

# Performance Comparison of Prefetching and Placement Policies using Parallel I/O<sup>\*\*\*</sup>

K. K. Lee<sup>†</sup>   M. Kallahalla<sup>†\*\*</sup>   B. S. Lee<sup>†</sup>   P. J. Varman<sup>†\*\*\*</sup>  
askklee@ntu.edu.sg   kalla@rice.edu   ebslee@ntu.edu.sg   pjv@rice.edu

<sup>†</sup>School of Applied Science  
Nanyang Technological University  
Singapore 639798

<sup>†</sup>Department of ECE  
Rice University  
Houston TX 77251

**Abstract.** We present a simulation study of several prefetching policies to improve the I/O performance of external merging using parallel I/O. In particular we consider traditional sequential prefetch, forecast-based greedy prefetching, and oblivious prefetching. In conjunction with the prefetching policies we evaluate the benefit of two different data placement strategies: run-level striping and block-random placement, in the presence of data skew.

We show that the I/O performance is greatly improved by using forecasting techniques. This method outperforms the other policies in achieving higher disk parallelism, and scales well with increased numbers of disks and increasing data skew. Additionally, the performance of block-random data placement is shown to be uniformly good, independent of the data skew, while the performance of run-level striped placement degrades with increasing skew.

## 1 Introduction

The increasing imbalance between the speeds of processors and I/O devices has resulted in the I/O subsystem becoming a bottleneck in many applications. The use of multiple disks to build a parallel I/O subsystem has been advocated to increase I/O performance and system availability [4], and most high-performance systems incorporate some form of I/O parallelism. In applications like database systems that perform large amounts of I/O the parallelism can provide significant performance improvements.

External (disk-based) merging is an important I/O-bound problem occurring frequently in database applications. In our study of parallel I/O we will use this problem as a representative case study. The input in the external merge problem consists of a number of sorted runs of data<sup>4</sup> that need to be merged together

---

\*\* A shorter version appeared in Proceedings of PDCN'97

\* A preliminary version of this paper has appeared in the proceedings of IASTED International Conference on Parallel and Distributed Computing Networks, 1997

\*\*\* Partially supported by a grant from the Schlumberger Foundation

<sup>4</sup> The sorted runs are often obtained as the first phase of an external sorting algorithm.

into a single sorted run. Typically the runs are too large to all fit in primary storage, and hence the algorithm must perform the merge with only a limited amount of main memory. In multiple-disk systems it may be possible to improve the performance of external merging by placing the data on several disks which can be accessed in parallel. This raises the problem of deciding how to store the data on the disks (*data placement problem*), and the problem of deciding which blocks to read in parallel (*prefetching algorithm*).

In this paper we consider the data placement and prefetching problems for external merging using multiple disks and a limited amount of main-memory buffer. We compare using simulation, the relative performance of several prefetching methods, sequential prefetch, greedy scheduling with forecasting and oblivious prefetching. In conjunction with the prefetching policies, we compare two data placement methods, run-level striping and block-random placement.

Sequential prefetch with run-level striping is the traditional method employed in many commercial database systems (e.g. DB2, SQL/DS). The idea of forecasting was originally proposed by Knuth [8], and has since been adapted to several different contexts to improve merge performance (see Section 4). In our adaptation for run-level striping on multiple input disks, forecasting can be incorporated with almost no overhead in an on-line manner. For block-random placement, a small amount of preprocessing of the blocks during run creation is needed to obtain an on-line schedule [2] (see Section 2.3). Our results show that forecasting results in markedly reduced I/O times and increased disk parallelism, in comparison with both sequential and oblivious prefetching schemes. In fact, by specializing the results of [16] to this problem, it follows that greedy scheduling with forecasting is the theoretically optimal prefetching algorithm in our parallel-disk model. In addition, we empirically compare the relative performance of run-level striping and block-random placement as a function of data skew. While forecasting is easier to implement with run-level striping, block-random placement has certain advantages in some situations. Since the block-random strategy is insensitive to the actual data values, it has uniformly good performance even for highly-skewed data. Secondly, if the output of the merge needs to be stored back on disk rather than consumed immediately, block-random placement facilitates parallelism in the writes.

The primary mechanism employed to obtain I/O parallelism is *prefetching*. Prefetching refers to the reading of a data block from a disk before it is needed by the computation. When a read operation is initiated at a disk, prefetch operations for blocks from other disks may also be scheduled in the same parallel I/O step, thereby increasing the I/O parallelism. These prefetched blocks are held in the disk buffer until they are needed. A second benefit of prefetching is the overlap of CPU and I/O operations, whereby CPU operations on memory-resident data proceed concurrently with the fetch of data not yet required by the computation.

Figure 1 shows how sequential I/Os are converted to parallel I/Os by prefetching. There are two disks holding blocks  $A_1, A_2$  and blocks  $B_1, B_2$  respectively. Suppose the computation required the blocks in the order  $A_1, B_1, B_2, A_2$ .

Rather than wait to read a block when it is requested (which could result in four non-overlapped I/O operations as in Fig. 1 (a)), prefetching can reduce the number of parallel I/O operations (Fig. 1 (b)): on the first parallel I/O,  $B_1$  is prefetched from disk 2 along with the demand I/O for  $A_1$  from disk 1; on the second I/O,  $A_2$  is prefetched along with  $B_2$ . Also CPU activity that computes using blocks  $A_1, B_1$  can be overlapped with the prefetch of blocks  $A_2, B_2$ , by initiating the reads earlier. Prefetching implies the ability to predict the future

CPU		$A_1$	$B_1$	$B_2$	$A_2$
DISK 1	$A_1$				$A_2$
DISK 2			$B_1$	$B_2$	

CPU		$A_1, B_1$	$B_2, A_2$
DISK 1	$A_1$	$A_2$	
DISK 2	$B_1$	$B_2$	

Fig. 1. (a) No overlap (b) Disk-Disk and Disk-CPU overlap

sequence of I/O requests. If the entire I/O-request sequence is known *a-priori*, a minimal I/O schedule may be constructed using an *off-line* algorithm. In contrast, in the external merging problem the I/O-request sequence depends on the values of data that are known only at run time. In this case the scheduling algorithm needs to *forecast* the blocks to fetch based on the current state of the computation. For external merging using run-level striping we show how this prediction can be made easily at run-time, thereby obtaining an *on-line* schedule. For the case of random block placement, a small amount of preprocessing on the blocks during the run creation phase is needed to support the creation of an on-line schedule.

The rest of the paper is organized as follows. In Section 2 a model of the I/O system and the details of the external merging problem are described. Simulation results on the I/O performance of the algorithms are presented in Section 3. In Section 4, previous work is discussed and compared, and the contributions of this paper are summarized.

## 2 Model

The system consists of  $D$  loosely-coupled independent disks that can be accessed in parallel. Each disk has a private disk buffer that is used to buffer data that have been read from that disk. In this model we disallow remote buffering, and require that data from a disk can only be stored at the local disk buffer. The benefit of remote buffering due to sharing of the buffers versus the increased communication traffic in moving data between buffers is highly system-dependent, and beyond the scope of this paper.

The smallest unit of disk I/O is a block. In order to amortize the disk seek and rotational latencies over several blocks, I/O is done in units of a *chain*. A chain is a sequence of  $n$  consecutive blocks from a run;  $n$  is called the chain length. In each parallel I/O step at most one chain on each disk can be read. The input consists of  $N$  sorted runs of data,  $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_N$ . A run is a sequence

of sorted elements organized in blocks. In run-level striping all the blocks of a run are placed on a single disk. Assume that the runs are distributed evenly across the disks,  $\rho = N/D$  runs to each disk. We will refer to a run by  $(d, r)$ , where  $d \in \{1, \dots, D\}$  is the disk number and  $r \in \{1, \dots, \rho\}$  is the local number of the run on disk  $d$ . In block-random placement, each chain is placed on one of the  $D$  disks with independent, uniform probability,  $1/D$ . Since successive chains of a run need not all be on a single disk, a pointer to the location of the next chain is maintained with each chain. This information is encoded with the chains at the time the run is created. In addition, in order to implement forecasting in this situation, each chain also holds the data value of the first element of the succeeding chain on that disk. No auxiliary information is necessary for the run-level striped placement.

The standard algorithm to merge the sorted runs works as follows [8]. The first chain of each run is read into buffer memory. The elements of these blocks are merged together in sorted order, until one chain exhausts all its elements. Then an I/O for the next chain of that run is made, and the merge temporarily halts until the I/O is complete. The potential parallelism of the multiple disks is not exploited by this method. In the next two sections we describe how the sequential prefetch and forecasting algorithms respectively increase the I/O parallelism by reading from several disks in parallel.

## 2.1 Sequential Prefetch using Multiple Disks

Sequential prefetch is employed only with run-level striped data placement, and is the traditional prefetch method used in single-disk systems to overlap CPU and I/O operations. This prefetching method employs  $T$ -block read ahead for each run. The parameter  $T$  reflects how much earlier an I/O request is made for a block before it is actually required by the computation. When the number of blocks of a run that have not yet been consumed falls below  $T$ , an I/O request is made for the next chain of that run. The computation proceeds concurrently with the I/O; an I/O is initiated for every run whose number of buffered blocks falls below the threshold. If several such requests are made to the same disk, they are queued and serviced sequentially. Requests made to different disks proceed concurrently. The disk parallelism is influenced by the threshold  $T$ , and the data distribution [10]. The worst-case amount of buffer memory required for each run is  $T + n$  blocks. A common choice of  $T$  that is used is  $T = n$ , whereby we have 1-chain lookahead. In this case a buffer of  $2n$  blocks is required for each run, and an I/O for the next chain of the run is initiated at the time that the first element of a chain is consumed by the merge. We assume  $T = n$  in this paper.

Consider a hypothetical example assuming four runs labeled **A**, **B**, **C** and **D**, and two disks. Runs **A** and **B** are assumed to be placed on disk 1 and runs **C** and **D** on disk 2. Each run is assumed to consist of twelve blocks, organized as four chains of 3 blocks each (see Figure 2(a)). The  $i^{th}$  chain of run **A** is referred to as  $A_i$ ; similarly for the other runs. The numbers within brackets represent the range of data values in that chain; the first (last) number is the value of the first

(last) element of the first (last) block of that chain. Each disk has a buffer of 12 blocks (two chains per run on that disk).

Initially chains  $A_1, C_1, B_1$  and  $D_1$  are read into buffer memory in two parallel I/O steps. When the first element of the first block of a chain is consumed by the merge, an I/O request is made for the next chain from that run. Hence, an I/O request is made to disk 1 for  $A_2$  when the first element of  $A_1$  (1) is consumed, and a request is made to disk 2 for  $C_2$  when the first element of  $C_1$  (3) is consumed. The merge continues until the last element of  $A_1$  (4) is consumed, at which point the merge is suspended temporarily until the I/O for  $A_2$  completes. In the example we have assumed that the CPU takes zero time to consume an element, so that I/Os for both  $A_2$  and  $C_2$  begin at the same time (step 3 in Figure 2(b)). After  $A_2$  is fetched, the merge proceeds until the last element of  $D_1$  (7) is consumed. At this time, I/Os for  $A_3$  and  $B_2$  will be pending at disk 1 (initiated when the first element of  $A_2$  (5) and the first element of  $B_1$  (6) respectively were consumed), and a request for  $D_2$  will be pending at disk 2. Hence  $A_3$  and  $D_2$  are fetched at the I/O at step 4. Once  $D_2$  is fetched the merge resumes, but stalls again on consuming the last element of  $B_1$ , waiting for the I/O for  $B_2$  to complete. Continuing in this manner we see that the entire computation requires 9 parallel I/Os.

<b>A</b>	$A_1 = [1 \dots 4]$	$A_2 = [5 \dots 20]$	$A_3 = [24 \dots 30]$	$A_4 = [31 \dots 33]$
<b>B</b>	$B_1 = [6 \dots 8]$	$B_2 = [10 \dots 15]$	$B_3 = [17 \dots 21]$	$B_4 = [23 \dots 35]$
<b>C</b>	$C_1 = [3 \dots 22]$	$C_2 = [23 \dots 26]$	$C_3 = [27 \dots 29]$	$C_4 = [31 \dots 37]$
<b>D</b>	$D_1 = [5 \dots 7]$	$D_2 = [9 \dots 14]$	$D_3 = [16 \dots 19]$	$D_4 = [28 \dots 45]$

	Step	1	2	3	4	5	6	7	8	9				
DISK1	Run A	$A_1$		$A_2$	$A_3$				$A_4$					
	Run B		$B_1$			$B_2$	$B_3$	$B_4$						
DISK2	Run C	$C_1$	$C_2$						$C_3$	$C_4$				
	Run D		$D_1$	$D_2$		$D_3$	$D_4$							
CPU				$A_1$	$D_1$	$B_1$	$D_2$	$B_2$	$D_3$	$A_2$	$B_3$	$C_1$	$C_2$	$C_3$

Fig. 2. (a) Example Data (b) Trace of Sequential Prefetch

## 2.2 Greedy Prefetching with Forecasting

they will be used. Associated with each disk is a prefetcher whose task is to keep the corresponding disk buffer full by repeatedly fetching the next chain that is required from that disk. Each disk operates independently of the others and greedily attempts to keep its buffer full by prefetching blocks that are deemed

to be “useful”. If “incorrect” blocks are prefetched then the buffer may fill up quickly with “unwanted” blocks, and performance will suffer.

When run-level striped placement is used, one can predict the block on any disk that will be required next by the merge, by examining the blocks from that disk that are currently present in the buffer. For each run from a disk, consider the last block of that run that has already been fetched into the buffer. Call this the *last-buffered* block of the run. The block from that disk that will be required next (called the next-required block) will be from the run whose last-buffered block is exhausted earliest. This can be determined by a comparison between the last elements of each of the last-buffered blocks from the disk. The one with the smallest last element, will be the first to be exhausted; the next block from that run is the one that is required the earliest. Thus, by keeping track of the last-buffered blocks from each run, and comparing the last elements in these blocks against each other, the block to be fetched next from any disk can be determined at run time.

```

Begin Initialization /* (done once only) */
 $\forall d = 1, \dots, D, \forall r = 1, \dots, \rho,$ 
{
  Read first chain of each run  $(d, r)$  into buffer in  $\rho$  parallel I/Os.
  Initialize  $LastElement[d][r]$  to the last element of the last block of the chain for run  $(d, r)$ .
}
 $\forall d = 1, \dots, D, BufferAvail[d] = (\eta - \rho)n;$ 
End Initialization
  Do forever in Parallel for all disks  $i$ 

  {
    /* Buffer manager will release buffers as they are consumed by the merge*/
    Wait until  $(BufferAvail[i] \geq n)$ ;
    /* Find run with smallest last element in the buffer*/
     $run[i] = k$  such that  $LastElement[i][k] = \min\{LastElement[i][j], \forall j = 1, \dots, \rho\}$ .
     $BufferAvail[i] = BufferAvail[i] - n;$ 
    Schedule read of next chain of  $run[i]$  from disk  $i$ ;
    Wait for I/O from disk  $i$  to be completed;
    Let  $B_i$  be the last element of the last block of the chain just read from disk  $i$ .
     $LastElement[i][run[i]] = B_i;$ 
  }
End do parallel

```

**Fig. 3.** Operation of Greedy Prefetcher using Forecasting

Figure 3 presents the operation of the prefetcher in pseudo-code. Each disk buffer is assumed to be of size  $\eta n$  blocks;  $n$  is the chain size, and  $\rho$  is the number of runs on each disk. The variable  $LastElement[d][r]$  is the value of the last

element of the last-buffered block of run  $(d, r)$ . Variable  $BufferAvail[d]$  tracks the number of free blocks of buffer memory for disk  $d$ . After the first chain of each run is fetched,  $n(\eta - \rho)$  free blocks remain in the buffer; these will be used to hold prefetched data. The merge process exhausts a block and requests the next block of that run from the buffer manager. If the block is in the buffer, the buffer manager returns it immediately to the merge process; else the merge halts till the block is read in. The variable  $BufferAvail[d]$  is incremented to reflect the freed buffer. As each I/O completes,  $LastElement[d][r]$  is updated with the value of last element of the last block of the fetched chain of run  $(d, r)$ . The prefetcher for a disk initiates an I/O for the next-required chain whenever the current I/O completes and there is sufficient buffer space for the next chain.

Figure 4 shows the I/O schedule created for the same four runs as in Figure 2(a). As before the CPU time has been shrunk to zero. After the first chain of each run has been fetched (step 2), the last-buffered blocks of disk 1 (respectively 2) are from chains  $A_1$  and  $B_1$  (respectively  $C_1, D_1$ ). Since the last element of  $A_1$  (4) is less than the last element of  $B_1$  (8), the next-required chain from disk 1 is  $A_2$ . Similarly, since  $7 < 22$ , the next-required chain from disk 2 is  $D_2$ . Hence, chains  $A_2$  and  $D_2$  are fetched in a parallel I/O at step 3. The last-buffered blocks on disk 1 now belong to the chains  $A_2$  and  $B_1$ , and the new next-required chain from disk 1 is  $B_2$ . For disk 2, the last-buffered blocks are from chains  $C_1$  and  $D_2$ , and the new next-required chain is  $D_3$ . In the next I/O at step 4, the chains  $B_2$  and  $D_3$  are fetched in parallel. Continuing in this fashion, we can see that all blocks will be fetched in 8 I/O steps, which is the best possible, since 16 chains must be fetched from 2 disks. In contrast, the sequential prefetch scheduling of Figure 2(b) required 9 I/O time steps. Note, also that although in the example zero CPU time was assumed, in this method CPU and I/O fetches are naturally overlapped. The consumption of the in-buffer blocks proceeds concurrently with the prefetch of the next-required blocks from disks.

	Round	1	2	3	4	5	6	7	8
DISK1	Run A	$A_1$		$A_2$			$A_3$		$A_4$
	Run B		$B_1$		$B_2$	$B_3$		$B_4$	
DISK2	Run C	$C_1$					$C_2$	$C_3$	$C_4$
	Run D		$D_1$	$D_2$	$D_3$	$D_4$			
CPU				$A_1$ $D_1$ $B_1$	$D_2$ $B_2$	$D_3$ $A_2$	$B_3$	$C_1$ $C_2$ $C_3$	

Fig. 4. Greedy Schedule with Forecasting

### 2.3 Block-Random Placement

In this section we describe the implementation of block-random placement. A disadvantage of run-level striping as in Figure 2 (a), is its susceptibility to data skew. As an extreme example suppose that runs on disk  $i$  had values that were smaller than those on disk  $i + 1$ ; then clearly the multiple disks provide almost no parallelism (unless the buffer is large enough to essentially hold the entire

data), as data is required from only one disk at a time. Block-random placement provides a method to make the performance independent of the skew in the data.

In this placement method, an entire run is not placed on a single disk. Instead each chain is placed on a disk that is independently chosen at random from the set of  $D$  available disks, with uniform probability  $1/D$  [6]. Each chain maintains a pointer to the next chain of that run. Figure 5 shows a random placement of the four runs of Figure 2(a). The entry in parenthesis  $(d, b)$  is a pointer to the next chain of that run, where  $d$  is the disk number of the next chain and  $b$  the disk block number on that disk. Thus the first chain of run **A**,  $A_1$  is stored on disk 1 at logical block number 1; the next chain of the run is stored on disk 2 logical block number 1. The entry  $(2, 1)$  stored with  $A_1$  is the logical address of  $A_2$ .

In order to implement forecasting with a block-random layout each chain must be further augmented with additional information. In particular, each chain must store the smallest value in the next chain from this run that is stored on the same disk [2]. This information is implanted in a chain during run formation. Thus for instance the smallest value of chain  $B_3$  is stored with chain  $B_1$ , the smallest value of  $C_3$  in  $C_2$ , of  $D_2$  in  $D_1$  and  $D_4$  in  $D_2$ . Similarly for chains on disk 2. The prefetcher maintains a  $D \times N$  table: the  $(d, i)^{th}$  entry indicates the implanted value in the last-fetched block of run  $i$  on disk  $d$ . The prefetcher for disk  $d$  fetches from the run that has the smallest value of the entries for disk  $d$ . After every I/O from disk  $d$  the entry corresponding to the run fetched from is updated.

Blk	1	2	3	4	5	6	7	8
D1	$A_1(2, 1)$	$B_1(2, 4)$	$B_3(2, 5)$	$C_2(1, 5)$	$C_3(2, 7)$	$D_1(1, 7)$	$D_2(2, 8)$	$D_4(null)$
D2	$A_2(2, 2)$	$A_3(2, 3)$	$A_4(null)$	$B_2(1, 3)$	$B_4(null)$	$C_1(1, 4)$	$C_4(null)$	$D_3(1, 8)$

**Fig. 5.** Block-Random Placement

### 3 Simulation Results

In this section we study the performance of different prefetching and data placement policies using a process driven simulator, YACSIM [5]. We introduce a Markov model for generating the access pattern. We call the correlation among the accesses *skew*. We then look at the performance of the different prefetching policies and the different data layout strategies, as the skew of the input data is varied.

The performance of a prefetching algorithm is measured by the time it takes to service a specified sequence of I/O requests. In analytic studies of the parallel disk model [13, 18] the usual measure used is the total number of parallel I/Os required to service the request sequence. However in practice the time taken to fetch a block is not constant and depends on several factors: the load, the data placement, and access time variations due to the physical disk geometry, for

instance. Incorporating all these factors together in an analytic model is generally intractable, and their effect on performance is best evaluated empirically.

To compare the performance of different policies we determine using simulation the total I/O time for a sequence of I/O requests; this will also be called the execution time of the algorithm. We also measure the average disk parallelism achieved by the algorithms and correlate the two. Parallelism refers to the average (over time) number of disks which are busy at any time. A high value of parallelism would indicate that the prefetching strategy is good as it means that fewer number of I/Os need to be performed to fetch all the blocks required in the merge.

Sector Size	256 Bytes
Track Per Cylinder	8
Sectors Per Track	113
Rotational Speed	4002 RPM
Track (Head) Switch Time	2.5ms
Transfer Rate	4Mbps
Seek Time (d in cylinders)	
Short	$3.45 + 0.597\sqrt{d}$ ms
Long	$10.8 + 0.012d$ ms
Boundary	616 cylinders

**Table 1.** Parameters of the Disk Simulated

### 3.1 Disk Model

For the simulation study we use the disk model detailed in [14]. The parameters of the modeled disk are presented in Table 1. The disk is modeled to use one read head per platter, all mounted on a synchronized spindle, so that at any time all heads access the same cylinder. To accurately model seek times, a nonlinear seek model is used: seeks are modeled as very short seeks (0 or 1 cylinder), short seeks (less than boundary) and long seeks (greater than boundary). When the track sought is in the same cylinder as the current location of the read head, only a head switch is required to get to the new track. If the track accessed requires a move to an immediately adjacent cylinder, then the time taken is the track switch time. It is only when the track being accessed is more than 1 cylinder away that an actual seek is performed, and the time taken for the seek depends upon the number of cylinders traversed. The time for a short seek is dominated by the acceleration and deceleration of the head and hence varies as the square root of the distance. During a long seek, considerable time is spent moving at the maximum velocity and hence the time varies linearly with distance. We model the rotational latency by using a random variable uniformly distributed over the time taken to complete one rotation. Finally the transfer time is modeled by the time taken to read data off the disk at the constant transfer rate.

Seek and rotational latency constitute a significant portion of the overall access time and are incurred each time an I/O is performed. A well-known method to reduce the average access time is to amortize this overhead over several blocks, by chaining together requests to physically contiguous blocks which are fetched in a single I/O operation. In this way repeated seeks are avoided when the data being fetched reside contiguously on disk. Since runs are read sequentially, chaining is easy to implement with run-level striped layout by simply fetching several blocks of data from a run in each I/O. In other situations, chains of data blocks must be explicitly written out to each disk during the run creation phase, to facilitate chaining during read-back. The performance benefit of chaining diminishes beyond a certain chain length, as the transfer time becomes the dominant cost. Moreover the size of the I/O buffer required increases, to account for the fact that in each I/O multiple blocks of data are fetched. Our studies on the disk simulated show a benefit of chaining till around 10 4KB blocks, beyond which the marginal improvement due to chaining was small [6]. Based on this we used a chaining of 10 in all our simulation studies.

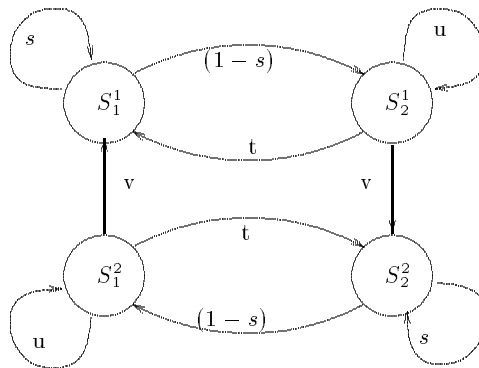


Fig. 6. Two-State Skew Model

### 3.2 Skew Model

In general, during the merge blocks are not requested uniformly from all disks. In skewed data, there is often a sizable correlation between the run whose block has been consumed and the run from which the next block will be consumed. The more skewed the data, the higher the correlation. We now introduce two models of skew used in our simulations.

The first model is the one-state model, parameterized by the quantity  $s$ , that controls the *skew*. Assume that at some instant, the last block that was consumed was from run  $r$ . In this skew model, the next block to be consumed is from the same run  $r$  with probability  $s$ ; the probability of the next block being from any other run is uniformly  $(1-s)/(N-1)$  where  $N$  is the total number of runs being merged. A state transition diagram for this model consists of  $N$  states, one for

each run. A state corresponds to the run from which the last block was consumed. The self-loop transition indicates that the next block is consumed from the same run (this occurs with probability  $s$ ), and there are transitions to each of the other  $N - 1$  states, each with probability  $(1 - s)/(N - 1)$ . The one-state model while simple and intuitive, imposes a sharp switch of locality between runs. With high skew, the average number of consecutive blocks requested from a run is very high.

A more elaborate model is required in cases where the average number of consecutive blocks requested from any particular run is not high, but in any set of consecutive requests there tend to be a lot of blocks from a particular run. This model is an extension of the one-state model. The intuition of this model is that it may be possible that the next block be requested from a different run, but with a high probability the total number of blocks requested from a particular run is large in a small window of consecutive requests. This is referred to as a two-state model.

Informally, in the two-state model at any time one run is called the *stuck* run. All other runs are said to be *unstuck*. The stuck run behaves as in the one-state model: if the last block consumed was from a stuck run, the next block is consumed from it with probability  $s$ , and from a different unstuck run, with uniform probability  $(1 - s)/(N - 1)$ . When the last block consumed is from an unstuck run, the next block consumed is from the current stuck run with probability  $t$ , and the same unstuck run with probability  $u + v$ . In the latter case, this run may become the current stuck run with probability  $v$  (the previous stuck run becoming unstuck), or it may remain unstuck with probability  $u$ .

In general for  $N$  runs, there will be a total of  $N^2$  states in a two-state model:  $S_i^j, 1 \leq i, j \leq N$ . State  $S_i^j$  represents the state when the last block consumed is from run  $i$  and the current stuck run is  $j$ .  $S_i^i$  is the state where the last block consumed is from the stuck run  $i$ . Each  $S_i^i$  has a self-transition with probability  $s$ , and a transition to state  $S_j^i, i \neq j$ , with probability  $(1 - s)/(N - 1)$ . The latter state,  $S_j^i$ , has a transition back to  $S_i^i$  with probability  $t$ , a self-transition with probability  $u$ , and a transition to  $S_j^j$  (making  $j$  the stuck run) with probability  $v$ . Typically  $t$  is set high to indicate a high probability of requesting the next block from a stuck run. For our simulations we set  $t = 0.8$  and  $u = v = 0.1$ . A state-transition diagram for the two-state model assuming  $N = 2$  is shown in Figure 6.

### 3.3 Performance Comparison

We first present the simulation study on the performance of the greedy forecasting algorithm. This policy is compared to a naive method (oblivious policy) which randomly chooses the next run from which to prefetch. In contrast to the forecasting method that always prefetches the next-required block from a disk, the oblivious policy prefetches the next unread block from a randomly-chosen run on that disk. Figure 7 shows the variation of the normalized number of I/Os performed by the oblivious and greedy forecasting policies against the size (in

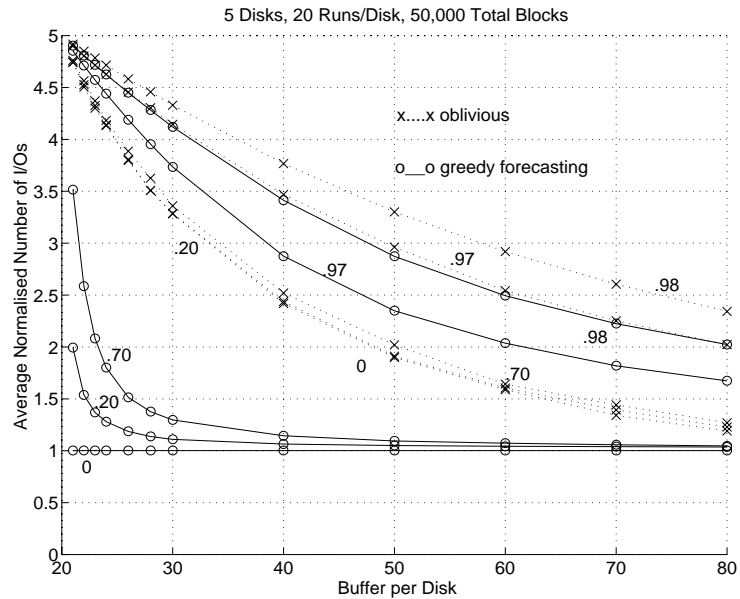


Fig. 7. Oblivious vs. Greedy Forecasting Policies

blocks) of each disk buffer. The normalized number of I/Os is the ratio of the total number of I/Os performed to the minimum number of I/Os that would be required if each I/O fetched one block from every disk. The simulation was performed using 5 disks, 20 runs per disk, and merging a total of 50,000 blocks of size 4KB each. The data was modeled using the one-state model of skew; and the figures show the variation of the normalized number of I/Os for both the policies for different values of the skew parameter  $s$ .

As the buffer size increases, the average number of blocks that can be prefetched during each I/O increases and hence the total number of I/Os decreases for both policies. Also as the skew increases the probability of requesting several contiguous blocks from the same run (and hence the same disk) increases causing the I/O parallelism to fall. This is manifested as an increase in the total number of I/Os with increasing skew, in both cases. However for any given skew and buffer size, the forecasting policy performs fewer number of I/Os than its oblivious counterpart. This is because during any I/O, the forecasting policy exploits its ability to accurately predict which block will be requested next; however the oblivious policy by making a guess at the next block, can fill up the buffer with useless blocks degrading the parallelism. Also note the rather stable performance of the forecasting policy for low and medium skews. This fact will be illustrated in greater detail later. In all ranges of the skew and buffer size the forecasting policy outperforms the oblivious policy. As the available buffer size increases, both policies approach the ideal minimum number of I/Os, for low and

medium skew. However, the forecasting policy achieves good performance even with a relatively small number of buffer blocks. Thus it is always beneficial to use forecasting to decide which block to fetch. We next compare the performance

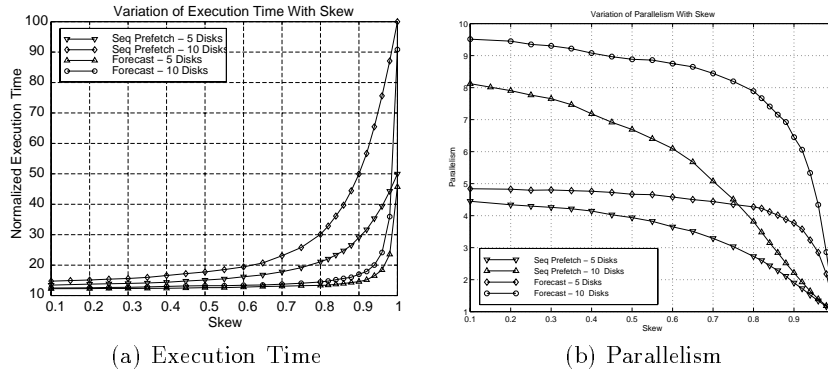


Fig. 8. Comparison of Sequential Prefetch and Forecasting

of the greedy forecasting algorithm with the sequential prefetch policy. We simulated systems consisting of 5 and 10 disks and examined the total execution time taken by the two policies to service the I/O requests. The load was kept at a constant 5 runs per disk, with each run consisting of 1000 blocks of size 4KB each. Since the performance of both the policies can be improved by chaining, both the policies use chains of 10 blocks. The trigger,  $T$ , of the sequential prefetch policy is set to the chain length, 10. Hence the buffer per disk needs to be 20 blocks per run; and thus each disk was simulated with a buffer of 100 4KB blocks. Skew was modeled using the one-state model.

Figure 8(a) gives the variation of the total execution time with data skew for both the policies, in systems consisting of 5 and 10 disks. The execution time has been normalized by the execution time of sequential prefetch in a system consisting of 10 disks, when data skew is 1. The total execution time for sequential prefetch policy is always higher than the corresponding time for the greedy prefetch policy. This is because the greedy forecasting policy makes more effective use of the available buffer by allocating it judiciously among the runs on that disk. The performance of both policies degrade with increasing skew but the sequential prefetch policy is more sensitive to skew. While the increase in execution time of the greedy forecasting is noticeable only beyond a skew of 0.85, the sequential prefetch policy starts degrading even at skew of 0.7. For skew greater than 0.7 the performance gap between the two widens considerably. Finally at very high skew close to 1, there can be no parallelism as blocks are consumed a disk at a time. This causes the performance of both algorithms to be equally bad at such high skew. The sensitivity of sequential prefetch to skew in general gets worse as the number of runs on a disk (load) increases.

The variation in execution time can be seen to mirror the variation in I/O

parallelism with skew, as shown in Figure 8(b). It is interesting to note that with a skew of 0.75 or higher the average parallelism achieved by the sequential prefetch policy on a system with 10 disks is less than the average parallelism attained by greedy forecasting on a system with 5 disks.

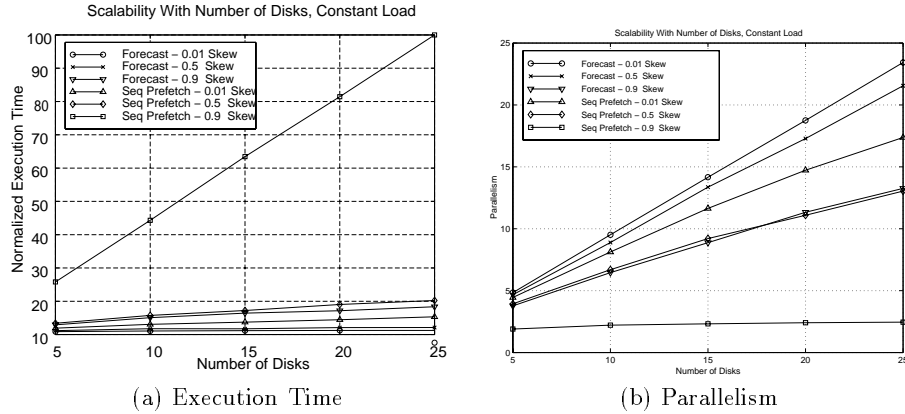
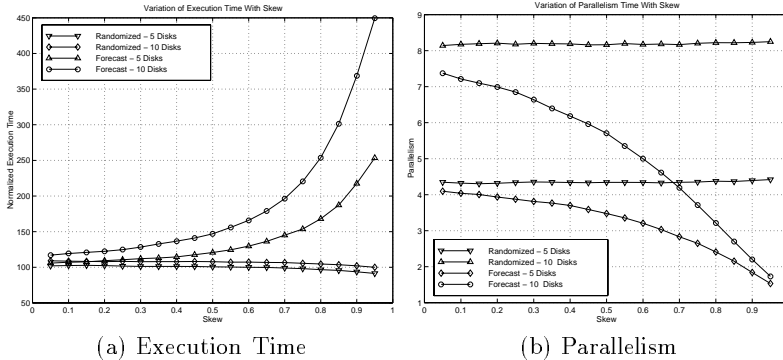


Fig. 9. Scalability of Sequential Prefetch and Forecasting

In addition to the variation of execution time with skew, it is also interesting to see the scalability of the prefetching policies with the number of disks in the system. To do so we simulated a merge consisting of 5 runs per disk (to keep the load per disk constant) with each run having 1000 blocks. The buffer at each disk was of size 100 blocks, and the chain length was 10 blocks. Simulations were performed for both sequential prefetch and greedy forecasting policies on systems consisting of different numbers of disks. Skew was modeled using the one-state model.

Figure 9(a) presents the variation in the execution time with the number of disks, for the two policies and three sample skews. Again the execution time has been normalized by the execution time of sequential prefetch in a 25 disk system with data skew of 0.9. For low (0.01) and medium (0.5) skew, the policies scale very well, with forecasting being consistently superior. Note that since the load per disk is constant the overall execution time ought to be constant with increasing number of disks for linear scalability. However for highly (0.9) skewed data, the sequential prefetch performs very badly. This can also be seen more pointedly in Figure 9(b), that shows the variation of parallelism with increasing number of disks. The parallelism attained by sequential prefetch when data skew is 0.9 is a constant! In contrast greedy prefetching is much more robust and even with data skew of 0.9 its parallelism scales as well as that of sequential prefetch with almost half (0.5 vs. 0.9) the amount of skew.

Finally, we examine the performance of the greedy forecasting policy under the two different data placements: Run-Level Striping and Block Random. As



**Fig. 10.** Performance Comparison of Run-Level Striping Layout and Random Block Layout

before we have 5 runs per disk with each run having 1000 blocks. The buffer was of size 100 blocks per disk and I/Os fetched blocks in chains of length 10 blocks. Skew was modeled using the more detailed two-state model.

Figure 10(a) shows the plot of the normalized execution time as the data skew is varied, in 5 disk and 10 disk systems, for both the data layout strategies. As expected, placing data blocks randomly makes the performance of the policy completely independent of the data skew, even while the striped layout deteriorates rapidly with data skew. This fact is confirmed in figure 10(b) which shows the variation of parallelism against the data skew. While the parallelism obtained in the random layout case is independent of data skew, it is drastically affected by striped layout. This demonstrates the advantage in combining forecasting with random data layout. Not only is the execution time lower than the corresponding striped layout, but also the performance of the system is virtually independent of data skew.

## 4 Discussion

There has been a substantial amount of previous work on the design of I/O-complexity optimal external sorting algorithms using multiple disks [1, 11, 12, 17, 18]. These include methods based on distribution sorting: both randomized [18] and deterministic [12], and variations of merge sorting [1, 11, 17]. These algorithms are of important theoretical significance, being the first, and in some cases the only, I/O-optimal algorithms known for these problems. However, the constants involved with some of these solutions can be relatively large, and no experimental study of their performance has been reported.

More recently, in [2], a randomized mergesort was proposed which performs an expected  $\Theta(N/D)$  I/Os to merge  $N$  data blocks. The algorithm is for a global buffer shared among the disks rather than the distributed disk buffers used in this paper. Theoretical complexity results on randomized merging for the distributed buffer model were presented in [3]. The forecasting method used in the run-level

striped placement method generalizes the ideas in Knuth [8]; the block-random placement described here was introduced and analyzed in [3, 6]; the implanting of forecasting information was suggested in [2]. Once again there has been no previous performance study of these algorithms relative to traditional methods.

The use of prefetching to speed up parallel I/O was addressed in [13] in the context of external merging using a global buffer, and two related prefetching algorithms were analyzed and compared relative to each other. Prefetching algorithms using sequential prefetch in a multiple disk environment were studied in [10] using Markov analysis for small configurations and simulation for larger systems. Both of these works assumed uniformly-random data (no skew) in the evaluation. Off-line prefetching algorithms in the global and distributed models were presented in [7] and [16] respectively. Competitive analysis of on-line prefetching with bounded lookahead has been analyzed in [3, 6] for the global buffer model, and [16] for the distributed buffer model.

Other works on improving the performance of external sorting and merging using a single disk system include [8, 9, 15, 19]. In particular, [19] used an off-line version of forecasting to determine the read order on a disk to minimize seek time. In the model of [15], it was argued that large block sizes and double buffering resulted in the best I/O performance for a single disk system.

In this paper we have presented an empirical evaluation, using simulations, of several prefetching policies to improve the I/O performance of external merging using parallel I/O. In particular we considered sequential prefetch used in many traditional systems today, forecast-based greedy prefetching, and oblivious prefetching (random choice of run to prefetch). In conjunction with the prefetching policy we evaluated the benefit of two different data placement strategies: run-level striping and block-random placement. We also introduced a method to model correlation in the data (skew), and evaluated the performance of these methods as the data skew varied.

Based on our study we can conclude that the I/O performance can be improved significantly by using forecasting techniques to determine the block to prefetch from any disk. Not only does this method result in better disk utilization compared to both sequential prefetch and oblivious schemes, but it also is less sensitive to variations in the data skew. Further it is shown that the performance of even the forecasting-based prefetching policy degrades when the skew gets sufficiently large. To maintain good performance in the presence of skewed data, a randomized data placement scheme was shown to be very useful. With such a block-random placement scheme and a forecasting-based greedy prefetch algorithm, the performance of external merging remains uniformly good independent of skew in the data, and scales well with increased numbers of disks.

## 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. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Journal of Parallel Computing*, 1997. Special Issue on Parallel I/O, To appear.
3. R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive Parallel Disk Prefetching and Buffer Management. In *5th Annual Workshop on I/O in Parallel and Distributed Systems*. ACM, November 1997. (Preliminary version in ACM SPAA'97 Revue).
4. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, 1994.
5. J. R. Jump. YACSIM: Reference Manual, Ver 1.1 ECE. Technical report, Rice University, April 1992.
6. M. Kallahalla. Competitive Analysis of Buffer Management for Parallel I/O. Master's thesis, Rice University, 1997.
7. T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *37th Annual Symposium on Foundations of Computer Science*, pages 540–549. IEEE, October 1996.
8. D. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
9. S. C. Kwan and J. L. Baer. The I/O Performance of Multiway Mergesort and Tag Sort. *IEEE Transactions on Computers*, 34(4):383–387, 1985.
10. K. K. Lee and P. Varman. Prefetching and I/O Parallelism in Multiple-Disk Systems. In *Proc. 24th Intl. Conference on Parallel Processing*, 1995.
11. M. H. Nodine and J. S. Vitter. Large-Scale Sorting in Parallel Memories. In *Proc. 1991 ACM Symposium on Parallel Algorithms and Architectures*, 1991.
12. M. H. Nodine and J. S. Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *Proc. 1993 ACM Symposium on Parallel Algorithms and Architectures*, 1993.
13. V. S. Pai, A. A. Schäffer, and P. J. Varman. Markov Analysis of Multiple-Disk prefetching Strategies for External Merging. *Theoretical Computer Science*, 128(1–2):211–239, June 1994.
14. C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, 1994.
15. B. Salzberg. Merging Sorted Runs Using Large Main Memory. *Acta Informatica*, 27(3):195–215, 1989.
16. P. J. Varman and R. M. Verma. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Proceedings of 1996 Symposium on Foundations of Software Technology and Theoretical Computer Science*. LNCS, Springer Verlag, December 1996.
17. J. S. Vitter and M. H. Nodine. Large-Scale Sorting in Uniform Memory Hierarchies. *J. Parallel and Distributed Computing*, 17:107–114, 1993.
18. J. S. Vitter and E. A. M. Shriver. Optimal Disk I/O With Parallel Block Transfer. In *Proc. 22nd ACM Symposium on Theory of Computing*, pages 159–169, 1990.
19. L. Zheng and Per-Åke Larson. Speeding up External Mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, April 1996.