this study, and describe the simulation model in section 2.2. Section 3 contains the results of the simulation experiments and their analyses. We conclude with section 4.

2 Overview of Prefetching Strategies

In this section the system model used in our study of external merging is introduced. We then describe two prefetching strategies for reducing the I/O cost of merging from multiple input disks. Analysis and simulation of the performance of the strategies are discussed in section 3.

We study the problem of merging k sorted runs using D independent disk units storing the input. Figure 2.1 shows the system architecture. The system model consists of a CPU, a RAM-based disk cache with a capacity of C blocks, and D disks containing a number of sorted runs each [15]. The disks operate independent of each other; the bandwidth of the channel between the I/O subsystem and main memory is assumed to be sufficiently large to support concurrent data transfer from all D disks. The output of the merge is assumed to be written out to a separate set of disks. To focus on the benefits of prefetching, the write traffic will not be considered in this study.

A general operation of the mergesort algorithm is as follows. A block from each run is brought into memory, and the records from each block are extracted and merged together in sorted order. When one of the input blocks is depleted, an I/O request is made for the next block from the run whose block was depleted. We refer to this block as the *demand-fetch* block, and the run in which the block occurs as the *demand-fetch* run. Once the demand-fetch block has been retrieved from disk, the merging continues.

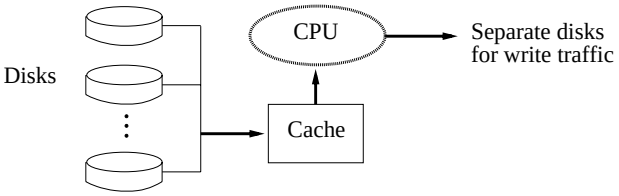

Figure 2.1: The system model consists of D disks, a RAM cache, and an infinitely fast CPU. Separate I/O subsystems exist for read and write traffic.

The I/O performance of this basic strategy using a single-disk system was studied by Kwan and Baer [13]. A random model of block depletion was used in the analysis : any one of the k blocks is chosen as the next block to be depleted with equal probability. A similar data model is assumed in this study. However, the data is assumed to be distributed among multiple disks. We study how prefetching can be used to reduce the total I/O time, and compare these using the single-disk results of [13] as the baseline.

Distributing the runs among several disks can reduce the total time for I/O due to several factors. We analyze the relative effects of the different factors in section 3. Since there are fewer runs on any single disk, the average seek distance for any single block is reduced, leading to a reduction in the seek component of the average I/O time. The I/O time can be further reduced by overlapping the requests at different disks, thereby introducing concurrency in their operation. To exploit the potential concurrency, blocks need to be *prefetched* and buffered in the cache until they are required, based on a cache management policy.

Two prefetching strategies are studied in this paper. We will refer to these as *intra-run* prefetching and *inter-run* prefetching respectively. Intra-run prefetching is a well-known method employed in single-disk systems to amortize the seek time and rotational delay associated with an I/O request [8]. In intra-run prefetching, whenever an I/O request for a block of data from a run is made, $N > 1$ contiguous blocks are read into memory. The first block that is read (the demand-fetch block) is used for the merge, and the remaining $N - 1$ blocks are held in the cache until they are required. For sufficiently large values of N, the seek time and rotational latency components of the total I/O time will be dominated by the transfer time. The optimal value of N depends on the disks mechanical parameters; larger values of N require a corresponding increase in the size of the cache.

When multiple disks are used for the input, intra-run prefetching can result in additional reduction in

I/O time. Two situations need to be differentiated. In a *synchronized* input operation, the CPU makes a request for the N blocks (1 demand fetch block and $N-1$ prefetch blocks) and waits until all N blocks have been read, before proceeding. In an *unsynchronized* input operation, the CPU resumes merging as soon as the demand-fetch block is available; the transfer of the remaining $N-1$ blocks overlaps with the operation of the CPU, until either all $N-1$ blocks have been read or the CPU makes an I/O request for blocks from a different run. In a single-disk system both synchronized and unsynchronized input have essentially the *same* total *I/O* time. If the blocks from run i are being prefetched, and the CPU makes a request for a block from a different run j, the latter request will be queued at the disk and serviced only after the current request is completed. The only difference between the synchronized and unsynchronized cases in this situation, will be the possible reduction in total time due to overlap of the CPU and I/O operations.

In a multiple-disk situation unsynchronized input can reduce the I/O time as well, due to the possibility of concurrent operations at different disks. In the situation mentioned above, if run j resides on a different disk from run i, the two I/O requests can proceed concurrently. This overlapped operation of possibly several disks will result in a reduction of total I/O time, over that arising due to reduction in seek time alone. We estimate the asymptotic concurrency in section 3.2.

In *inter-run* prefetching, whenever a request is made for a demand-fetch block from a particular run, an I/O request is made to each of the other disks as well; these requests prefetch blocks from the disks not containing the demand-fetch run. Thus, concurrency in the disk operations is explicitly introduced by prefetching blocks which have not yet been requested from the other disks. These blocks are held in the cache until they are required. Unlike intra-run prefetching where the size of the cache and the merge order fixes N (the number of blocks prefetched), in inter-run prefetching the cache neeeds to be managed dynamically.

Several issues regarding the management of the cache need to be addressed. The first question that arises is the choice of the run from which to prefetch. In the absence of any information, the natural choice is a *random* one - *i.e.* any one of the runs on the disk is chosen with equal probability. This policy is adopted in this study. In [15], other policies based on maintaining information about the head positions on each disk were also studied. The performance benefits of these heuristics over the random strategy were insufficient to warrant maintaining the detailed information needed for their implementation.

A second question deals with the policy to be followed when the cache does not have sufficient space to buffer all the blocks we would like to prefetch. One alternative is to prefetch only as many blocks as can be buffered in the available space in the cache; the choice of which of the blocks to prefetch can be made randomly. A second alternative is to *not prefetch* any blocks from the other disks if the cache cannot accommodate all these blocks. In this case only the demand-fetch block is read. When a sufficient number of cache blocks have been freed by depletions, prefetching from all disks is resumed. The second alternative is the one used in this study. The reason for this choice is based on the Markov analysis reported in [16]. In that paper we consider both alternatives for handling an almost-full cache, for the case of D disks with one run per disk. We show analytically that the average I/O parallelism obtained by the second alternative is *superior* to making a random choice, for all reasonable values of cache size and number of disks. We conjecture that the same effect holds true in this situation where each disk has several runs stored on it. A complete analysis is however, beyond the scope of this paper. Intuitively using the first alternative (a greedy policy), by filling up the cache we delay the chances of returning to a state where all D disks can be used concurrently. The average I/O parallelism obtained by the larger number of partial fetches using the greedy policy is less than that of the second policy, where we sacrifice partial concurrency so as to free up cache space quicker.

Finally we note that intra-run and inter-run prefetching are not mutually exclusive, and can be combined. That is, whenever a demand-fetch block is required, $N-1$ blocks from that run are prefetched, and N blocks from each of the other disks are concurrently prefetched as well, subject to the availability of free cache space. Our simulation results indicate that this combination of inter-run and intra-run prefetching leads to the best I/O time for reasonable cache sizes.

2.1 Performance Measures

Three components of the I/O cost will be considered in this study. These three components of the disk's access time — seek time, rotational latency, and transfer time — depend on the mechanical parameters of the disk/head assembly. To aid in the analysis, the following quantities are defined:

$\mathcal{S}$: The seek time per cylinder.

$\mathcal{R}$: The average rotational latency. This quantity is chosen to be half of the time taken for one full revolution of the platters [13].

$\mathcal{T}$: The transfer time per block is exactly $\mathcal{T}$.

m : The length of each run in cylinders.

N : The number of blocks fetched from each run on each access.

k : The total number of runs. For simplicity in the subsequent analysis, k will be assumed to be a multiple of D. If not, $\lceil \frac{k}{D} \rceil$ should be substituted for k/D.

π : The average time to fetch one block. This quantity is the sum of the average seek time, average rotational latency, and transfer time.

Two performance measures will be employed in the evaluation. The first is the *total time* for the merge using a particular prefetching strategy. The second, which indicates the sensitivity of the degree of prefetching to the cache size, is the *success ratio*.

This statistic is the probability that a given prefetch can be initiated because the cache has enough room to accommodate all blocks. This measure is only meaningful for the inter-run prefetching strategy, since the number of blocks that are prefetched varies dynamically. In intra-run prefetching, for a given cache size and merge order, N is automatically fixed to obtain a success ratio of 1. To focus on the I/O performance, most of the experiments are performed assuming an infinite-speed CPU. The effect of differing CPU speeds is then studied in a separate set of experiments.

The seek time for an I/O request will be the product of the seek distance and $\mathcal{S}$. While it is known that such a a linear relationship overestimates the seek penalty, it has been found to be sufficiently useful to justify its simplicity [13].

The disk model [3] used by the simulator has the following parameters: $\mathcal{T} = 2.048$ ms, $\mathcal{R} = 8.333$ ms, and $\mathcal{S} = 0.04$ ms/cylinder, 14 heads, 52 sectors/track, 1258 tracks/disk, and 512 bytes/sector. On each access a 4096-byte block is fetched. To simulate this larger block size, the cylinder size is maintained by modeling 7 heads, 13 sectors/track, and a 4096-byte sector.

2.2 Simulation Model

The Rice C Simulation Package (CSIM) [4] is used to construct a process-oriented simulation model of the *merge phase* of external mergesort. The simulation model consists of a CPU, a cache, a disk subsystem, and blocks of data. No data records are actually maintained, since the merge will follow the *block-depletion model*, in which the next block is chosen at random. At each step, a run which still contains unmerged blocks of data is chosen at random; a block will now be depleted from this run. The next block from that depleted run is the demand-fetch block. If the demand-fetch block exists in the cache, it is retrieved from the cache without any I/O operations; else an I/O request must be initiated. If the cache contains enough room for prefetches, a request for these prefetches will be initiated along with the request for the demand fetch; otherwise, only the request for the demand-fetch block is issued.

Each request for a block — demand fetch or prefetch — will be queued at the appropriate disk as an individual request. A separate process is allocated for each request, and this process suspends while that block is in the appropriate disk queue. In this manner, each block can be serviced independently. Since each block is treated separately, prefetches can be processed concurrently while the CPU continues with the merge. In order to synchronize prefetches, a mechanism which allows the CPU to wait on a prefetch is implemented. This general framework for the I/O provides the generality necessary to simulate several prefetching strategies.

The results of 8 trials are averaged to produce each graph. Each run is comprised of exactly 1000 blocks and occupies $m = \frac{1000}{91} \simeq 10.9890$ cylinders. Each disk block is 4096 bytes in length and holds 64 records. The total data size consists of 1.6 million records when $k = 25$, and 3.2 million records when $k = 50$. The two prefetching strategies, discussed in Section 2 were simulated with two disk configurations— $D = 5$ and $D = 10$— and several data sizes— $k = 25$, $k = 50$ runs and $k = 100$ runs. For reasons of space, the results for $k = 100$ are not presented here.

3 Simulation Results and Analysis

In this section we discuss each of the prefetching strategies in detail. Simulation results for the various cases are presented. Simple analytical formulae to predict the performance, at least for asymptotic values of the parameters, are derived. These serve to validate the simulation, and describe the limiting behavior of the various strategies. We describe the results for a single disk system in section 3.1, and for multiple disks in section 3.2. In each case, we first analyze the case of no prefetching, followed by the appropriate prefetching strategies. When relevant, we distinguish between synchronized and unsynchronized prefetching as defined in section 2.

3.1 Results for a Single Disk

The first model that we study is that of a single disk, beginning with the case of no prefetching, followed by that using intra-run prefetching.

No Prefetching, Single Disk: In this model, introduced by Kwan and Baer [13], k runs are placed contiguously on a single disk, and the cache consists of k blocks, one for each run. When a memory-resident block of a run is depleted, the next block of that run is fetched from the disk. The block to be depleted is chosen randomly from the k runs with equal probability.

For this model, Kwan and Baer [13] determine the average seek distance. To extend the analysis to handle the case of multiple disks the probability distribution governing the seek distance is explicitly derived below. The derivation closely follows the analysis of [13] for the average seek distance.

Let the runs be numbered from 1 through k. Let $\mathbf{x}$ be a random variable that represents the number of moves (seek between adjacent runs) made on an I/O request. Let $\mathcal{P}(\mathbf{x} = i)$ denote the probability that $\mathbf{x}$ has the value i, $0 \leq i \leq k-1$. Then:
P($\mathbf{x} = 0$) = 1/k
For $1 \leq i \leq k-1$: P($\mathbf{x}$ = i) = $2/k - 2i/k^2$
The expected number of moves, $E(\mathbf{x})$, is therefore given by:

$$E(\mathbf{x}) = \sum_{i=0}^{k-1} i\ \mathcal{P}(\mathbf{x} = i) = k/3 - 1/3k$$

The expected number of moves can be approximated by $k/3$ with negligible error. Since a run occupies m consecutive cylinders, and the average number of moves is $k/3$ runs, the average time to access a block is given by [13]:

$$\pi \quad = \quad m\left(\frac{k}{3}\right)\mathcal{S} + \mathcal{R} + \mathcal{T} \qquad (1)$$

With only one input disk there is no overlap of different I/O operations. For an infinitely fast CPU the total execution time is just the total I/O time, which is the product of π, the average I/O time for a block, and the total number of blocks. Since there are k runs, and each run is chosen to contain exactly 1000 blocks, the total execution time is given by $1000k\pi$.

Substituting the values in equation 1, we obtain for $k = 25$, $\pi = 14.03$ms, and for $k = 50$, $\pi = 17.70$ms. These evaluate to a total time of 351.1 and 885.3 seconds respectively. The corresponding numbers obtained by simulation are 351.03 and 884.06 seconds respectively, as seen in Figures 3.2 (a) and (b) (Demand Run Only, 1 disk) with $N = 1$.

Intra-run Prefetching, Single Disk: To reduce the I/O time using a single disk, *intra-run* prefetching can be used. In this strategy N blocks are read from the demand-fetch run on each I/O operation. This reduces the average time to access a block by amortizing the initial seek time and rotational latency over N consecutive blocks.

In order to fetch N blocks from the demand run on every fetch, a cache with a capacity of kN blocks is necessary and sufficient. When N blocks have been fetched from each of the k runs, the cache will contain kN blocks. All subsequent depletions can continue without requiring a fetch until some run no longer has any cached blocks. At least N blocks must have been depleted before this fetch is required, and hence, the fetch of N blocks can be accommodated.

The average I/O time using intra-run prefetching for a single disk is given by:

$$\pi = m\left(\frac{k}{3N}\right)\mathcal{S} + \frac{1}{N}\mathcal{R} + \mathcal{T} \qquad (2)$$

As expected, as N increases, the average I/O time per block tends asymptotically to the transfer time. The graphs in Figures 3.2 (a) and (b) (Demand Run Only, 1 disk), show the reduction in time as N, the number of blocks prefetched is increased, for the case of 25 and 50 runs respectively. The simulation corresponds well with the formula derived above. For example, with $N = 10$ and $k = 25$, the estimated and simulated I/O times are 81.19 seconds and 81.55 seconds respectively. Similarly, for $k = 50$ and $N = 10$, the estimated and simulated I/O times of 180.69 and 181.78 seconds closely match each other. The asymptotic I/O time in the case of large N is the product of the transfer time $\mathcal{T}$ and the number of data blocks. This is a lower bound on the I/O time for a single disk. For $k = 25$ and $k = 50$, these evaluate to 51.2 and 102.4 seconds respectively. For the largest value of N simulated, $N = 30$, the simulated (estimated) times are 61.8 (61.2) and 129.6 (128.5) seconds respectively, indicating that the asymptote has not yet been attained.

3.2 Multiple-Disk Results

In the study of multiple input disks, we will assume that the k runs are equally distributed over D disks. We first discuss the case without prefetching, followed

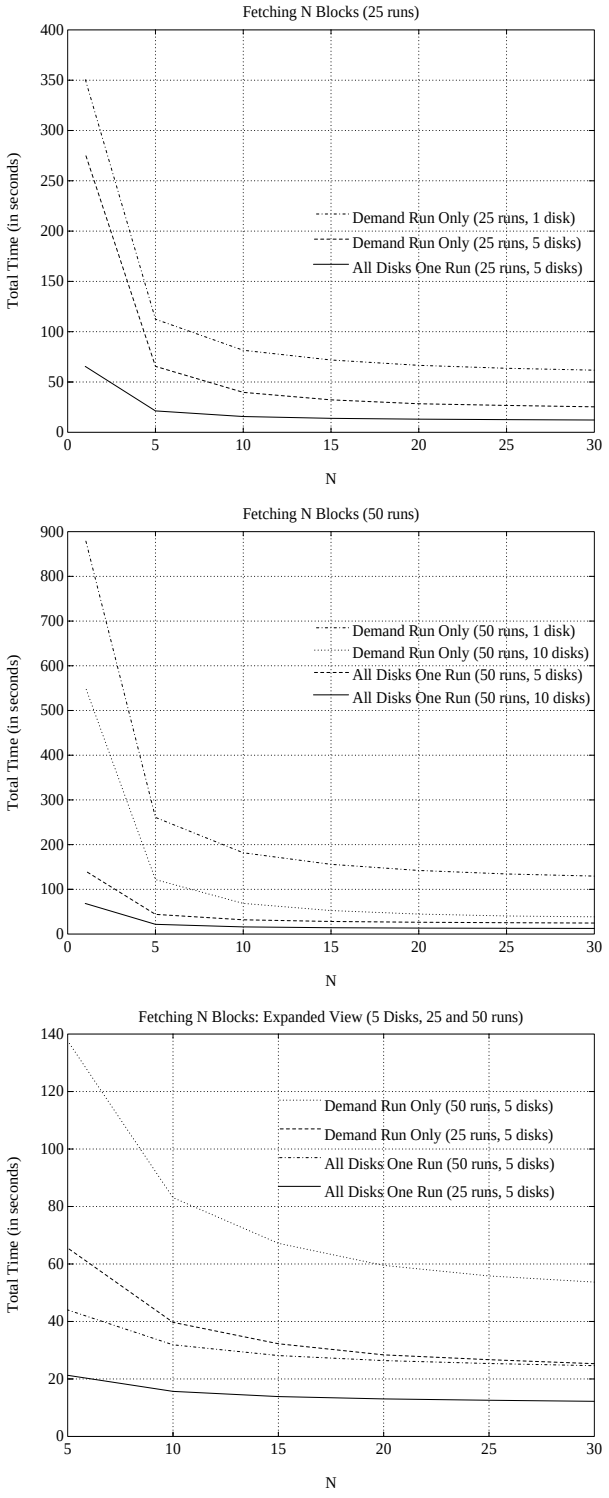

Figure 3.2: The effect of fetching N blocks from each run for $k = 25$ and $k = 50$ runs (1000 blocks/run). Unsynchronized prefetching is used.

by discussions of intra-run prefetching and inter-run prefetching respectively.

No Prefetching, Multiple Disks: When multiple disks are used for the input runs, the most basic strategy, analogous to the method of Kwan and Baer for single disks [13], fetches only the demand-fetch block on an I/O operation. Even without any prefetching, the I/O time is reduced over the single-disk case, due to the reduction in the average seek distance on each disk.

It is easy to show that the sequence of requests to any disk is random, and hence the single-disk analysis holds for each disk. Since each disk contains k/D runs, the average seek will require $k/3D$ moves. The rotational latency, however, is not affected. Therefore, the average time to fetch one block is given by:

$$\pi = m\left(\frac{k}{3D}\right)\mathcal{S} + \mathcal{R} + \mathcal{T} \qquad (3)$$

The cache requirements are exactly the same as that for the corresponding single disk case – one block for each of the k runs. The formula closely predicts the simulation results. The estimated time using the equation above for $k = 25$, $D = 5$ is 277.8 seconds, and for $k = 50$, $D = 10$ is 555.7 seconds. Corresponding simulation times are 277.0 and 555.5 seconds respectively (see Figures 3.2 (a) and (b)). In Figure 3.2 (a) (Demand Run Only, 1 disk and 5 disks) with $N = 1$, the total time has reduced from 351.0 to 277.0 seconds for 25 runs. Similarly in Figure 3.2 (b), the time reduces from 884.6 to 555.5 seconds when distributing the 50 runs among the 10 disks.

Intra-Run Prefetching, Multiple Disks: When intra-run prefetching is employed with multiple disks, we need to distinguish between the two cases of synchronized and unsynchronized prefetching.

In the *synchronized* case, $N > 1$ contiguous blocks are fetched from the demand-run on a request for the demand-fetch block. The CPU does not proceed with merging until all blocks have been read into the cache. Thus, there will be no overlap of different I/O requests at the different disks. However, the amortization in seek time and rotational latency over the prefetched blocks in the demand run will result in a reduction of the average I/O time. Noting that there are k/D runs per disk, the average time to fetch one block is given by:

$$\pi = m\left(\frac{k}{3ND}\right)\mathcal{S} + \frac{1}{N}\mathcal{R} + \mathcal{T} \qquad (4)$$

For the synchronized case, the total time is the product of π and the total number of data blocks, since there is no overlap in the operation of the disks. For the case of $k = 25$, using 5 disks and prefetching $N = 10$ blocks on every read, the I/O time using the above formula evaluates to 73.9 seconds. This is supported by the simulation results shown in Figure 3.3 (Demand Run Only, Synchronized) at the leftmost point of the graph, where the simulated time is

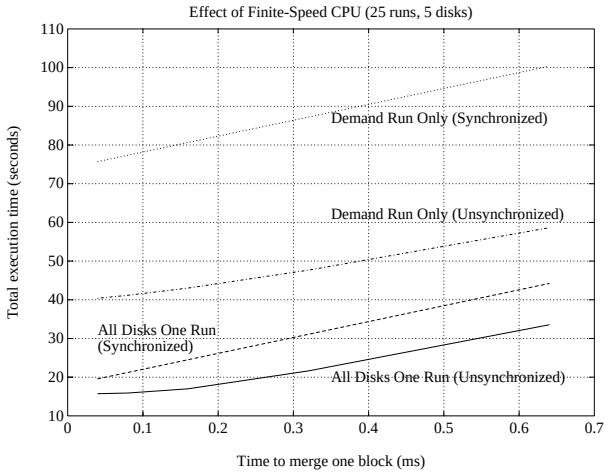

Figure 3.3: The effect of a finite-speed CPU. $D = 5$ disks. $k = 25$ runs. $N = 10$.

75.7 seconds. This corresponds to an "almost infinite" speed CPU, so that the total time plotted is approximately the I/O time.

In the *unsynchronized* case, additional reduction in the total I/O time may be possible, due to overlap with I/O requests being processed at other disks. Once the first block of a fetch of N blocks has been serviced, the CPU can proceed immediately, deplete another block, and issue a fetch for another N blocks. If these N blocks are issued at another disk, they will be serviced concurrently with the pending $(N-1)$ fetches.

Predicting the total execution time, however, is not straightforward due to this overlap. We estimate the asymptotic average parallelism obtained due to this overlap for large N. Note that without overlap (i.e. in the synchronized case) the asymptotic I/O time would be the same as that for a single disk. This is because the seek-distance reduction obtained by spreading the runs over multiple disks will no longer be significant. The I/O time for both single and multiple disks will approach the total transfer time.

We estimate the average number of overlapping I/O requests for large N using the following simple model.

Consider a game in which there are D urns. In one "round" of the game the player is given $D + 1$ balls, which must be thrown in sequence into the initially empty urns. In any throw there is a $1/D$ probability that the ball will go into a particular urn. The round ends whenever a ball is thrown into an already occupied urn. If the round ends after $j + 1$ balls have been thrown, define the *length* of the game to be j, $1 \leq j \leq D$. We want to determine the average length of a round.

Let Q_j be the probability that the round has length at least j, and P_j the probability that the length is exactly j. Obviously, $Q_0 = 1$. Suppose that the round has not ended after throwing j balls. The conditional probability that the game will end after throwing the $j + 1^{th}$ ball is j/D, since j of the D urns are already occupied. The conditional probability that the game

will not end after throwing the $j+1^{th}$ ball is $(D-j)/D$. Hence, for $1 \leq j \leq D$:

$$\begin{aligned} Q_j &= ((D-j)/D)Q_{j-1} \\ P_j &= (j/D)Q_{j-1} \end{aligned}$$

The correspondence between the game and the I/O problem being addressed is as follows. The urns correspond to the D disks. Throwing a ball into an empty urn corresponds to successfully initiating an I/O operation at a free disk. If the ball is thrown into an already occupied urn, the corresponding I/O request is queued at the disk. In this case, no further I/O requests can be made at least until the previous request to the disk has been serviced.

We make two assumptions to simplify the model. By focusing only on the case of large N, we can neglect any variance (due to the random nature of the seek and rotational delays) in the actual completion time of the individual disks, and assume that all disks finish at the same time. This corresponds to one round of the game. The length of the round is the number of disks that operated concurrently in that round. The second assumption is that in a sequence of D requests, the probability of making two requests for the same run is negligible. This assumption makes the analysis less tedious, but does not pose any conceptual difficulties; simulation indicates that this is a sufficiently good predictor. Under these approximations, the average length of a round is the average number of concurrent requests being handled by the disks.

The expected value of the number of concurrent requests is therefore given by[1] :

$$\sum_{j=1}^{D}(j\ P_j) = \sum_{j=0}^{D}(\ Q_j) \quad = \quad \sqrt{\frac{\pi D}{2}} - \frac{1}{3} + O(D^{-1/2})$$

The first equality follows by simplifying using the defining equations, and the closed-form summation is derived in [10]. The first two terms of the above expression is numerically evaluated for $D = 5$, $D = 10$ and $D = 20$. These result in an average overlap of 2.51, 3.66 and 5.29 respectively. Note that the best possible overlaps would be 5, 10 and 20 respectively.

Consider the case of $D = 5$ and $k = 25$. From equation 4, with $N = 30$, the synchronized I/O time is 58.75 seconds. Using the speedup of 2.51 due to overlap predicted by the equation above, the estimated (asymptotic) unsynchronized time for $k = 25$ and $D = 5$ is $58.75/2.51 = 23.4$ seconds. Figure 3.2 (c) (Demand Run only, 25 runs, 5 disks), with $N = 30$ has the simulated unsynchronized time of 25.3 seconds. For 10 disks, and 50 runs the estimated time is $117.51/3.66 = 32.1$ seconds; the simulated time shown in Figure 3.2 (b) (Demand Run Only, 50 runs, 10 disks), with $N = 30$ is 38.5 seconds. The discrepancy in both cases arises since the largest value of N simulated ($N = 30$) is less than that needed for asymptotic conditions. The significant fact to be noted is that the average concurrency that can be obtained by the overlap is only proportional to $\sqrt{D}$, rather than maximum D possible.

[1] We thank a referee for providing the simplification and reference cited below

Inter-Run Prefetching, Multiple Disks: With *inter-run prefetching*, the concurrency in the operations of the disks can be increased by employing a larger-sized cache. In contrast, with intra-run prefetching the concurrency asymptotes at $\sqrt{D}$, well below the maximum possible.

The inter-run prefetching strategy assuming *synchronized* prefetching, and combined with N-block intra-run prefetching is described by the pseudocode in Figure 3.4. Initially, the cache is loaded with N block from each of the k runs. At the start of any iteration of the loop, at least one block from each run will be present in the cache. The leading block from each run is a potential candidate for being depleted next. One of these k leading blocks is chosen with equal probability $1/k$, and depleted. If that run still has blocks present in the cache, no I/O operation is needed. However, if that run no longer has any cached blocks, an I/O operation will be required to retrieve the next block from that run before the merge can continue. If there is enough cache space to read N blocks from each disk, DN blocks are fetched– these consist of N blocks from the demand-fetch run, and N blocks from some run on each of the other disks. The choice of the run on each disk from which to prefetch is made randomly. If there is not sufficient cache space to prefetch from all disks, only the demand-fetch block (from the run that was depleted) is fetched.

The D independent disks will be able to fetch blocks from independent disk locations. In the case of synchronized prefetching, these DN reads must all complete before the CPU can proceed. For the unsynchronized case, as soon as the demand fetch block is available, the CPU can continue merging, and issue the next I/O request.

The fraction of I/O requests that were for DN blocks is the statistic *success ratio* determined by the simulation. We first analyze the total I/O time assuming a large enough cache to maintain a success ratio very close to 1. In the synchronized case, the time for the fetch of ND blocks is determined by the latest of the D disks to complete its service. The service time $\mathbf{S_i}$ for disk i is given by $\mathbf{S_i} = \sigma + \rho + \mathcal{T}N$, where σ and ρ are random variables denoting the seek time and rotational latency at that disk. The expected I/O time for all D blocks to be read is given by $E(max(\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_D}))$. The probability distribution governing σ is derived in section 3.1, and ρ is a random variable uniformly distributed between 0 and $2R$. A *crude approximation* for the expected value can be derived by assuming σ to be a constant using its expected value of $mk\mathcal{S}/3D$. The constant terms in $\mathbf{S_i}$ can be taken outside the *max* function. The expected value of the maximum of D independent random variables, each uniformly dis-

/ C is the cache size in blocks. k is the number of runs. D is the number of disks*/*

Initial State:
Add the first N blocks from each run to the cache.
For (i=1...k) a[i]= N;
/ a[i] is the number of blocks from run i in cache. */*
num_free_cache= C−kN;

Do **till all blocks merged**
{
Randomly choose a run, j, from which to deplete a block.
a[j]= a[j]−1;
num_free_cache= num_free_cache+1;
If (a[j]==0) { */* A run has emptied */*
If (num_free_cache $\geq$ (DN)) {
Randomly choose 1 run from each of the D−1 disks not containing the demand-fetch run.
Let these be $i_1, i_2, \ldots, i_{D-1}$.
Fetch N consecutive blocks from run j and from each of the runs $i_1, \ldots i_{D-1}$;
a[j] = a[j] + N; For ($k = 1, \ldots, D-1$) a[i_k] = a[i_k] + N;
num_free_cache= num_free_cache − (ND); }
Else { */* Less than (DN) free cache blocks */*
Fetch 1 block from run j;
a[j]= a[j] + 1;
num_free_cache= num_free_cache − 1; }
}
Else /* Next block of run j available in cache */
{ */* continue merging */* }
}

Figure 3.4: Inter-Run Prefetching

tributed between 0 and $2R$ is $2RD/(D+1)$. Hence:

$$E(max(\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_D})) \simeq \frac{mk\mathcal{S}}{3D} + \frac{2RD}{D+1} + \mathcal{T}N$$

Since ND blocks are read in this time interval, the average I/O time per block is:

$$\pi = \simeq \frac{mk\mathcal{S}}{3ND^2} + \frac{2R}{N(D+1)} + \frac{\mathcal{T}}{\mathcal{D}} \qquad (5)$$

For the case of $k = 25$, $D = 5$ and $N = 10$, this evaluates to $\pi = 0.702$ms, and the total I/O time is $25 * 1000 * \pi = 17.6$ seconds. The simulation time for this case (19.6 seconds) can be seen in Figure 3.3(All Disks One Run, Synchronized) at the fastest CPU speed.

The unsynchronized case is difficult to analyze except for large N. In this case the I/O time is $1000k\mathcal{T}/D$, the transfer time divided by the number of disks. This is a lower bound on the time using D disks. For $k = 25$ and 50 with $D = 5$, this evaluates to 10.2, 20.4 and 40.8 seconds respectively. Figure 3.5 shows the trend towards the lower bounds as N is increased. The value of N needed to reach the lower bound is much larger than 10. For $N = 30$, the I/O times for the two cases obtained by the simulation were

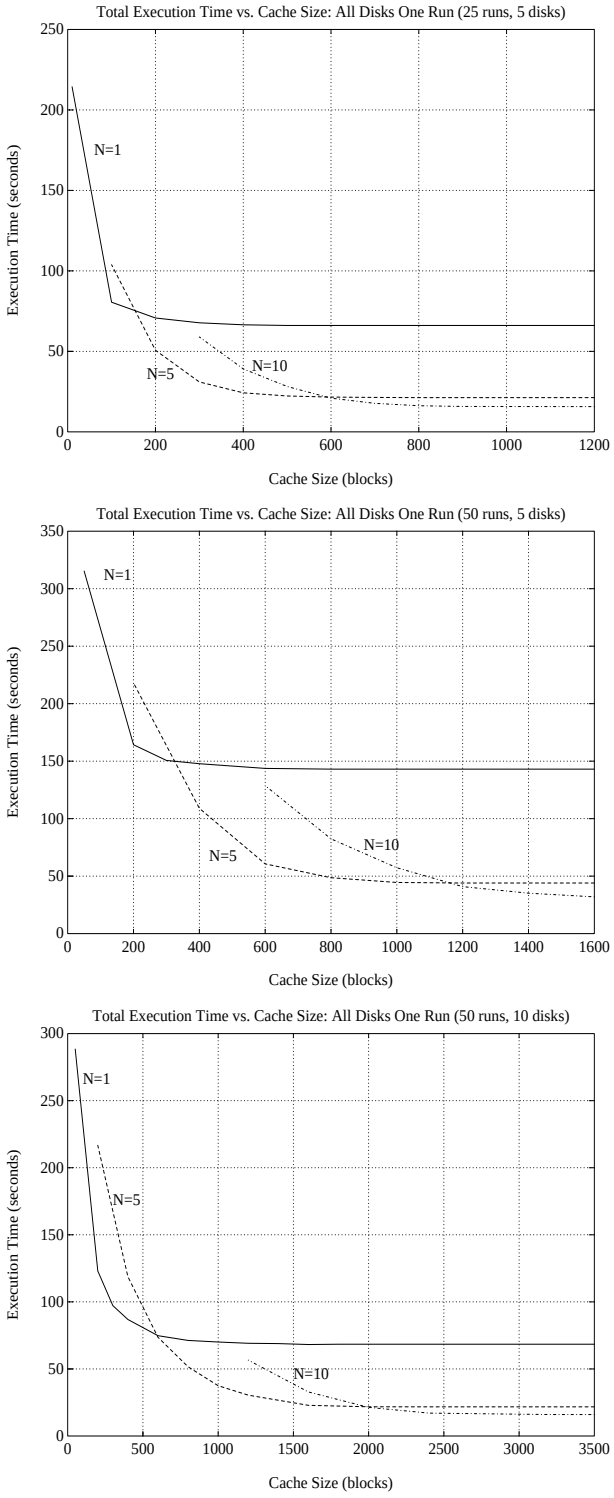

Figure 3.5: Cache requirements for $k = 50$ runs with $D = 5$ and $D = 10$ disks. Unsynchronized prefetching is used.

12.2 and 24.6 seconds respectively. This is not shown in the figures as the corresponding cache size is very large.

In the inter-run prefetching strategy, the average concurrency among the disks is determined by the success ratio. A value of 1 indicates that at every I/O operation an I/O request was initiated at all disks. The larger the cache size, the greater the probability that the cache has enough space to prefetch from all the disks, and the higher the success ratio will be. Figure 3.6(a), (b) and (c) show the relation between the cache size and the success ratio for different values of k and D. In each graph, the effect of increasing N, the degree of intra-run prefetching, on the cache size required to maintain a given success ratio is also shown.

The corresponding execution times using inter-run prefetching for different cache sizes are shown in Figure 3.5 The asymptote in each case corresponds to a success ratio of 1. For different ranges of cache sizes, the best execution time is obtained by using different values of N. Large values of N decrease the average seek and rotational latency of a block, but have a smaller success ratio for a given cache size. As N decreases, the increased concurrency in the operation of the disks provides a greater reduction in the I/O time than the increased average seek and rotational penalty. For a given cache size, there is an optimal value of N which provides the best tradeoff. Increasing the cache size, allows one to increase the value of N further reducing the I/O time. The lower bound of $1/D$ of the total transfer time is approached as the cache size, (and hence N) is increased. Recall that this was not possible with inter-run prefetching alone.

A comparison of the intra-run and inter-run prefetching can be seen in Figure 3.3. The legend Demand Run Only refers to intra-run prefetching and the legend All Disks One Run to the inter-run strategy. The inter-run prefetching strategy with $N = 10$ clearly outperforms the intra-run prefetching strategy, over the entire range of CPU speeds considered.

4 Summary and Conclusions

External mergesort is often I/O bound due to the need to retrieve the sorted runs from external storage. With the increases in the size of main memory in computer systems, multiple disks and aggressive prefetching can be employed to reduce the I/O time significantly.

Two prefetching strategies were studied in the paper. The performance of these strategies was evaluated using simulation for differing numbers of disks, numbers of runs, cache size and varying CPU speed. In most cases, simple analytical expressions were derived to explain the results for the asymptotic behavior of the strategies.

Intra-run prefetching is often employed in single-disk systems to reduce the average seek and rotational latency by amortizing their cost over a fetch of $N > 1$ blocks. In a multiple-disk situation intra-run prefetching can further reduce the I/O time due to overlap at the disks. The average I/O concurrency attained for large values of N (and hence cache size) is $O(\sqrt{D})$.

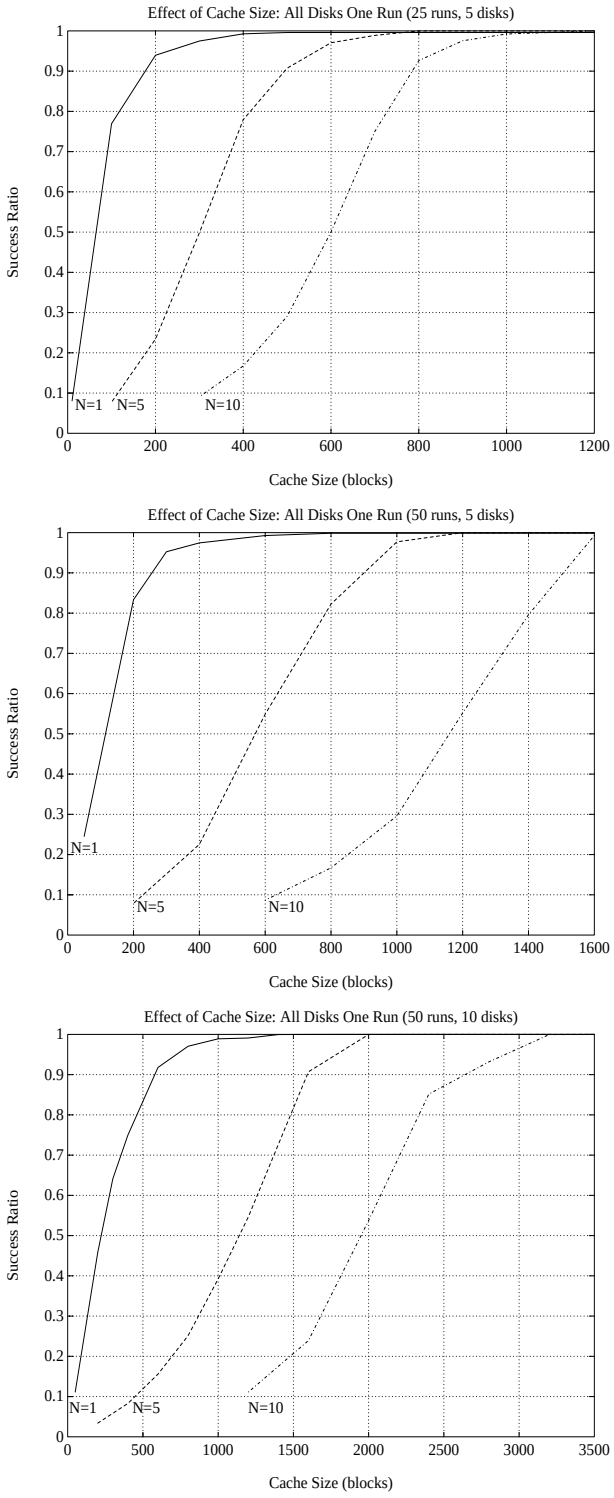

Figure 3.6: Cache requirements for $k = 25$ runs with $D = 5$ disks.

Hence, once this cache size is reached, the total execution time can not be decreased further by increasing the size of the cache.

Inter-run prefetching can be used to further decrease the I/O time by increasing the concurrency in the operation of the disks. For a given cache size, there is an optimal value of N, which best balances the success ratio (and hence the concurrency) and the possible amortization in seek and rotational penalties. The lower bound on I/O time using multiple disks is just the total transfer time divided by D. As the cache size is increased, the I/O time for inter-run prefetching with an appropriate choice of N, approaches this lower bound.

Acknowledgements: We thank the anonymous referees for their careful reading of the manuscript and their valuable suggestions.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[2] R. Agrawal, S. Dar, and H. V. Jagadish. Direct Efficient Transitive Closure Algorithms: Design and Performance Evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, 1990.

[3] Digital Equipment Corporation. *RA81 Disk Driver User Guide*. 1982.

[4] R. G. Covington. CSIM: An Efficient Implementation of a Discrete-Event Simulator. Master's thesis, Rice University, 1985.

[5] D.J. DeWitt D. Bitton, H. Boral and W.K. Wilkinson. Parallel Algorithms for Relational Database Operations. *ACM Transactions on Database Systems*, 8(3), 1983.

[6] G. A. Gibson. Performance and Reliability in Redundant Arrays of Inexpensive Disks. In *Proc. Int. Conf. on Management and Performance Evaluation of Computer Systems (CMG'89)*, pages 381–391, 1989.

[7] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding Techniques for Large Disk Arrays. In *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 123–132, 1989.

[8] B. R. Iyer and D. M. Dias. System Issues in Parallel Sorting for Database Systems. In *Sixth International Conference of Database Engineering*, pages 246–255, 1990.

[9] M. Y. Kim. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, 1986.

[10] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, 1973.

[11] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[12] D. F. Kotz and C. S. Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Computing*, 1(2):218–230, 1990.

[13] Sai–Choi Kwan and Jean-Loup Baer. The I/O Performance of Multiway Mergesort and Tag Sort. *IEEE Transactions on Computers*, 34(4):383–387, 1985.

[14] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proc. ACM Sigmetrics Conference on Measurement and Modeling on Computer Systems*, pages 69–77, 1987.

[15] V. S. Pai. Performance Analysis of Parallel I/O Models for External Mergesort. Master's thesis, Rice University, 1991.

[16] V. S. Pai, A. Schaffer, and P. Varman. *Markov Analysis of Multiple-Disk Prefetching Strategies for External MergeSort*. TR-9112, Electrical and Computer Engineering, Rice University, May, 1991.

[17] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 109–116, 1988.

[18] A. L. N. Reddy and P. Banerjee. An Evaluation of Multiple-Disk I/O Systems. *IEEE Transactions on Computers*, 38(12):1680–1690, 1989.

[19] A. L. N. Reddy and P. Banerjee. Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Computing*, 1(2):140–151, 1990.

[20] K. Salem and H. García–Molina. Disk Striping. In *Proc. Second IEEE Int. Conf. on Data Engineering*, pages 336–342, 1986.

[21] B. Salzberg. Merging Sorted Runs Using Large Main Memory. *Acta Informatica*, 27:195–215, 1989.

[22] J. D. Ullman and M. Yannakakis. The Input/Output Complexity of Transitive Closure. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 44–53, 1990.