

# Speeding up External Mergesort

LuoQuan Zheng and Per-Åke Larson \*

## Abstract

External mergesort is normally implemented so that each run is stored contiguously on disk and blocks of data are read exactly in the order they are needed during merging. We investigate two ideas for improving the performance of external mergesort: interleaved layout and a new reading strategy. Interleaved layout places blocks from different runs in consecutive disk addresses. This is done in the hope that interleaving will reduce seek overhead during merging. The new reading strategy precomputes the order in which data blocks are to be read according to where they are located on disk and when they are needed for merging. Extra buffer space makes it possible to read blocks in an order that reduces seek overhead, instead of reading them exactly in the order they are needed for merging. A detailed simulation model was used to compare the two layout strategies and three reading strategies. The effects of using multiple work disks were also investigated. We found that, in most cases, interleaved layout does not improve performance, but that the new reading strategy consistently performs better than double buffering and forecasting.

**Index terms:** sorting, external sorting, mergesort, run placement, buffering strategy

## 1 Introduction

Sorting is a classical problem in both theoretical and practical computer science. Sorting is used not only to produce well organized output, but may also be part of a more complex operation such as a relational join. Files are often maintained sorted on a key attribute in order to facilitate searching and processing.

External sorting refers to sorting more data than can be held in memory at one time. Mergesort [8] is the algorithm most commonly used for external sorting. Mergesort consists two phases: run formation and merging. In the run formation phase, the data to be sorted is divided into smaller sorted sets, called runs. Replacement selection [8] is the most popular run formation method because the runs produced are expected to be twice the size of memory. The runs are written to external storage as they are produced. In the merge phase, blocks from each run are read into memory, the records are extracted and merged into the final sorted output.

I/O time normally determines the elapsed time for mergesort. This paper investigates the I/O behaviour of mergesort, assuming that magnetic disks are used as secondary storage. We focus on the handling of intermediate results, i.e. the runs to be merged. In particular, we study how the elapsed time of mergesort is affected by different layout strategies and reading strategies.

---

\* Authors' address: P.-Å. Larson, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1; palarson@uwaterloo.ca ; LQ. Zheng, IBM Canada Laboratory, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3; rzheng@vnet.ibm.com

We make no particular assumptions regarding the source of the input data and the destination of the sorted output, except speed. For the problem to be interesting, input and output data must be produced and consumed sufficiently fast to make sorting the bottleneck. We assume that the runs are stored on one or more separate (work) disks – this is common practice. We also make the simplifying assumption that the space used for storing runs is contiguous on the work disks. This can be viewed as a first approximation of an extent based file system. Furthermore, we assume that the sort can be completed in a single merge pass. Given today’s main memory sizes and the use of disks, it is hardly ever necessary to use multiple merge passes [13].

The rest of the paper is organized as follows. The next section gives a brief summary of previous work of direct relevance to this study. In Section 3, the traditional mergesort algorithm is analyzed. Areas of inefficiency are identified, and several ideas for improvement are outlined. In Section 4, a heuristic algorithm for read scheduling is presented. Simulation results comparing different layout and reading strategies, under a wide range of sorting configurations, are presented in Section 5. In Section 6, we study the effects of storing the runs on multiple disks. Section 7 explains why our new reading strategy performs pays off. We summarize and conclude the paper in Section 8.

## 2 Background

Sorting is probably the most exhaustively studied problem in computer science. There is a vast literature on internal sorting, but less on external sorting. Early studies of external sorting focused on using tape as the secondary storage device. Knuth [8] provides extensive coverage of the fundamentals of sorting.

Kwan and Baer [9] studied the I/O performance of  $k$ -way mergesort. Their disk model assumes that seek time is proportional to the seek distance and rotational latency of half a revolution is charged to every disk access. The asymptotic I/O complexity of mergesort is studied using this model. They assume that seek time and rotational latency together dominate the transfer time. Their conclusion is that minimizing the number of merge passes by selecting  $k$  as large as possible does not always give the best performance.

Verkamo [16] compared the performance of mergesort and distributive sorting. He also pointed out that the internal sort phase and the external merge phase need not use the same amount of memory. A space-time integral was used as a measure of performance. His analysis favours small buffers, because the speedup obtained by increasing the buffer size is not enough to compensate for the additional space requirement.

Both Kwan and Baer, and Verkamo assumed that the block transfer time is much smaller than the rotational delay and seek time. Salzberg [13] pointed out that the availability of large main memory makes this assumption obsolete. She gave an example where the buffer size is 1Mb. This makes the disk overhead only 7.5% of the transfer time, even if the average seek time and rotational latency is charged to every disk access. She also showed that using two input buffers per run gives much better overlap of processing and I/O. Her recommendation is to use all available memory for input buffers and to have two input buffers per run.

Since sorting is such a time consuming task, exploiting parallelism is a natural next step. A good introduction to parallel sorting can be found in [2] and [5]. One of the strategies proposed to improve the performance of the merge phase is to have many processors organized into a tree. Each

leaf processor merges a portion of the initial runs and passes the result to its parent, and every interior processor merges the outputs from its children and passes the output to its parent. The final result is obtained at the root. Parallelism is exploited both through pipelining merge steps between levels of the tree, and through concurrent merging performed by processors on the same level.

Beck et al. [4] implemented this idea on a number of backend processors. Due to hardware limitations, very small buffers (1Kb) were used. They also studied different layout strategies. However, their findings are heavily influenced by the small buffer size.

Salzberg et al. [14] also studied this idea applied to a network of loosely-coupled processors where each processor has local disks and a large amount of main memory. They recommend using enough memory and processors so that only one merge pass is needed at each processor.

Another idea for parallelizing the merge phase is to split each sorted run into range-disjoint partitions, and have each processor merge data from a separate partition. Combining the outputs of all the processors gives the final results. Quinn [12], Iyer et al. [7] and Varman et al. [15] investigated algorithms for partitioning.

Aggarwal and Vitter [1] showed that mergesort is an optimal external sorting method (up to a constant factor) in the total number of I/O operations required. They also studied the use of  $P$  disks to obtain I/O concurrency. However, they made the unrealistic assumption that any set of  $P$  blocks can be fetched in one parallel operation. Vitter and Shriver [17] extended the analysis to the case where the  $P$  blocks in a parallel operation are stored on different disks. Under this model, an optimal deterministic algorithm is presented in [10]. The basic idea of the algorithm is to distribute the data blocks from each run over the  $P$  disks and, during the merge phase, the block that is the earliest to be used in the merge process from each disk is read into memory and merged into an almost sorted list. One additional pass is used to transform the list to a completely sorted one. While the last pass only adds a constant factor to the number of I/O operations, it transfers the whole set of data in and out of main memory again. A more practical approach is to use more buffer space to hold the data blocks in memory.

Pai and Varman [11] used a Markov chain to model prefetching in a multiple-disk environment. Only the merge phase is studied. Their study is mostly analytical, and the number of I/O operations is the only cost measured. Each run is stored on a separate disk, which means there is no I/O parallelism in the run formation phase.

### 3 Improving Mergesort

In this section, we analyze the traditional mergesort algorithm, identify areas of inefficiency and review ideas for improving performance. We assume a single work disk and leave the consideration of multiple disks to Section 6.

#### Layout Strategies

Data is accessed in different order in the two phases of mergesort. In the run formation phase, all data blocks from one run are written onto disk before any block from the next run is written. In the merge phase, the data block needed next in the merge process may belong to a different run every time. The exact order in which the data blocks are needed depends on the ordering of the

input records. This difference results in conflicting objectives for the placement of data blocks on disk.

Traditionally, mergesort uses *contiguous layout*, which favours the run formation phase. All data blocks from a run are placed consecutively on disk. This means that sequential I/O is used to write the runs onto disk, and hence the time required is minimum. However, in the merge phase, a seek is typically required each time a data block is read. If the input data is randomly distributed, the seek distance is (approximately) some multiple of the run size.

*Interleaved layout* is aimed at reducing the movement of the disk arm in the merge phase by placing data blocks from different runs next to each other. The first block from each run lies in consecutive disk addresses, followed by the second block from each run, and so on. The idea is to spend more time writing the runs to disk, laying them out in such a way that when data blocks from different runs are needed in the merge phase, they will be relatively close to each other.

For interleaved layout to be competitive, the additional cost for writing the runs has to be small. Compared with contiguous layout, writing the runs to disk using interleaved layout increases disk overhead in two ways: the number of seeks is higher, and the rotational latency may also increase. Interleaved layout always requires more seeks during the write phase because, when each run is written, the disk head has to travel through the entire portion of the disk that stores all the data. Compared with contiguous layout, interleaving with  $n$  runs will increase the number of seeks by a factor of  $n$ .

Interleaved layout may cause significant rotational latency if the block size and the exact location of each block are not carefully chosen. For example, if the block size is chosen so that the number of blocks that fit in one track is the same as the number of runs and, at each interleaving round, data blocks are always placed in a fixed order according to which run they are from, then the data blocks from the same run are always placed at the same rotational angle. When a run is written out, each block incurs a rotational delay of almost one disk revolution. This undesirable overhead can be avoided by making the block size equal to a full track.

## Reading Strategies

Using a larger block size reduces the total number of reads and the overhead incurred in the merge phase. If the amount of memory used is fixed, this means that fewer input buffers are available. The merge process requires at least one buffer per run. If no extra buffers are used, I/O cannot be overlapped at all with processing. Reading is stopped until some buffer becomes empty and merging has to wait until more data has been brought in.

Knuth [8] suggests the *forecasting* technique for reading. Forecasting uses one extra buffer. When one block from each run resides in memory, the last key of each block can be extracted and the keys compared to determine which block will be emptied first. The extra buffer is then used for reading the next block from that run while the CPU is working on the data already in main memory. The forecasting technique cannot perfectly overlap processing and read time.

Salzberg [13] strongly advocates using two input buffers for each run to achieve better overlap of processing and reading. First one block from each run is brought into memory. Then the merge process is started and the second block from each run is read at the same time. As soon as a buffer is emptied, it is used for reading another block from the same run. This is called the *double buffering* strategy. The drawback of this strategy is that either more memory is required or a smaller block

size has to be used, which means more disk reads.

Suppose that we have many more buffers available than the number of runs. The extra buffers can be used for reading and buffering some data blocks that are not immediately required for the merge process but are located right after a block that is required. There will be no seek or rotational overhead associated with these reads. If these blocks are needed for merging soon, the buffers they occupy will be emptied quickly and can be reused for reading other data blocks that are stored “nearby”. This observation lead us to consider more sophisticated reading strategies.

## Block Consumption Sequence

Merging consumes the data blocks in a certain order. We use the term (*block*) *consumption sequence* for the sequence in which data blocks are needed in the merge process. This sequence can be precomputed by extracting the highest key from each data block (during run formation) and sorting the keys. The storage required for storing these keys is low because the number of blocks is much smaller than the total number of records to be sorted. For example, if there is 200 Mb of data and the block size is chosen to be 33 Kb (one track on a Fujitsu M2344K disk), there are about 6200 blocks. Saving and sorting the keys takes little time. (The sorting can be done while the last run is being written to disk, so the elapsed time of the process is not affected.) However, storing the keys requires additional memory during the run formation phase. Either some memory has to be allocated for storing all the keys or they have to be written to disk. If the keys are kept in memory, the runs produced will be a little bit shorter, and hence we might end up with more runs to merge. If the keys are written to disk, additional I/O is needed. Neither way is expensive. In the previous example, if each key is 10 bytes long, it takes about 65 Kb to store the information needed to determine the consumption sequence. This is affordable in most systems today.

Knowledge of the block consumption sequence and extra buffer space can be exploited to speed up reading. Extra buffers can be used for reading and storing data blocks that are not needed immediately for merging but require little or no disk overhead to read. When deciding which blocks should be read ahead, the consumption sequence can help us choose blocks that will be used early in the merge process. This reduces the time buffers are occupied. The exact read schedule can be computed before the merge phase begins. This gives a new reading strategy that is applicable to any file layout. We call this the *planning strategy*. None of the reading strategies presented above attempts to reduce disk overhead in this way. An algorithm for computing read schedules is presented in the next section.

## 4 Read Scheduling

The following notation is used in this section:

- $n$  is the number of runs,
- $T$  is the number of data blocks,
- $B$  is the number of input buffers available,
- $M = (M_1, M_2, \dots, M_T)$  is the block consumption sequence, where  $M_i$  identifies a data block,

- $R = (R_1, R_2, \dots, R_T)$  is a read sequence, where  $R_i$  identifies a data block,
- $L$  is a function that maps data blocks to disk slots, and
- $L(M_i)$  and  $L(R_i)$  are the disk slot of the  $i$ th block in  $M$  and  $R$ , respectively.

Data blocks are labeled  $1, 2, 3, \dots, T$  in the order they they are output during run formation. The disk slots (addresses) are numbered  $1, 2, 3, \dots, T$  in the sequential order of the disk. For contiguous layout,  $L$  is simply the identity mapping, that is,  $L(i) = i, 1 \leq i \leq T$ . The read sequence  $R^S$  that satisfies  $L(R_i^S) = i, 1 \leq i \leq T$  represents a simple sequential scan. However, a simple sequential scan may not be a feasible read sequence. We may end up in a situation where all buffers are filled with data blocks required later in the merge and there is no space left for reading the block that is needed to continue the merge. The following definition characterizes a feasible read sequence precisely.

**Definition:**

A read sequence  $R$  is a *feasible read sequence* with  $B$  input buffers if  $\{M_1, \dots, M_{k-B+n}\} \subseteq \{R_1, \dots, R_k\}$  for all  $k$  such that  $B \leq k \leq T$ .

Intuitively, the definition states that, at any stage of the merge process where  $k > B$  blocks have been read,  $k - B$  of them have been consumed and emptied by the merge process, and there is at least one block from each run in memory currently used for merging.

The cost of a (feasible) read sequence  $R$  is measured by the time required to read in the  $T$  data blocks in the order specified by  $R$ . Since the transfer time is the same for all read sequences, we only need to consider the disk overhead incurred for each read. Provided that we know the location of each block, the cost of  $R$  can be calculated as the sum of the seek times and rotational latencies incurred.

Since there is a finite number of read sequences, an optimal read sequence must exist for a given number of input buffers. However, computing the optimal read sequence is expensive. The definition of feasibility does not provide any easy way to select all the feasible paths other than trying all possible paths. (The problem can be viewed as a traveling salesman problem with an additional constraint resulting from the feasibility requirement.)

We therefore consider heuristic algorithms. A natural approach is to start with the consumption sequence  $M$  and improve on it by some heuristic while retaining feasibility. Figure 1 presents an algorithm based on this idea. In each iteration, the algorithm extends the current read sequence  $R_1, R_2, \dots, R_{i-1}$  by inserting  $M_i$  into an appropriate place in the sequence. The idea is to move  $M_i$  to a place in  $R_1, R_2, \dots, R_{i-1}$  where it can be read with little or no disk overhead. The placement is based on the ordering in the current read sequence and the disk locations where these data blocks are stored. To ensure feasibility, the algorithm keeps track of a sequence  $F$  where  $F_j$  is the number of free buffers remaining immediately before reading  $R_j$ . When  $M_i$  is to be placed in  $R_p$  for some  $p < i$ , an extra buffer is needed to store  $M_i$  until blocks  $R_p, \dots, R_{i-1}$  have been read. Therefore, the  $F$  values of these blocks are reduced by one to reflect the allocation of a buffer for storing  $M_i$ . When  $F_j$  for some block  $j$  is reduced to zero, no block occurring after  $j$  in the consumption

sequence can be read before  $j$ . The variable  $s$  keeps track of the last position in  $F$  having the value zero.

```

 $R_1 := M_1; s := 1;$ 
let  $F_j := B - n$ , for  $1 \leq j \leq T$ ;
for  $i := 2$  to  $T$  do
  { Find the earliest position  $p$  where  $M_i$  can be inserted }
   $p := i$ ;
  for  $j := s$  to  $i - 1$  do
    if  $R_j$  resides on the same cylinder as  $M_i$  and  $L(R_j) > L(M_i)$ 
    then  $p := j$ ; exit loop; endif;
  endfor;
  { Insert  $M_i$  into  $R_p$  but make room first }
  for  $k := i$  downto  $p + 1$  do
     $R_k := R_{k-1}$ ;
     $F_k := F_{k-1} - 1$ ;
    if  $F_k = 0$  and  $s \leq k$  then  $s := k + 1$ ;
  endfor;
   $R_p := M_i$ ;
endfor;

```

Figure 1: Heuristic algorithm for computing read schedules.

We actually tested three different heuristic algorithms. The one in Figure 1 consistently outperformed the other two ([18]). The algorithm attempts to minimize the number of seeks, but does not take into account seek distance or rotational delay.  $M_i$  is moved to an earlier position if it can be read together with some other block residing in the same cylinder. Furthermore, we impose the restriction that blocks within a cylinder be read in increasing order. The algorithm is best suited for the case when the block size is a multiple of the track size.

The reason for requiring that blocks within a cylinder be read in increasing order is interesting. Experiments on a Sequent Symmetry using Fujitsu MK2344K disks revealed that the tracks within a cylinder do not all start at the same rotational angle but are shifted by approximately one sector between consecutive tracks. The explanation appears to be that when a disk is formatted, each track is formatted with a separate write command and by the time the format request for the next track arrives, the disk has already rotated a few degrees past the beginning of the previous track. This is barely noticeable when doing sequential I/O. However, reading the tracks in reverse order would take twice as long because an extra revolution is required for each read. This is true even if the offset between tracks is smaller than a sector. We believe that other disk systems exhibit the same behaviour but have not experimentally verified this.

The following example illustrates the algorithm and shows that it does not guarantee an optimal solution. Suppose that a cylinder holds five blocks and there are four runs and six buffers, that is, two buffers can be used for storing data not immediately needed for merging. Let  $L(M_i)$ , the disk addresses for the block consumption sequence, be

1 2 3 6 11 7 8 12 13 16 4 5 17 18 19

(Blocks on the same cylinder are underlined.) Reading in the block consumption sequence  $M$  takes 8 seeks. The heuristic algorithm produces the following read sequence which requires 6 seeks:

1 2 3 6 7 8 11 12 13 16 17 18 4 5 19

However, the following read sequence is also feasible and requires only 5 seeks:

1 2 3 6 7 8 11 12 13 4 5 16 17 18 19

The complexity of the algorithm is  $O(T^2)$  where  $T$  is the total number of data blocks. Each data block is considered once, and when inserting a block in the appropriate place in the read list, at most  $T$  blocks are affected. The computation can be done while the last run is being written to the disk, so the elapsed time for sorting is not affected. The amount of memory required by the algorithm is linear in  $T$ . This portion of the memory is different from the memory used for storing the highest key of each data block, because it is not needed in the run formation phase. Writing the last run normally takes much longer than computing the read schedule so we can wait until several blocks have been written and use the memory freed up.

## 5 Comparison of Different Strategies

When evaluating the I/O performance of mergesort, many researchers simply estimate the average time for a disk access and use this as the cost for each disk access (see e.g. [16] [13]). This type of analysis gives rather crude estimates. In reality, the I/O time depends on where data blocks are located on disk and in what order they are read. To obtain more realistic estimates, we decided to use a detailed simulation model

The simulator consists of a disk simulator and a CPU simulator. The disk simulator determines the completion time of each I/O request, as described in more detail in Appendix A. The CPU simulator determines the time required to produce an empty input buffer and generates an I/O request to fill the buffer. The time to produce an output record is assumed to be constant. The number of output records produced before an input buffer becomes empty can easily be calculated by examining the highest key of active buffer. This number is then multiplied by the time per output record to obtain the time required to produce an empty input buffer. The number of records remaining in each input buffer in use is also calculated at this time.

A sort can be viewed as a pipeline consisting of three stages: an input process, the actual sort process, and an output process. We assume that the input process and the output process are sufficiently fast so that the throughput of the pipeline is determined by the actual sort process. The elapsed time reported for our experiments includes the time for writing the runs on disk, the time to read all the data blocks for merging, and the processing time during merging. More precisely, the elapsed time reported is the time from the write of the first block of the first run until the last sorted record has been delivered to the output process. Time for run formation is not included. If replacement selection is used (and the CPU is sufficiently fast), the internal processing required for run formation can be completely overlapped by the writing of the runs. The time for computing

the consumption sequence and the read sequence for the planning strategy is not included either because they can be computed while the last run is flushed to disk.

All numerical results presented in this paper are averages of ten experiments with randomly generated input data. In most of the experiments, data was uniformly distributed among the runs. That is, the number of records with keys in a given range was about the same in each run. However, we also examined the effects of data skew (partially sorted input). The following method was used to simulate various levels of skew. The keys in a run  $i$  were uniformly distributed in a range  $Low_i$  to  $High_i$ . Each run had a key range of the same length but the key ranges of run  $i$  and run  $i + 1$  could be set to overlap. A parameter  $\alpha$  was used to control the overlap of the key ranges for run  $i$  and  $i + 1$  so that  $Low_{i+1} = (1 - \alpha)High_i + \alpha Low_i$ . Setting  $\alpha = 1$  produces completely random data. Decreasing  $\alpha$  increases the data skew (modeling partially sorted data). Setting  $\alpha = 0$  is equivalent to the input file already being sorted.

To validate the simulator, simulation results were compared against experimental results obtained on a Sequent Symmetry, a shared-memory multiprocessor system. The machine used for the experiments has ten Intel 80386 processors running at 16 Mhz, 64 Mb of memory, and eight Fujitsu M2344K disks. The system runs DYNIX, a version of the UNIX<sup>1</sup> operating system. A raw disk partition is available for experimental purposes on each disk. The raw disk interface gives complete control over the placement of blocks and bypasses the operating system's disk cache.

## Experimental Results

The main purpose of the initial experiments was to validate and calibrate the simulation model. Experimental results obtained on the Sequent machine were compared with simulation result obtained by setting the simulator's disk and CPU parameters appropriately. The experiments were done for both contiguous and interleaved layout. The effective disk transfer rate is just below 2Mb/s. The CPU time for merging is about 80% of the pure transfer time. Data was uniformly distributed among four runs. The forecasting technique was used for reading, requiring five buffers.

Figure 2a) and Figure 2b) show the elapsed time for sorting 10Mb of data using five buffers of size 4Kb, 8Kb, 11Kb, 16Kb, 32Kb, 33Kb (1 track) and 66Kb, respectively. Figures 2c) and 2d) plot the elapsed time for sorting files of different sizes. The buffer size was 99Kb (3 tracks). The simulation results are in excellent agreement with the experimental results. The results in Figure 2 also show that the two layout strategies give similar results. Interleaved layout performs slightly better with small buffers. However, later experiments showed that this is true only for uniformly distributed data, small buffers, and a small number of runs.

Figure 3 shows the results from (simulation) experiments with skewed data and large number of runs. All six combinations of layout and reading strategies were used. In these experiments, the buffer size was 99Kb for forecasting and double buffering, and 33Kb for planning.

Figure 3a) shows the results for sorting 200Mb of data distributed among four runs with varying levels of data skew. Forecasting uses five buffers (of size 99Kb), double buffering eight buffers (of size 99Kb), and planning 18 buffers (of size 33Kb), for a total of 495Kb, 792Kb, and 594Kb, respectively. Of the three reading strategies, planning has the best performance both for contiguous and interleaved layout. When data is skewed, the data blocks needed for merging tend to be

---

<sup>1</sup>UNIX is a registered trademark of AT&T Bell Laboratories

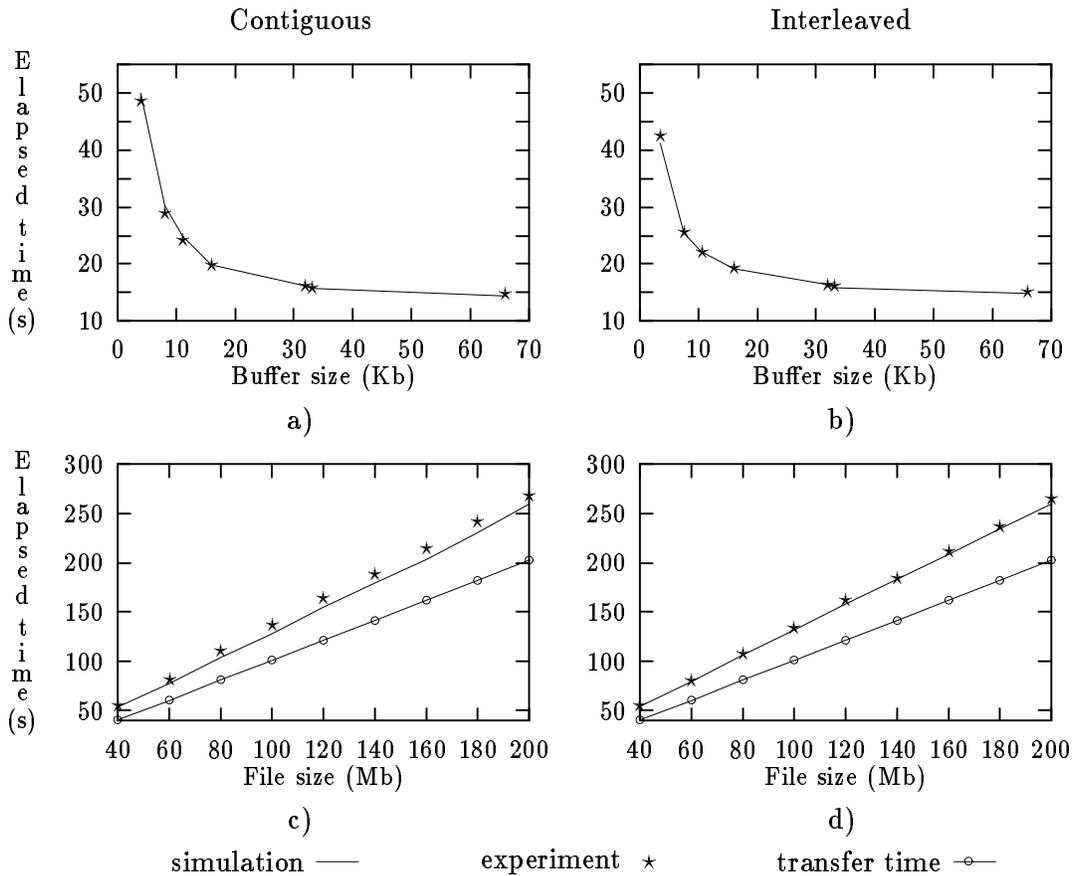


Figure 2: Validation and calibration of the simulator

concentrated in one or a few runs. This causes the performance of interleaved layout to deteriorate, while the performance of contiguous layout improves. In other words, contiguous layout takes advantage of partially sorted input.

Figure 3b) shows the results of sorting 200Mb of data evenly distributed (no skew) among 4 to 20 runs. Double buffering uses two buffers (of size 99Kb) per run and planning was set to use six buffers (of size 33Kb) per run, i.e. the same total amount of buffer space. Forecasting uses less buffer space: one buffer per run plus one spare (each of size 99Kb). For interleaved layout, the performance deteriorates when the number of runs increases. This holds for all three reading strategies. The extra time spent in writing data to disk is not fully compensated in the reading phase. Increasing the number of runs has little effect on double buffering and planning with contiguous layout.

Based on these experiments, we concluded that contiguous layout is preferable to interleaved layout. Interleaved layout is vulnerable to data skew and its performance deteriorates when the number of runs increases. Contiguous layout actually exploits data skew and is largely unaffected by the number of runs.

The amount of memory available in the run formation phase determines the number of runs

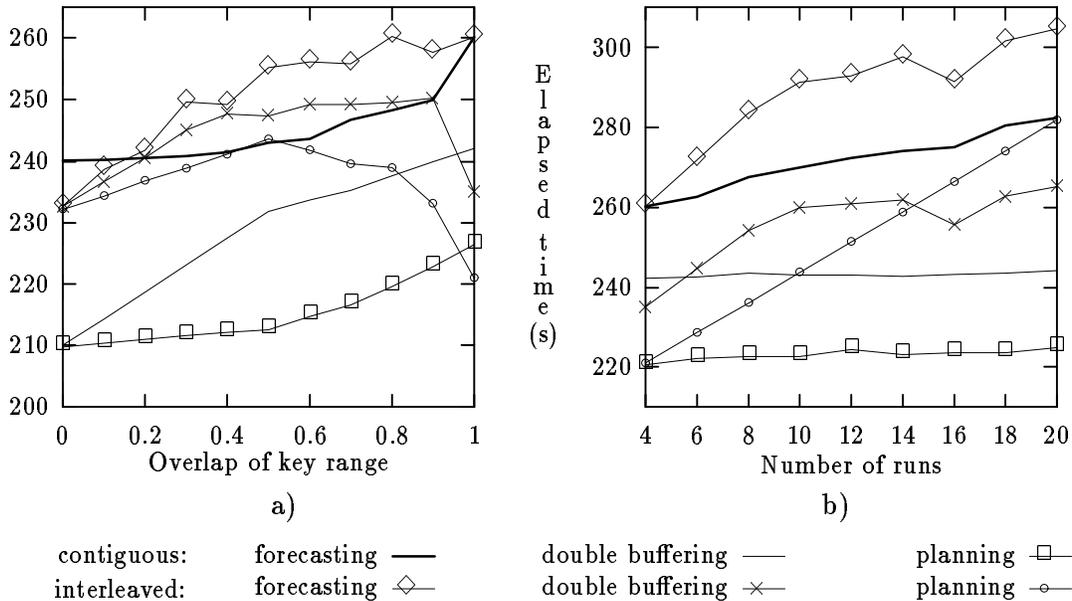


Figure 3: Effects of data skew and number of runs

formed. Similarly, the amount of memory available in the merge phase determines the buffer size and the number of buffers. Our next experiments examined the effect of varying the amount of memory used.

Figure 4 summarizes the results for the case when the same amount of memory is used in both phases. Figure 4a) plots the elapsed time as a function of memory size for the three reading strategies. The layout strategy is contiguous. The CPU speed was set so that the processing time is about 40% of the transfer time. The total amount of data is 400Mb, with no skew. Note that the number of runs now varies with the memory size – more memory results in fewer runs. The buffer size for planning is always 33Kb (1 track) and the number of buffers varies with the memory size: 5Mb corresponds to about 155 buffers. For the other two strategies, the number of buffers is determined by the number of runs and the buffer size varies. The buffer size is set to be the largest possible multiple of 33Kb (one track). Planning gives the best performance for all memory sizes. The elapsed time is less than 10% over the transfer time when the memory size is 8Mb or more. Double buffering can achieve similar performance but it requires more memory than planning. The performance of double buffering is poor if the memory size is 6Mb or less. In this situation, the simple forecasting technique performs better. If the amount of memory available is only 3Mb, the elapsed time for double buffering, forecasting and planning is 1320s, 700s and 578s, respectively.

Figure 4a) shows that using more than 10Mb of memory for the planning strategy improves performance only marginally. In a multi-user environment, we might not be willing to use all the memory available to achieve a very small improvement in one application. For example, using 10M of memory instead of 7M to sort 400M of data with the planning strategy improves performance by 17 seconds.

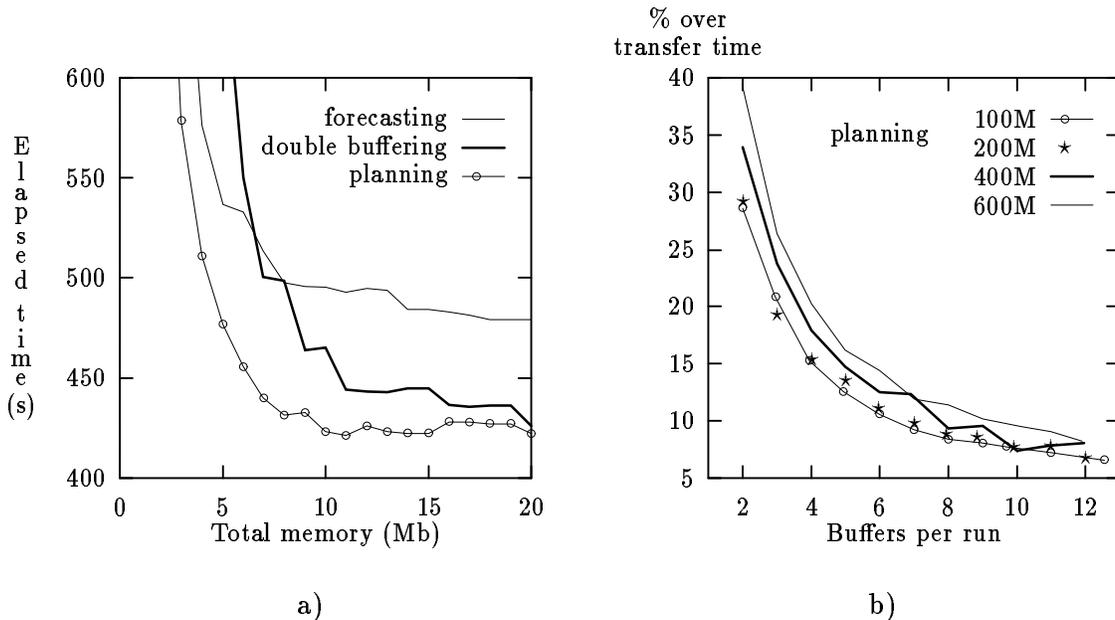


Figure 4: Effect of varying the amount of memory (both phases)

Let  $r$  denote the number of buffers per run used with the planning strategy. Figure 4b) plots the relative overhead as a function of  $r$  for files of four different sizes. Relative overhead is simply the percentage by which the elapsed time exceeds the transfer time. The relative overhead decreases rapidly when the number of buffers per run increases. Excellent performance is observed when  $r$  is in the range from 6 to 10. It does not seem worthwhile to use more than 10 buffers per run.

Consider sorting a file of size  $f$  using planning with  $r$  buffers of size  $b$  per run. If the amount of memory used in both phases is the same, the total amount of memory  $m$  needed to sort the file can be computed as follows. Using replacement selection, each run is expected to be of size  $2m$ . The relation between number of buffers and number of runs gives a formula for  $m$ :

$$r \times \frac{f}{2m} = \frac{m}{b} \implies m = \sqrt{rfb/2}. \quad (1)$$

Assume that  $b = 33Kb$  and  $r = 6$ . A file of 100Mb can then be sorted using 3.27Mb of memory while a file of 1Gb needs about 10Mb. Considering the memory sizes available today, this does not seem excessive.

We now relax the assumption that the same amount of memory is used in the two phases. We fix the number of runs (and hence the amount of memory used in the run formation phase) and study the effect of varying the amount of memory in the merge phase. Figure 5a) shows the elapsed time for sorting 400Mb of data evenly distributed among 10 and 40 runs. The amount of memory used is expressed as tracks (33Kb) per run. The CPU speed was set so that the processing time is about 40% of the transfer time. Both double buffering and planning are relatively insensitive to the number of runs if the amount of memory is proportional to the number of runs. The advantage of planning over double buffering is significant.

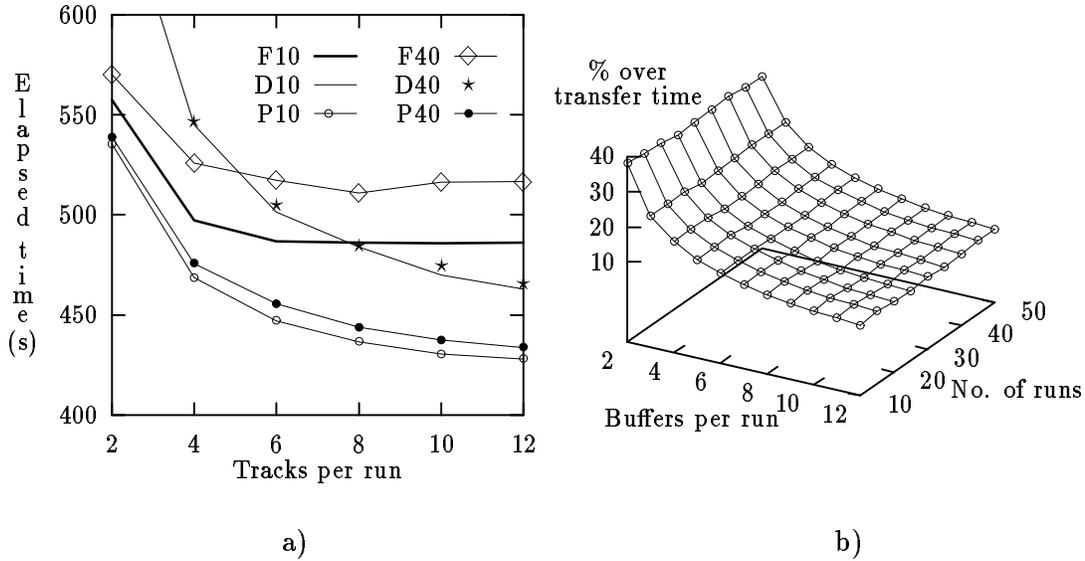


Figure 5: Effects of varying the amount of memory (merge phase only)

Figure 5b) shows the results of sorting 500Mb using the planning strategy. The buffer size is one track (33Kb). When the number of buffers per run is fixed, the performance is stable regardless of the number of runs to be merged. The amount of memory used in the merge phase should be determined by the number of runs produced in the run formation phase.

### Summary of Findings

- The performance of interleaved layout is better than contiguous layout only under very restrictive conditions: small buffers, few runs and no data skew. Even under these conditions, the advantage over contiguous layout is small.
- The performance of interleaved layout deteriorates with skewed data but contiguous layout is able to take advantage of data skew.
- The performance of interleaved layout deteriorates when the number of runs increases.
- Given the same amount of memory, planning performs better than double buffering and forecasting in all situations.
- Planning needs six to ten buffers per run during the merge phase to achieve good performance.
- Double buffering can achieve elapsed times similar to planning, but it requires more memory to do so.

## 6 Multiple Disks

The data rate of a single disk is limited and it is unlikely to increase dramatically in the near future. In contrast, advances in processor design have resulted in steady increases in processor speed. This exacerbates the imbalance between processing and I/O for external sorting. An obvious way to get further I/O speedup is to use multiple (work) disks for storing the runs.

We consider both fully synchronized and unsynchronized disks. Fully synchronized means that both arm movement and rotation are synchronized. When the disks are synchronized, the information in a data block is distributed and stored in the same relative position on each disk. All disks are involved in servicing an I/O request. Unsynchronized disks work completely independently and an I/O request affects only one disk.

It is easy to model the I/O behaviour of fully synchronized disks. Synchronizing  $P$  disks effectively increases the track size and the transfer rate by a factor of  $P$ . Therefore, we can simply model a set of synchronized disks as a single disk with larger tracks and faster transfer rate.

There are several ways to distribute the data blocks over a set of unsynchronized disks. One alternative is to keep the data blocks from each run together and distribute the runs among the disks. This layout is used in [11]. In the merge phase, data blocks from different runs can be read in parallel. In addition, there are fewer runs on each disk so fewer seeks are needed. However, there is no parallelism in the write phase. Furthermore, the I/O load is not evenly distributed among the disks unless the number of runs happens to be a multiple of the number of disks.

Another alternative is to distribute the blocks of each run among all the disks. The simplest policy is round-robin: store the first block on the first disk, the second block on the second disk, and so on. Both the writing during run formation and the reading during merging can now be parallelized. It is possible to keep multiple consecutive blocks from a run on the same disk. In the extreme case, a run is divided into  $P$  sequences of consecutive data blocks, and each sequence is written onto a disk in parallel. This is not a good idea. If the data happens to be skewed, reading will concentrate on one disk at a time. To achieve the best speed-up, the work load needs to be balanced so that all disks are kept busy.

The forecasting technique does not perform well for multiple disks. Only one input buffer is available for reading, so the disks cannot work in parallel. Double buffering is a viable alternative. However, if the number of runs is small (especially if there are fewer runs than disks), there may not be enough buffers to keep the disks fully occupied.

To extend the planning algorithm to multiple disks, we must be able to determine whether two blocks are stored in the same cylinder on the same disk. Suppose we have  $P$  disks, numbered  $0, 1, \dots, P - 1$ . If the blocks are distributed in a round-robin fashion, block  $i$  will be stored on disk  $i \bmod P$  in slot  $\lfloor i/P \rfloor$ . It is now straightforward to modify the heuristic planning algorithm in Figure 1 to handle multiple disks.

However, the algorithm tends to group data blocks from the same cylinder together in the read sequence. This means that read requests will be concentrated on one disk for a while, then on another disk, and so on. Consequently, this simple extension of the algorithm may not fully realize the potential for I/O parallelism. We therefore examined another variant of the algorithm, here called planning with dedicated buffers. The modification consists of treating each disk as a separate input stream. The consumption sequence is split into subsequences, one for each disk, and the input buffers are divided evenly among the disks. The planning algorithm is then applied

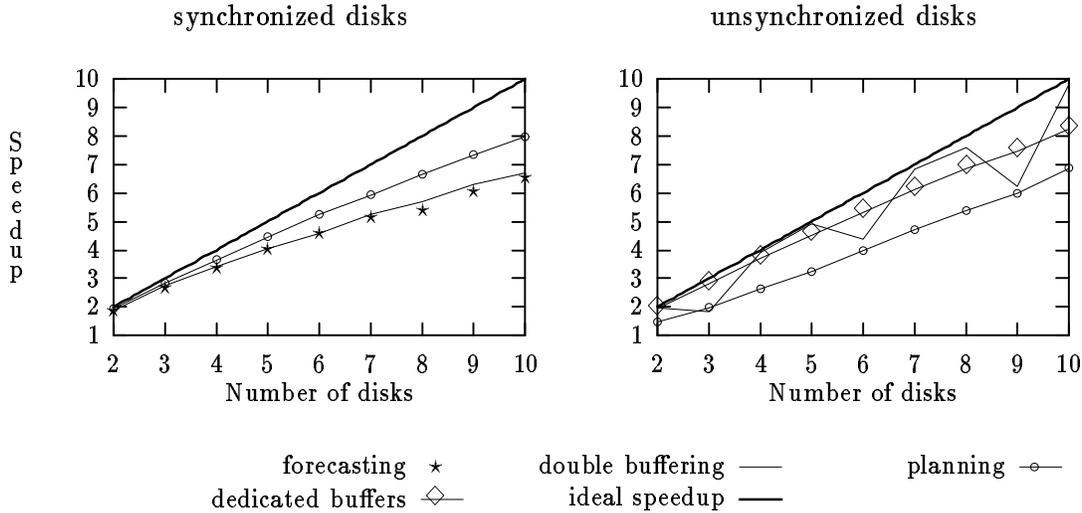


Figure 6: Speedup when using multiple disks

to each subsequence. During the merge phase, there is no explicit coordination among streams; whenever a buffer becomes empty, the next block in the read sequence for the corresponding disk is read.

Figure 6 plots the speedup obtained for synchronized and unsynchronized disks and different reading strategies. The speedup for  $m$  disks is defined as the ratio of the elapsed time when using one disk to the elapsed time when using  $m$  disks. In the ideal situation, the slope of the speedup curve is one. These experiments used an input file of 400Mb, divided into 20 runs, no data skew. The CPU speed was set high: the processing time for a block was set to about 8% of the block transfer time. Blocks were distributed in the round-robin fashion discussed above. 10Mb of memory was used in both phases. The buffer size is 462Kb (14 tracks) for forecasting, 231Kb (7 tracks) for double buffering and 33Kb (one track) for planning.

For synchronized disks, forecasting and double buffering achieve about the same speedup. However, the planning algorithm displays the best speedup. Note that the same speedup does not imply the same elapsed time because different methods have different elapsed times for the base case of one disk. The planning strategy has not only the best elapsed time but also achieves the best speedup.

Figure 6 also shows the results for unsynchronized disks. The curve labeled “dedicated buffers” plots the results of the modified planning algorithm where each disk is treated as a separate input stream. The advantage of treating each disk as a separate input stream is clear. The speedup curve for double buffering is somewhat erratic. This is (probably) caused by not having enough input buffers to balance the work load among the disks.

## 7 Why Planning Pays Off

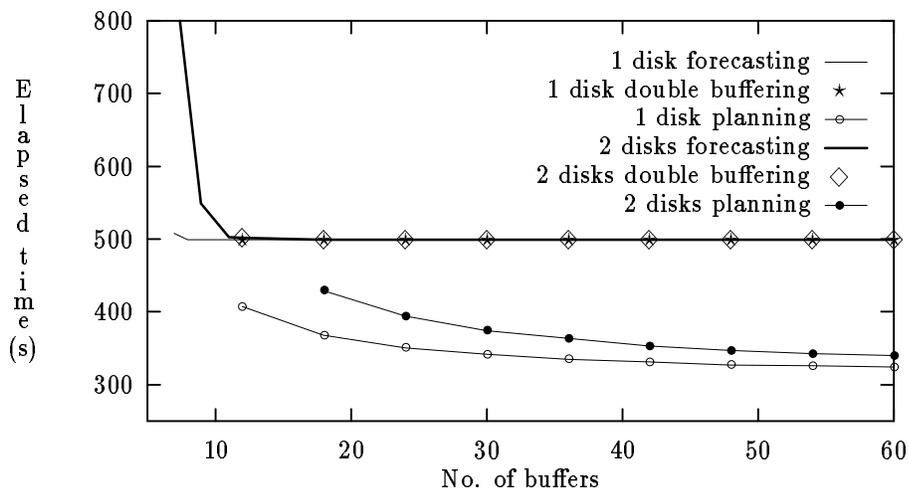


Figure 7: Comparing planning with extended forecasting and double buffering

The superior performance of the planning strategy is the result of two factors: using more buffers per run and preplanning of the read sequence. The question then is: How much does each factor contribute to the overall performance improvement? The experiments reported in this section provide the answer to this question.

Let us first consider how we might extend forecasting and double buffering so as to make use of additional buffers. As we have seen in previous sections, they cannot achieve the same performance as planning simply by using larger buffers but, perhaps, this can be achieved by using more buffers per run.

Extending double buffering is straightforward: simply allocate more than two buffers per run and whenever a buffer becomes empty fill it from the run to which it is allocated. This modification does not require knowledge of the consumption sequence. Forecasting can be extended by allocating a number of extra buffers without assign buffers to runs. Whenever a buffer becomes empty, fill it with the next block in the consumption sequence. It is easy to see that forecasting with extra buffers is equivalent to planning where the read sequence is equal to the consumption sequence.

Figure 7 compares planning with extended forecasting and extended double buffering. The first three curves plot the elapsed time for sorting 300Mb of data evenly distributed over six runs and using one work disk. Contiguous placement is used. The buffer size is one track (33Kb) for all three reading strategies. The total transfer time is 300 seconds. The other three curves plot the elapsed time for sorting twice the amount of data using two disks and with twice the CPU speed.

The results show that the performance of forecasting can be improved by using a few more buffers, especially for two disks. (The additional buffers make it possible to keep multiple disks busy.) However, the elapsed time for extended forecasting and extended double buffering levels off quickly. As soon as there are enough buffers to keep the disk(s) fully occupied, additional buffers do not result in any improvement. On the other hand, the elapsed time for planning continues to

decrease as the number of buffers increases. The reason is simple: (pre)planning the read sequence reduces the number of seeks. If enough buffers are available, the total seek time (disk overhead) can be reduced to a few percent of the transfer time.

## 8 Conclusion

### Summary

We investigated methods of improving the performance for external mergesort, concentrating on the handling of intermediate results, i.e., the runs to be merged. We examined two file layout strategies (contiguous and interleaved) and three reading strategies (forecasting, double buffering and planning).

The planning strategy reduces the read time by exploiting additional buffer space and knowledge of the block consumption sequence. We first showed how to obtain the block consumption sequence and how this information might be useful in reducing the read time. We then gave a simple heuristic algorithm for (pre)computing read sequences based on the consumption sequence and the placement of data on disk. To the best of our knowledge, the idea of explicitly computing the block consumption sequence and exploiting it to reduce read time is new.

A large number of simulation experiments were performed to determine the effects of buffer size, CPU speed, data skew, amount of buffer space, and number of runs. The results show that interleaved layout performs better than contiguous layout only under very restrictive conditions: few runs, small buffers, and no data skew. However, even in this situation, the difference between contiguous and interleaved layout is small. The performance of interleaved layout deteriorates rapidly if the data is skewed, or the number of runs is high.

The results also show that the planning strategy consistently gives better performance than forecasting or double buffering. This is mainly the result of a reduction in the number of seeks. Beyond a certain point, double buffering and forecasting do not benefit from additional buffer space. Planning, on the other hand, continues to benefit. The total seek overhead can be reduced to a few percent of the elapsed time using only a moderate amount of buffer space.

We also examined the effects of using multiple disks for storing the runs. We rejected the layout strategy that stores each run on a separate disk because writing cannot be parallelized. The planning strategy was extended with the objective of not only minimizing disk overhead, but also keeping all disks working. The planning strategy gives the best performance also in this case. It achieves close to optimal speed up, provided that the CPU is sufficiently fast to keep up with the combined transfer rate.

### Further Work

Our heuristic planning algorithm is simple but certainly not optimal. In particular, the algorithm moves a data block to the earliest convenient position, even if it can be read at a later time without additional overhead. This means that a data block not needed for merging may be read earlier than necessary and, consequently, occupying a buffer longer than necessary.

For multiple disks, we suggested dividing the buffers equally among the disks, dedicating each input buffer to reading data blocks from a single disk and scheduling reads for each disk inde-

pendently. Alternative ways for read scheduling (without dedicating buffers) may further improve performance.

The study reported in this paper assumed that the space used for storing runs is contiguous on disk. This can be viewed as a first approximation of an extent-based file system. However, many systems – Unix, in particular – do not have extent-based file systems and give the user little or no control over where individual “chunks” of a file are placed on the disk. The planning strategy needs to be modified significantly to be applicable to systems of this time. The seek cost and rotational delay must now be estimated and explicitly taken into account when planning the read sequence. Work in this direction is currently in progress.

## Implementation Recommendations

Based on the findings of this study, a number of implementation recommendations emerge.

- Contiguous layout is preferable to interleaved layout because of its robustness. It performs well regardless of the number of runs to be merged and actually exploits data skew.
- For reading, the planning strategy is preferable. It consistently outperforms forecasting and double buffering by reducing the number of seeks.
- The buffer size should be chosen according to the characteristics of the disks used. For the planning strategy, a buffer size of one track is appropriate in most cases.
- The planning strategy needs 6-10 buffers per run to perform well. If the file size is known, and the same amount of memory is used in both phases, formula 1 on page 12 can be used to determine the amount of memory. If the file size is not known in advance, use whatever memory is available during run formation. The amount of memory to use in the merge phase can then be determined according to the number of runs obtained.
- If multiple (unsynchronized) disks are used, distribute the blocks of each run among the disks in round-robin fashion. The following simple adaptation of the basic planning algorithm was found to perform well. Divide input buffers equally among the disks and dedicate each buffer to reading data blocks from a specific disk. Schedule the reads for each disk independently using the planning algorithm.

## Acknowledgements

This research was supported by the Natural Science and Engineering Research Council. The experiments reported in section 7 were prompted by a question raised by one of the anonymous referees.

## References

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

- [2] Akl, S.G., *Parallel Sorting Algorithms*, Academic Press Inc., Orlando, Florida, 1985.
- [3] Baer, J.L., *Computer Systems Architecture*, Computer Science Press, Rockville, MD, 1980, pp. 255-257.
- [4] Beck, M., Bitton, D., Wilkinson, W.K., Sorting Large Files on a Backend Multiprocessor, *IEEE Trans. Comput.*, 37, 1988, pp. 769-778.
- [5] Bitton, D., Design, Analysis, and Implementation of Parallel External Sorting Algorithms, Ph.D. dissertation, Univ. Wisconsin-Madison, TR 464, Jan. 1982.
- [6] Hennessy, J.L., and Patterson, D.A., *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann Publishers, 1990 (pp. 557-559).
- [7] Iyer B.R., Ricard R. G., and Varman P.J., Percentile Finding Algorithm for Multiple Sorted Runs, *Proc. of the 15th Intl. Conf. on Very Large Data Bases*, Amsterdam, 1989, pp. 135-144.
- [8] Knuth, D.E., *The Art of Computer Programming: Volume 3*, Addison-Wesley, Reading, Ma, 1973.
- [9] Kwan, S.C., Baer, J.L., The I/O Performance of Multiway Mergesort and Tag Sort, *IEEE Trans. Comput.*, C-34 Special Issue on Sorting, 1985, pp. 383-387.
- [10] Nodine, M.H., Vitter, J.S, Greed Sort: An Optimal External Sorting Algorithm for Multiple Disks, Department of Computer Science, Brown University, Technical Report CS-90-04, February 1990.
- [11] Pai, V.S., Varman P.J. Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis, *Eighth International Conference on Data Engineering*, February 1992, pp. 273-282.
- [12] Quinn, M.J., Parallel Sorting Algorithms for Tightly Coupled Multiprocessors, *Parallel Computing*, North-Holland, 1988, pp. 349-357.
- [13] Salzberg, B., Merging Sorted Runs Using Large Main Memory, *Acta Informatica*, 27, 1989 pp. 195-215.
- [14] Salzberg, B., Tsukerman, A., Gray, J., Stewart, M., Uren, S., Vaughan, B., FastSort: A Distributed Single-Input Single-Output External Sort, *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, May, 1990, pp. 94-101.
- [15] Varman, P.J., Scheufler, S.D., Iyer, B.R., Ricard, G.R., Merging Multiple lists on Hierarchical-Memory Multiprocessors, *Journal of Parallel and Distributed Computing*, 12, 1991, pp. 171-177.
- [16] Verkamo, A.I., Performance Comparison of Distributive and Mergesort as External Sorting Algorithms, *Journal of Systems and Software*, 10, 1989, pp. 187-200.

- [17] Vitter, J.S., Shriver, E.A., Optimal Disk I/O with Parallel Block Transfer *Symposium on the Theory of Computing*, May, 1990 pp. 159-169
- [18] Zheng, L.Q., Speeding up External Mergesort, Master’s thesis, University of Waterloo, 1992

## A Modeling Disk Access Time

In this appendix, we present in detail our model for estimating disk access time. The model relies only on parameters generally provided by disk manufacturers. Table 1 lists the parameters for the Fujitsu M2344K disk used in our experiments.

Definition	Symbol	Fujitsu M2344K
Number of cylinders	$N_c$	624
Tracks per cylinder		27
Sectors per track		66
Sector size		512 bytes
Rotational speed		3600 rpm
Positioning time		
Track-to-track	$S_{min}$	4 ms
Average	$S_{avg}$	16 ms
Maximum	$S_{max}$	33 ms

Table 1: Disk parameters

### Seek Time Function

We make the assumption that moving the disk arm over a fixed distance takes the same amount of time regardless of where the arm is initially located. Therefore, seek time can be modeled as a function  $s(x)$  of the seek distance  $x$ .

To move the disk arm from one cylinder to another, the disk arm must accelerate to overcome inertia, reach its maximum traveling speed, decelerate as it reaches the requested position, and then wait to allow the arm to stop vibrating. For short distances, the acceleration phase plays a larger role. A common technique (see e.g. [3] [6]) is to model short seeks by a square root function. This gives the following seek time function:

$$s(x) = \begin{cases} \sqrt{x-1} + a(x-1) + S_{min} & \text{if } x \leq \beta \\ \frac{S_{max}-b}{(N_c-1)-\beta}(x-\beta) + b & \text{otherwise.} \end{cases}$$

$\beta$  represent the distance for the disk arm to reach its maximum speed. Once a value for  $\beta$  has been chosen, we have two linear equations to solve for parameters  $a$  and  $b$ . By definition, the average seek time  $S_{avg}$  is calculated as  $S_{avg} = \frac{\sum w(x)s(x)}{\sum w(x)}$ . The weight  $w(x)$ ,  $x = 0, \dots, N_c - 1$ , is the number

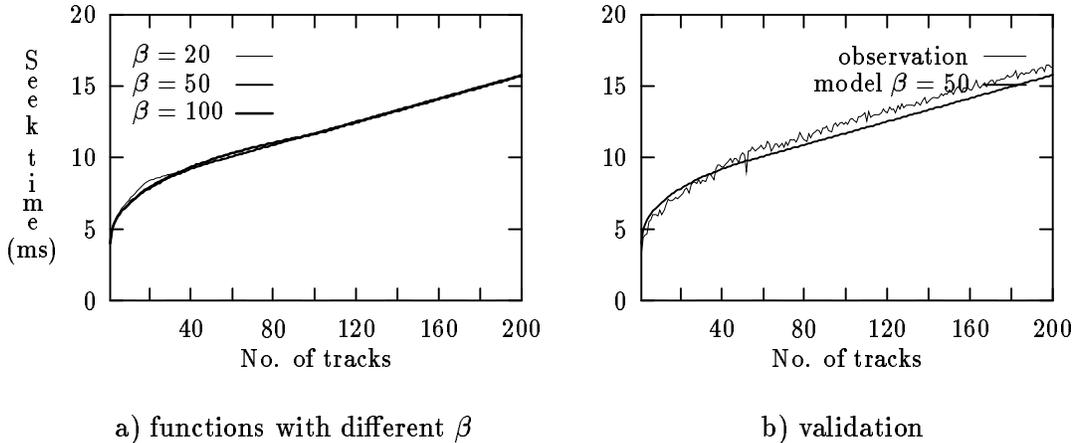


Figure 8: Seek time function

of possible seeks of distance  $x$ . It is not hard to determine that  $w(0) = N_c$  and  $w(x) = 2(N_c - x)$  for  $x > 0$ . The second equation,  $a(\beta - 1) + S_{min} = b$ , is used to force  $s(x)$  to be continuous.

It is not clear what value to use for  $\beta$  without further knowledge about the disk characteristics. Figure 8a) shows three functions with different  $\beta$  values. As we can see, the model is not very sensitive to the  $\beta$  value. There is a large range to choose from without significantly changing the seek time function.

We arbitrarily set  $\beta$  to 50 cylinders. The resulting seek time function for the M2344K disk is then

$$s(x) = \begin{cases} \sqrt{x-1} - 0.02653(x-1) + 4 & \text{if } x \leq 50 \\ 0.04066(x-50) + 9.69995 & \text{otherwise.} \end{cases} \quad (2)$$

The following experiment was conducted to measure the real seek time. Two I/O requests are issued with the first one for a sector on some cylinder, and the next one for a sector on another cylinder  $x$  tracks away. The completion times for both requests are measured. The second request is issued after the first request with a delay that is small enough to guarantee that the first one has not finished. (This ensures that the second I/O request is sent to the disk controller immediately after the first request has completed.) However, to determine the seek time, we need to find out which sector in the target track can be read without rotational delay. We repeat the experiment for each sector in the target track. The minimum completion time (minus transfer time) represents the seek time for distance  $x$ . The results are plotted in Figure 8. The model  $s(x)$  (based on the published disk parameters in Table 1) and the experimental results are in good agreement.

## Disk Simulator

The behaviour of a disk can be simulated by keeping track of the location of the disk arm and the time that each I/O request is issued and completed. When serving an I/O request, if the request occurs some time  $t$  after the last request has completed, the position of the disk heads is

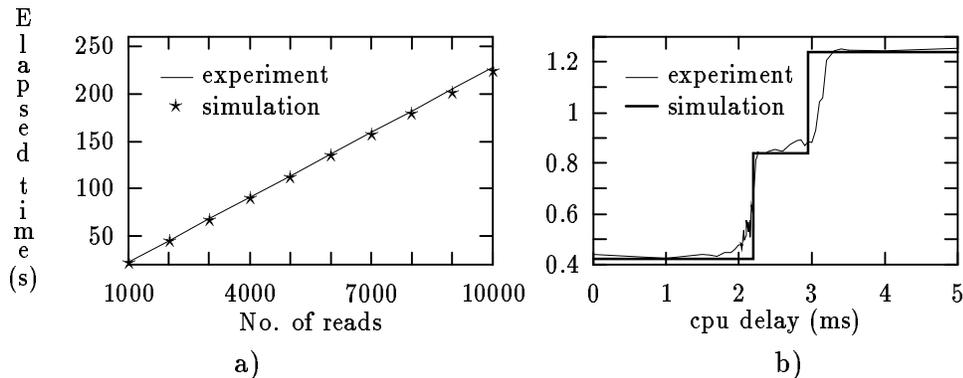


Figure 9: Validation of disk model

appropriately adjusted to simulate that the disk was idle for time  $t$ . The completion time of the request is then calculated by adding the (estimated) seek time, rotational delay to reach the desired sector, and the transfer time.

Figure 9 shows the results of two experiments performed to validate the disk simulator against the M2344K disk. In the first experiment, data blocks of size 8Kb distributed over 200 cylinders were randomly selected and read. The simulated time and the the actual time observed on the Sequent machine are plotted in Figure 9a). The two match closely. In the second experiment, 100 blocks of 8Kb were read sequentially (using double buffering) with different time delays between reads. This delay represents the time to process a block. The operating system introduces a further delay because of the processing required to schedule the I/O request. If the combined delay is small, the total cost is virtually equal to the transfer time. When the combined delay exceeds the transfer time of a block, there is a sudden increase in total time as shown in Figure 9b). I/O requests are not issued fast enough to keep the disk busy, and each request incurs a large rotational delay. (Note that Figure 9b) shows the elapsed time *per 8Kb block*, not the total elapsed time.) The transfer time for an 8Kb block is 4.04 ms. The time required by the operating system to process an I/O request is 1.85 ms (estimated). Consequently, when the time to process a block exceeds  $(4.04 - 1.85) = 2.19$  ms, the time per block suddenly increases. The effective transfer rate is reduced to half, from four 8 Kb blocks per revolution to two blocks.

## Biographies

Per-Åke (Paul) Larson received his Ph.D. degree in business administration and operations research from Åbo Swedish University, Finland in 1976. He was Assistant Professor of of Computer Science at Åbo Swedish University from 1974 to 1981. In 1981 he joined the University of Waterloo, Waterloo, Ont., Canada where he is currently Professor of Computer Science. His research interests include multidatabase systems, query processing, query optimization, and file structures. He has published widely in the database field. Dr. Larson is a member of the Association for Computing Machinery and the IEEE Computer Society.

LuoQuan Zheng received the B.Math and M.Math degrees in Computer Science from the Uni-

versity of Waterloo in 1990 and 1992, respectively. He is currently working on the Parallel Database project at IBM Canada Laboratory.