
PLACEMENT-RELATED PROBLEMS IN SHARED DISK I/O

J.B. Sinclair, J. Tang* and P.J. Varman

Department of ECE, Rice University, Houston, Texas 77251

** Informix Software, Menlo Park, California 94025*

ABSTRACT

In a shared-disk parallel I/O system, several processes may be accessing the disks concurrently. An important example is concurrent external merging arising in database management systems with multiple independent sort queries. Such a system may exhibit instability, with one of the processes racing ahead of the others and monopolizing I/O resources. This race can lead to serialization of the processes and poor disk utilization, even when the static load on the disks is balanced. The phenomenon can be avoided by proper layout of data on the disks, as well as through other I/O management strategies. This has implications for both data placement in multiple disk systems and task partitioning for parallel processing.

1 INTRODUCTION

Concurrent access to a shared parallel I/O system by multiple processes raises new and interesting problems that impact system performance. One example of such a system is several *independent* and *concurrent* jobs (either multiprogrammed or on separate CPUs) accessing data on a set of shared disks [6]. Another is a parallel processing environment where subtasks of a single computation share data on multiple disks. The behavior of such systems is complex and difficult to predict due to the interaction of several factors. The sequence of logical block accesses made by a process depends on the underlying computation and in general on the data. The physical disk accesses depend on the data mapping or *placement policy* which assigns logical blocks to actual disks and locations on the disk. The *disk scheduling policy* influences the sequencing of requests from different processes queued at a disk, resulting in differing I/O

interactions. Finally, *buffer availability* affects the progress of the computation and influences the actual (run time) request sequences seen at a disk.

In this article we use the example of multiple, independent, external-merge jobs as a case study for the investigation of the performance of parallel I/O systems. We compare the effects of different data placement policies on the completion times of the jobs. We show how intuitively reasonable *statically* load-balanced data placement policies can demonstrate quite unexpected *dynamic* behavior which results in large slowdowns caused by serialization of the disks. Furthermore, such systems demonstrate anomalous behavior with respect to buffer availability, as performance degrades with increasing numbers of buffers. Appropriate disk scheduling policies are shown to alleviate this loss of performance. We develop and analyze a model to explain this behavior. The model's predictions closely match those obtained from simulation.

There has been much recent attention on speeding up I/O for specific computational problems using new I/O-efficient algorithms [1, 2, 3, 4, 7], or efficient prefetching and disk scheduling [5, 8, 14]. The usual model employed consists of a single computational process, a fixed amount of main-memory buffer, and a set of multiple, independent disks. A block can be fetched (written) from (to) each of the disks in one parallel read (write) operation. From the viewpoint of I/O operations, the computation essentially remaps the data (possibly after some transformation) among the multiple disks. The objective is to use the smallest number of parallel I/O operations, subject to the given buffer size, or equivalently to maximize the I/O parallelism for a given buffer size.

Although a single-process abstraction is appropriate in many contexts, a multi-process model is needed to understand and optimize system behavior in the application mentioned above. Similar considerations can be expected to arise at the operating system level in new parallel file systems like Vesta [9], which provide shared parallel disk and buffer services to multiple processes. The interaction of multiple processes often results in unexpected behavior resulting in severely degraded performance. While some earlier studies (e.g., [4]) did consider multi-level memory using multiple processors, those models are closer to a shared-nothing than to the shared-disk architecture studied here [10].

One consequence of our study of shared-disk parallel I/O systems with concurrent processes is the identification of a set of conditions under which the system can exhibit instability. By instability we mean the development of a *race* among the processes, with the winner(s) monopolizing the system resources and making progress, while all other processes come to a virtual halt. The effect is to serialize the usage of disks, degrading I/O parallelism and disk utilization.

This happens even though the load is symmetrically distributed and balanced among the disks. In the worst case, this results in a slowdown proportional to the number of disks; i.e., there is almost no performance gain from using multiple disks.

The results have implications for both independent jobs and parallel programming situations. In cases where the allocation of data to disks can be planned, the problem can be avoided using appropriate data placement that prevents the race from developing. Alternatively, disk scheduling or buffer management strategies that block the development of the race can be employed. In the parallel programming case, the application usually has control over the number and types of subtasks generated. The possibility of instability implies that task partitioning based simply on dividing the I/O load evenly among tasks, may be insufficient to obtain good run-time behavior due to serialization of the I/O.

The rest of this article is organized as follows. In Section 2 we describe external merging and in Section 2.1 the model of the system and the external merge algorithm used. Section 2.2 describes different data placement policies for multiple independent external merges, and their performance is compared using simulation. Placements resulting in job racing referred to earlier are identified, and used as the basis for the analysis of unstable behavior. Two related problems are considered in Sections 3 and 4. The first deals with a two-disk two-task system; each task makes a copy of a file on one disk on the opposite disk. The system is analyzed and the potential instability in a straightforward file-copying application is identified. In Section 4 the more complex situation that describes external merging is analyzed, and the conditions for unstable behavior are determined. Possible solutions to instability based on disk scheduling and buffer management are described in Section 5. The key features of the article are summarized in Section 6.

2 EXTERNAL MERGING

Multiple external merging arises in database management systems executing multiple independent sort queries. Since the data is usually too large to fit in main memory, an external sorting algorithm (usually external merge sort [15]) is employed. A job accesses its data from the database in batches. Each batch is sorted in main memory and the sorted run written out to a temporary file on an auxiliary set of disks. After all the data has been exhausted, the multiple runs that make up a data set are merged together into a single output run

using an external merging algorithm [15]. In keeping with the design of current high-performance database systems like DB2, SQL DS, etc., each run is placed entirely on a single disk without striping.

A random-block-depletion model is used to model the merge [12]. Each merge job chooses the leading unused block of any of its runs with uniform probability. The CPU depletes this block, generates a block of output, and requests the next block from that input run. If that block is in the cache, the request is immediately satisfied; otherwise, the CPU is blocked until an I/O fetch for that block is completed.

2.1 System Model

The system model consists of j concurrent processes or jobs, an I/O subsystem with d independent disks D_0, D_1, \dots, D_{d-1} , and a disk cache (disk buffer). Each process performs a sequence of reads and writes based on the merging model described above. A process suspends on reads, waiting for all the reads issued to be completed before progressing. Writes are asynchronous; a process continues after issuing a write request.

Data is stored on the disks and in the cache in *blocks*. A block is the smallest unit of I/O, often a 4 Kbyte page in commercial systems. The average times to read a block from and write a block to a disk are r and w , respectively. r and w depend on several hardware parameters of the disk that determine the seek, rotational latency and transfer time, as well as logical parameters such as the data layout, access patterns and prefetching employed. Physically consecutive blocks can be read or written together, reducing the overhead per block. The *read blocking factor* b_r and *write blocking factor* b_w indicate the number of blocks read or written together.

In many commercial systems, I/O is initiated only when the blocks in memory for a run have been exhausted. In this strategy, known as *demand I/O*, jobs are delayed on every read access. *Prefetching*, a method in which I/O operations are initiated before they are needed, allows parallelism in the disk accesses. Specifically, the model uses anticipatory *intra-run prefetching*. On each I/O request, b_r contiguous blocks of data are fetched from a run. When the first block of an b_r -block prefetch is consumed, the next b_r -block prefetch from the same run is initiated. Also each time a block is consumed, a block of data to be written to a disk is generated and buffered in the cache. When b_w write blocks have been buffered, a write operation is initiated.

If a free cache block is unavailable when requested, a job waits until one is released. We assume a large (or unlimited) cache size to concentrate on specific issues of data placement on performance. The completion time for a job is the simulation time at which all writes for that job are done, and the completion time for a set of jobs is the completion time of the job that finishes last.

To focus on the I/O performance, the model assumes infinite-speed CPUs. Additional simulation verifies that the behavior with finite-speed CPUs is essentially unchanged as long as the entire system is not CPU-bound. Requests are queued at the disk and unless otherwise stated are serviced using an FCFS policy. For simulation, a linear seek model with seek time of 0.04 ms/cylinder, average rotational latency of 8.33 ms, and block transfer time of 1.024 ms/block were used. Unless noted otherwise, a job has 20 runs, each of which contains 1000 blocks, and each block has 4K bytes. The read and write blocking factors are $b_r = 12$ and $b_w = 40$, respectively.

2.2 Performance of Placement Policies

Fig. 1 illustrates four possible run allocation policies for the case when the number of jobs is no more than the number of disks (*i.e.*, $j \leq d$). Each job has some number of input runs (indicated by Read) and a single output run (indicated by Write). No disk will hold more than one output run. To utilize all disks efficiently the I/O load should be divided as equally as possible among the d disks. Since every output run is placed on a separate disk, we need only consider the placement of the input runs. A *read (write) disk* is used only for input (output); a *read/write disk* is used for both.

- **Policy 1:** *Dedicated Write Disk for Each Job.*

j of the disks are used as write disks and the remaining $d - j$ as read disks. Disk D_k , $0 \leq k \leq j - 1$, is used exclusively for the output run of job k ; the input runs of each job are spread evenly among the remaining read disks.

- **Policy 2:** *Intra-job Separate Read and Write Disks*

Each job uses $d - 1$ disks for input and the remaining disk for output. Job k , $0 \leq k \leq j - 1$, uses disk D_k for its output run, and its input runs are spread evenly among the remaining disks. Thus, there will be $d - j$ read disks and j read/write disks.

- **Policy 3:** *Intra-job Shared Read and Write Disks*

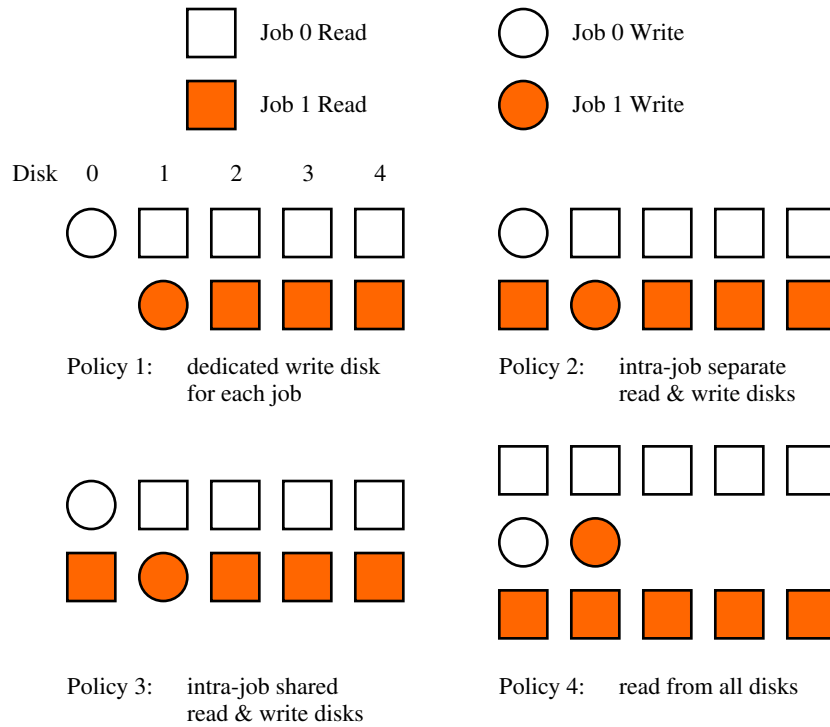


Figure 1 Run placement policies for a system with 2 jobs and 5 disks

This allocation is obtained by beginning with the allocation of Policy 2 above, and then permuting the input runs on the disks as follows. The input runs of job k , $0 \leq k \leq j - 1$, are moved from disk $D_{(k-1) \bmod j}$ to disk D_k . As in Policy 2, there will be j read/write disks and $d - j$ read disks. However, unlike Policy 2, each read/write disk also has input runs of the *same* job that read from that disk.

■ **Policy 4: Read from All Disks**

Each job uses all d disks for input and one disk for its output run. There are j read/write disks and $d - j$ read disks.

The performance of these policies, determined by simulation, is shown in Fig. 2.

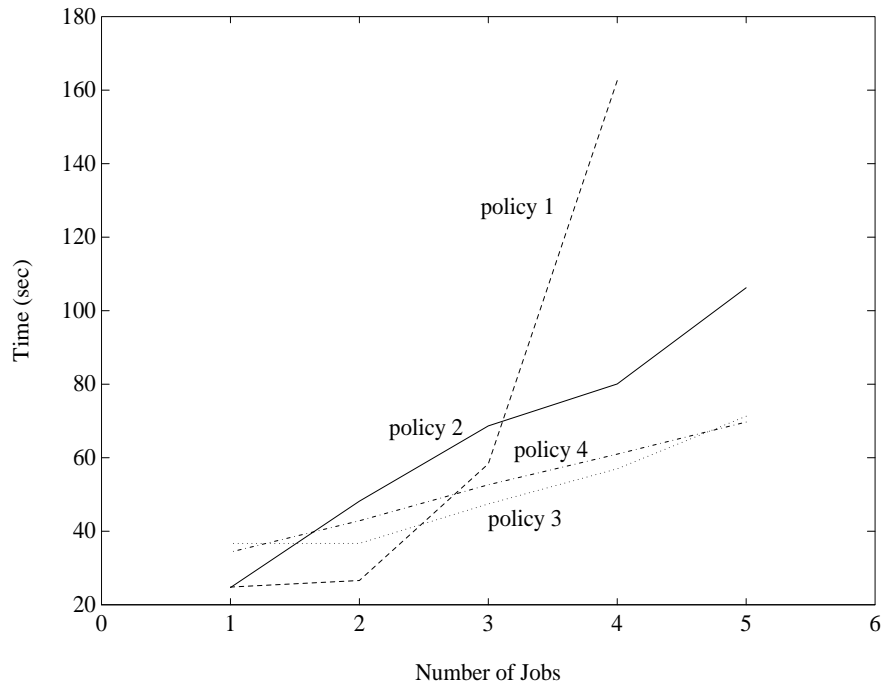


Figure 2 Completion time of all jobs for $1 \leq g \leq 5$ for the placement policies described in Fig. 1 with 5 disks

Policy 1 is motivated by the recognition that the write of an output run would tend to be the performance bottleneck for a small number of jobs. Consequently, each disk that is allocated an output run is not loaded any further. For 1 and 2 jobs, the total time matches the time required for writing an output run. As the number of jobs increases, the number of disks allocated for reads decreases, and the load on read disks begins to exceed that of write disks. As may be seen, the performance rapidly deteriorates once the system becomes input-bound.

The allocation in Policy 2 attempts to preserve the best-case performance of a single job (that of Policy 1), with reasonable performance as the number of jobs increases. It also has the advantage that the allocation for a job is independent of the allocation of other jobs. Consequently, such a policy can be easily implemented in the case of staggered job arrivals. An important

fact to be noted in Fig. 2 is that the increase in the time for Policy 2 as the number of jobs is increased *cannot* be accounted for merely by the increase in the load on each disk. Even though there is perfect symmetry in the placement of runs for each job, jobs progress at different rates, with a significant amount of serialization among all jobs, which degrades the performance. The reason for this racing behavior will be discussed later.

Policy 3 presents one method that controls the racing inherent in the Policy 2 layout. The input runs of each job are permuted so that every output disk also contains input runs from the *same* job. Fig. 2 shows the significant performance improvement of Policy 3 over that of Policy 2. For 5 jobs, the completion time for Policy 2 is 105 seconds, while the time for Policy 3 is 70 seconds. (The time difference increases as the data size increases.) One disadvantage of Policy 3, however, is that the policy cannot be applied in the case of staggered job arrivals; the number of jobs in the system must be known prior to laying out data on disks.

Policy 4 is straightforward. All input runs are distributed evenly on the set of d disks. There is at most one output run on any disk. Although it never performs as well as Policy 3, its results are at most 10% worse. Policy 4 has the advantage of accommodating staggered job arrivals. All input runs of a new job are evenly divided on the set of d disks; its output run can be placed on any disk without an output run.

The underlying reason for the race behavior is the difference in the rates of read and write service (since reads are spread across multiple disks). Intuitively, one job (say job A) gets slightly ahead and places some number of write requests in the queue for its output disk. This slows down all the other jobs since they have input runs on that disk, but not A since it does not have any input on that disk. The delay for reads at this disk decreases the demand on the other $d-1$ disks, allowing job A to get further ahead in its reads. In turn, job A will generate even more write requests at its output disk, slowing down the rest of the jobs even further. Eventually, only job A is progressing, and all other jobs come to a virtual halt, waiting for read service at job A 's output disk. When A completes, the remaining jobs race against each other, and this pattern repeats.

As these results show, the choice of an appropriate placement policy depends on the values of j and d , and the job arrival pattern. When $j < \frac{d}{2}$, Policy 1 provides the best performance. The placement of input and of output runs is relatively balanced (see Fig. 1). With prior knowledge of j and $j \geq \frac{d}{2}$, Policy 3 should be used for run placement. Policy 4 is the most appropriate when there

are staggered job arrivals in the system, as it consistently provides reasonable, albeit suboptimal, performance. For more details, see [11].

3 ANALYSIS OF A 2-DISK SYSTEM

Consider the following simple example of a two-disk, two-process system. The input consists of $2N$ blocks of data divided equally and placed on physically consecutive blocks of the disks D_0 and D_1 . We refer to the blocks on D_0 and D_1 by $(0, i)$ and $(1, i)$, $1 \leq i \leq N$, respectively. The problem is to permute the data so that block (j, i) is moved to $((j + 1) \bmod 2, \pi(i))$, for some permutation π on $\{1, \dots, N\}$. That is, block i on disk 0 (disk 1) is moved to block $\pi(i)$ on disk 1 (disk 0).

In the framework of [1, 2, 7] the permutation problem posed above has a straightforward solution. It can be performed using an optimal number of parallel I/O's and two blocks of buffer storage. Blocks i from each of the two disks are read in one parallel access, and are then written out in parallel to $\pi(i)$ on the opposite disks from which they were read. The total number of parallel I/Os is $2N$, which is optimal (since if all data were on a single disk, $4N$ block accesses would be required). The time required is $N(r + w)$, which is a lower bound on the time required by any solution. In an environment where read and write times are nondeterministic, greater flexibility and (usually) better performance can be obtained by decoupling the parallel accesses; i.e., accesses at one disk proceed independently of accesses at the other disks.

3.1 Simulation Results

We simulated this 2-disk system with π as the identity permutation, making the problem simply to copy each of the files from the disk on which it initially resides to the other disk. Each of the two processes P_0 and P_1 reads 10000 blocks from a disk and writes them to the other disk. b_r is fixed at 8 while b_w is varied. Fig. 3 shows a plot of the time for both processes to complete as b_w is increased (i.e., w is decreased). It also shows the lower bound $N(r + w)$. Unexpectedly, the time for completion of P_0 and P_1 diverges significantly from that predicted by the load on the disk, especially as w increases, indicating that in the simulated system, the disk utilizations must have been significantly less than 100%. In fact, when $b_w = 1$, the completion time is almost 60% worse

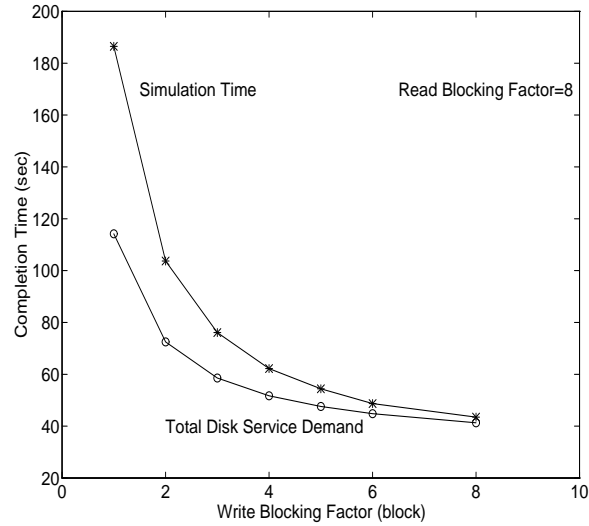


Figure 3 Performance of the two-disk system

than expected (190 vs 115 seconds). In the following section, we analyze this system and explain the loss in disk parallelism and performance.

3.2 Analysis

The loss in disk concurrency in the system is caused by the serialization of the processes P_0 and P_1 . Due to the nondeterministic nature of the read and write times, one of the processes (say P_0) gets a little ahead of the other. That is, the queue of pending writes at D_1 (the disk to which P_0 writes) is longer than the queue at D_0 . Since disk scheduling is FCFS, P_1 is delayed until all these writes are serviced. During this time P_0 continues to read from D_0 , and send more write requests to D_1 . These queue up behind the read request of P_1 already in the queue. If the average block read time is smaller than the average block write time, then by the time P_1 's request reaches the head of D_1 's queue and is serviced, P_0 has put an even larger number of write requests into the queue at D_1 . Hence P_1 's next read is delayed even longer.

To simplify the analysis, we assume fixed times for reads and writes. Since a job suspends on reads, there can be at most one read request pending at any disk queue. Let $f = w/r$ be the ratio of the average write and read block access times. Assume that at time $t = 0$, D_0 has only one read request pending, while D_1 has a read request and N_0 write service demand pending¹. $T_1 = N_0 + rb_r$ is the time required for D_1 to service every request in the queue at $t = 0$. Let N_1 be the amount of write service demand that arrived at D_1 during the interval $(0, T_1)$. From $t = 0$ to $t = T_1$, only one read request is serviced at D_1 . As soon as P_1 has this read request serviced, it sends b_r/b_w write requests to D_0 . Assume that these write requests to D_0 are serviced before T_1 (a pessimistic assumption). P_0 is completing reads from D_0 at a rate $1/rb_r$ during the interval $(0, T_1)$, except while D_0 is servicing the b_r/b_w write requests from P_1 , which takes time wb_r . Thus, during $(0, T_1)$, D_0 spends $T_1 - wb_r$ time to service reads from P_0 , which translates into $(T_1 - wb_r)/rb_r$ read requests serviced. These newly read blocks are written to D_1 . Therefore,

$$\begin{aligned} N_1 &= [T_1 - wb_r] \frac{1}{rb_r} \frac{b_r}{b_w} wb_w \\ &= N_0 f + wb_r(1 - f). \end{aligned}$$

Let $\alpha = wb_r(1 - f)$. We rewrite the above formula as

$$N_1 = N_0 f + \alpha. \quad (1.1)$$

Consider the relation between the write service demand at D_1 at T_0 and T_1 . From Equation 1.1, we get

$$N_0 > N_1 \Leftrightarrow N_0(1 - f) > wb_r(1 - f),$$

or

$$N_0 > N_1 \Leftrightarrow (f < 1 \text{ and } wb_r < N_0) \text{ or } (f > 1 \text{ and } wb_r > N_0). \quad (1.2)$$

Similarly,

$$N_0 < N_1 \Leftrightarrow (f < 1 \text{ and } wb_r > N_0) \text{ or } (f > 1 \text{ and } wb_r < N_0). \quad (1.3)$$

wb_r , the critical threshold for N_0 , is the amount of write demand generated for each read completed. From Eqs. 1.2 and 1.3 we can see that when $f < 1$, the write demand at D_1 remains relatively stable, oscillating about the threshold value wb_r . If N_0 exceeds the threshold, the system moves to reduce the demand at T_1 (Eq. 1.2); if it falls below the threshold the system moves to increase the

¹The pending write service demand is the write time per block multiplied by the number of pending write blocks in the queue.

write demand (Eq. 1.3). Hence the queues remain bounded in size and the system is stable. The behavior for $f > 1$ is very different. If the write demand N_0 reaches above the threshold wb_r (Eq. 1.3), the system moves to increase the write demand to $N_1 > N_0$. If N_0 falls below the threshold then the system moves to decrease the write demand further; however, this behavior cannot be sustained since the completion of a single read request at D_0 puts wb_r write demand at D_1 .

The queues at T_1 are the same as at T_0 , except the write service demand at D_1 is N_1 rather than N_0 . Let N_i , $i \geq 0$, be the write service demand at D_1 at time T_i . Since N_0 is arbitrary, from Eq. 1.1, we can write N_i as

$$N_i = wb_r + f^i(N_0 - wb_r). \quad (1.4)$$

From Eq. 1.4, we have:

$$(f < 1) \Rightarrow (f^i \rightarrow 0) \Rightarrow (N_i \rightarrow wb_r)$$

$$(f > 1) \Rightarrow (f^i \rightarrow \infty) \Rightarrow (N_i \rightarrow \infty)$$

The system is stable if $f < 1$ and unstable if $f > 1$ and $wb_r < N_0$. That is, if the write demand in any queue exceeds that in the other by wb_r at any time, then for $f > 1$, the queue grows without bound. Since read and write times are nondeterministic, and a single read generates a write demand of wb_r , this initial condition will be reached with very high likelihood.

This race condition is consequently caused by a process (say P_0) sending writes to a disk (D_1) which already has a backlog of writes. When the backlog is large enough, P_1 completes reads at D_1 at a rate which is insufficient to prevent P_0 from further increasing the backlog on D_1 , eventually forcing P_1 to come to an effective halt. Fig. 4 shows how the number of pending write requests at one of the disks in the simulated system grows unbounded as time progresses for different values of b_w and b_r . (The other disk had almost no pending writes, as predicted). The number of pending write requests for $b_r=8$ increases much faster than for $b_r=4$, since $f = w/r$ is greater in the first case. The values for the slopes agree closely with those predicted by the analysis.

While P_1 's progress is virtually stopped, D_0 is busy servicing reads while D_1 is busy servicing writes. When P_0 runs out of blocks to read at D_0 , the backlog of writes at D_1 must be serviced while D_0 is idle. When all backlogged writes have been written, P_1 will begin reading from D_1 , and D_0 will begin servicing the writes sent to it. The larger the value of f , the larger the backlogged queue of writes and the larger the percentage of idle time at a disk.

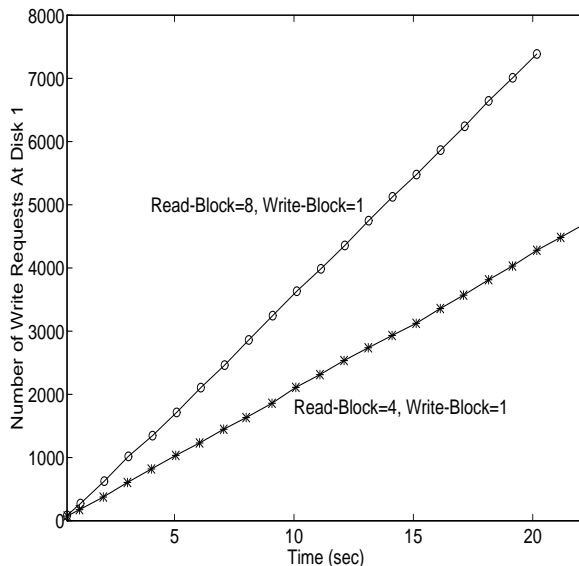


Figure 4 Number of pending write requests at d_1 as time progresses for $b_w=1$ and $b_r=4, 8$

4 ANALYSIS OF A MULTI-DISK SYSTEM

A generalization of the example in the previous section is the following permutation, which is a special case of a related permutation arising in external merging by independent, concurrent jobs [6]. There are d concurrent processes, P_0, P_1, \dots, P_{d-1} . The input for P_i is the $(d-1)N$ blocks of data $(j, i), 1 \leq i \leq N, 0 \leq j \leq d-1, j \neq i$. Initially this data is spread out among the $d-1$ disks $D_j, j \neq i$. P_i must interleave the records of these $d-1$ streams and write out the interleaved records to disk D_i . Fig. 5 illustrates this for $d=3$. Note the correspondence between this and the layout of Policy 2.

All processes perform the interleaving independently and concurrently as follows. Each process P_i reads a block of data from each of its $d-1$ input disks; the records in these $d-1$ blocks are interleaved, and $d-1$ output blocks are written out to its output disk D_i . In the model of [1, 2, 7], it is easy to find a schedule of disk reads and writes that performs the permutation in $(2Nd - N)$ parallel I/Os, which is close to the minimum possible, $(2Nd - 2N)$.

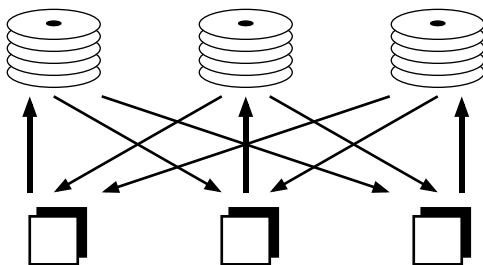


Figure 5 Record interleaving permutations, $d = 3$

Assume that at time $t = 0$ one disk D_m has a total queued write demand of N_0 and all other disks have no queued demand. Since none of the other disks $D_{i \neq m}$ have any queued demand, all processes $P_{i \neq m}$ are blocked waiting on reads from D_m , and the total read demand queued at D_m is therefore $(d-1)b_r r$.

Let t_1 be the time required by D_m to service the entire demand $N_0 + (d-1)b_r r$ queued at $t = 0$. We call this interval a *depletion cycle*. During the cycle $(0, t_1)$, each disk $D_{i \neq m}$ receives a total service demand of $(d-1)b_r w + (d-2)b_r r$, excluding access requests from P_m . The first term is due to the single write that P_i generates to its write disk in this interval, and the second term is due to the single read request that D_i receives from every other process $P_{j \neq i, m}$.

Except for this newly acquired demand on D_i , reads by P_m would be serviced unimpeded on D_i . The worst case is that the writes to all disks $D_{i \neq m}$ during $(0, t_1)$ do not overlap with one another (this assumes that $(d-1)b_r w \geq b_r r$). We will assume that this is true for the interval $(0, t_1)$. This is a reasonable assumption because it is straightforward to show that even if it is not true during this cycle, it must be true for the next cycle.

P_m will generate new reads to a disk $D_{i \neq m}$ at rate $1/r b_r$ during the portion of the cycle when its reads are not blocked by other processes' service demands. Let $\hat{f} = f(d-1)$. Thus the total amount of write demand that P_m will generate during the cycle is

$$\begin{aligned} N_1 &= [t_1 - (d-1)(d-1)b_r w] \frac{1}{b_r r} \frac{b_r}{b_w} b_w w (d-1) \\ &= \left[N_0 + (d-1)b_r r (1 - \hat{f}) \right] \hat{f} \end{aligned}$$

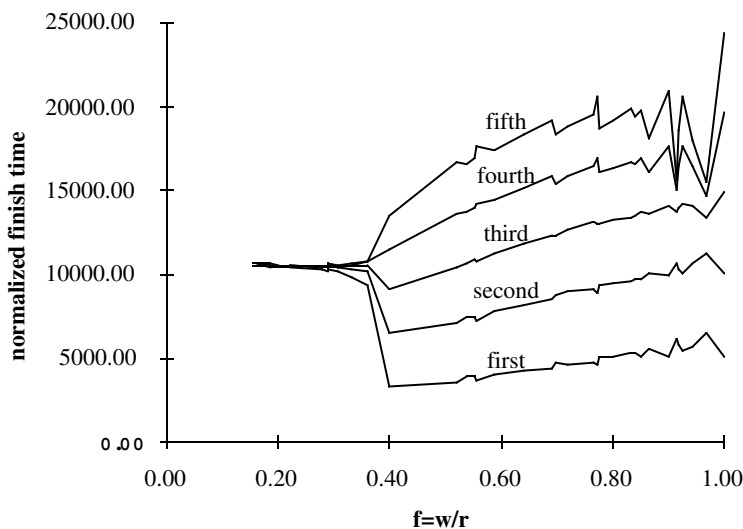


Figure 6 Job write finish times: 5 jobs, 5 disks

The system will be unstable if $N_1 > N_0 \Leftrightarrow (d-1)b_r r(1-\hat{f})\hat{f} > N_0(1-\hat{f})$. If $\hat{f} > 1$, then if the load at D_m reaches or exceeds the threshold $(d-1)b_r r \hat{f}$, the write demand at that disk will grow without bound.

The larger d and $b_r w$ are, the longer it may take the system to reach this point, but eventually the random nature of the service times will allow one process, in effect, to block the remaining processes until all of its writes have been serviced. The remaining processes are not completely stalled, but only get one read completed during each depletion cycle, and the cycles rapidly grow in length. For large f and N , the slowdown over the minimum possible time $2N(d-1)$ approaches d .

Fig. 6 shows the results from a simulation of a system with $d = 5$. The figure plots the times at which each process has had all of its writes serviced. Since the actual order of finishing is random, the individual curves refer to the first, second, etc., process to finish. The finish times are normalized by the sum of the average read and write times per block, which is proportional to the total service demand. This was done because the simulations were run for a range of read and write blocking factors (write blocking of 1, 2, and 4, and read blocking from 1 to 12). The blocking factors determine the average read and write times per block and hence affect the finish times.

Fig. 6 shows fairly good agreement with the analysis. For $f = w/r$ less than 0.30, all the processes finish at approximately the same time. As f increases the finish times of the processes begin to diverge because of I/O serialization. Considerable fluctuation in the finish times was noticed for different simulation runs, due to variability in the amount of work that has completed before a queue exceeds the threshold required for serialization to begin.

5 SOLUTIONS TO RACING

Appropriate run placement can be used to eliminate the occurrence of the race condition, as discussed in Sec. 2.2. However, it may not be always possible to use this method, as in the case of staggered arrivals of jobs. Two other methods to avoid the race condition are: the use of appropriate disk-scheduling policies and buffer management. We discuss each of these in turn.

In Round-Robin (RR) disk scheduling, I/O requests of different jobs are queued at separate queues at a disk, and the queues are serviced in a round-robin fashion. Read-Priority (RP) scheduling gives read requests priority over writes. Fig. 7 shows the performance of these two scheduling policies and compares the results to that of FCFS for the Case 2 placement. RR and RP show significant improvement compared to FCFS for more than one job. The completion times of all jobs using either RR or RP disk scheduling are comparable to that achieved using the placement of Case 3.

A second mechanism for avoiding race conditions is to allocate an equal portion of the buffer space to each job. All pending write blocks of a job must be held in its own portion of the buffer. Fig. 8 shows anomalous behavior of the system; that is, increasing buffer size degrades performance significantly. For 3 jobs, the completion time goes from 46 to 60 to 67 seconds when the buffer size is increased from 2000 to 4000 to 12000 blocks. The phenomenon is contrary to our expectation that more buffer space improves performance.

Both scheduling policies and buffer partitioning avoid the race condition by insuring that a job cannot block other jobs by getting far enough ahead. In RR scheduling, when a job gets ahead and accumulates write requests at a disk, the disk queue gives equal amount of services to requests of each job, even read requests from other jobs arriving much later than the write requests. Thus, these read requests do not have to wait until all pending write requests are serviced, preventing the leading job from getting further ahead. The RP

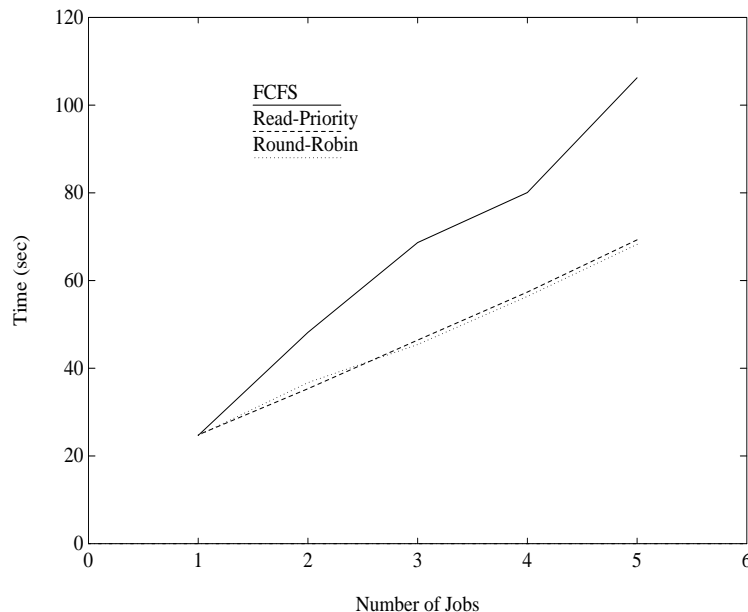


Figure 7 Completion time of all jobs for $1 \leq J \leq 5$ for the placement policy of Case 2 with 5 disks and different scheduling policies

scheduling policy works for a similar reason. When a job queues up write requests at a disk, read requests from other jobs can bypass all the waiting writes and get serviced quickly. No race condition can develop in this case, since no job's reads are delayed by a large number of queued writes. In the limited buffer case, when a job gets ahead, it can only go as far as its buffer space allows. The limited buffer space forces the job to wait and allows any lagging jobs to catch up. In Fig. 8, increasing buffer space allows the race condition to develop more fully, and performance suffers correspondingly.

The disk-scheduling policies provides a straightforward method to control the performance degradation caused by the race condition and is independent of any particular run placement policy. Simulation data suggests that the performance of both the Round-Robin and Read-Priority schemes are quite insensitive to the size of the disk buffer. The performance of the limited buffer management scheme, however, is sensitive to the amount of buffer space. Also, the buffer space needs to be controlled individually for each job. A slight increase in

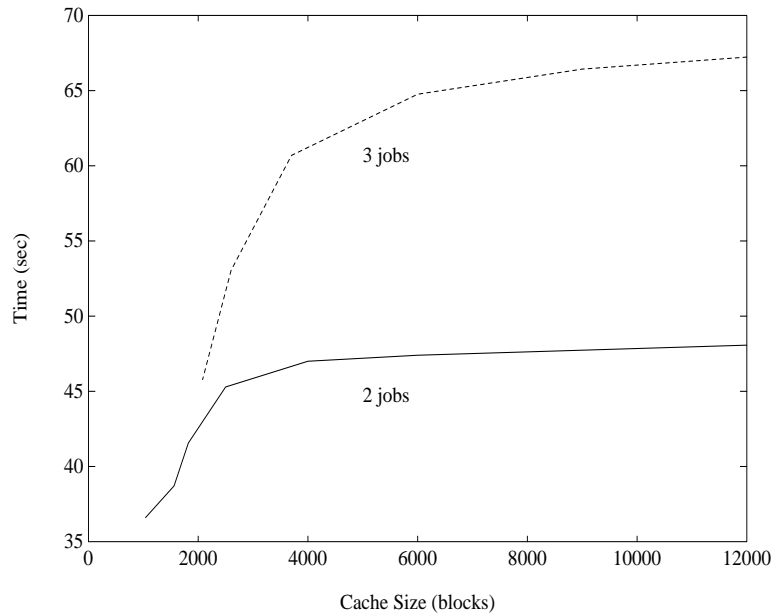


Figure 8 Completion time for 2 and 3 jobs for the placement policy of Case 2 with 5 disks and different cache sizes

buffer size may give rise to a large performance deterioration, as evidenced in Fig. 8. In practice, it may be difficult to implement this policy if the buffer is managed on a global basis.

6 SUMMARY

Several conditions are necessary for instability of the form described in this paper. First, each process must have distinct input and output disks. If a process reads from a disk to which it is also writes, no race can occur. Secondly, reads must be faster than writes; this may arise due to differences in blocking factors for reads and writes or parallelism in the input stream. Third, the disk scheduling policy plays a role in determining if the system will become unstable. Finally, a process should not be held up for any other resources. Under these conditions it is possible for one of the disks to build up a substantial queue of pending writes; all other processes reading from that disk are slowed down.

Since the process writing to that disk does not have any reads on that disk, it is not blocked and adds more write requests to the backlogged disk. This results in an even bigger write queue at that disk. The positive feedback causes the processes reading from that disk to become effectively stopped.

Controlling the instability can be addressed at several levels, each of which tries to negate some necessary condition. The placement of data on the disks or the partitioning of a task among parallel processes can be judiciously chosen to prevent disjoint read/write disks for a process. Disk scheduling policies which either give priority to read requests or at least do not allow them to wait indefinitely at the tail of large write queues also prevent this instability (see [11]). Finally, the buffer size allocated to each process can be limited, so that a process waits for want of buffers. This approach conflicts with the requirement of larger buffer space to improve the disk parallelism required by some permutations [2, 1, 7].

Acknowledgements

This work was partially supported by NSF and DARPA grant CCR 9006300.

REFERENCES

- [1] Aggarwal, A., and Vitter, J.S., "Input/Output Complexity of Sorting and Related Problems," *Comm. ACM*, September 1988, pp. 1116–1127.
- [2] Cormen, T.H., "Fast Permuting on Disk Arrays," *J. Parallel and Distributed Computing*, 1993, pp. 41–57.
- [3] Nodine, M.-H., and Vitter, J.-S., "Large-Scale Sorting in Parallel Memories," *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 29–39.
- [4] Nodine, M.-H., and Vitter, J.-S., "Optimal Deterministic Sorting in Large-Scale Parallel Memories," *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1992.
- [5] Pai, V.S., Schäffer, A.A., and Varman, P.J., "Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging", *Theoretical Computer Science*, June 1994, pp. 211-239.

- [6] Sinclair, J.B., Tang, J., Varman, P.J., and Iyer, B., "Impact of Data Placement on Parallel I/O Systems," Proc. Int. Conference on Parallel Processing, August 1993, pp. 276–279.
- [7] Vitter, J.S., and Shriver, E.A.M., "Optimal Disk I/O with Parallel Block Transfer," Proc. ACM Symposium on Theory of Computing, 1990, pp. 159–169.
- [8] Zheng, L.Q., and Larson, P.-A., "Speeding Up External Mergesort," Tech. Rept. CS-92-40, Dept. of Computer Science, University of Waterloo, August 1992.
- [9] Corbett, P.F., Feitelson, D.G., Prost, J.-P., and Baylor, S.J., "Parallel Access to Files in the Vesta File System," Supercomputing 1993, November 1993, pp. 472–481.
- [10] DeWitt, D.J., and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems," Comm. ACM, June 1992, pp. 85–98.
- [11] Tang, J., "Performance Study of Parallel I/O Systems," Masters Thesis, Dept. of Electrical and Computer Engineering, Rice University, 1993.
- [12] Kwan, S.C., and Baer, J.-L., "The I/O Performance of Multiway Mergesort and Tag Sort," IEEE Trans. Computers, April 1985, pp. 383–387.
- [13] Pai, V.S., and Varman, P.J., "Prefetching with Multiple Disks for External Mergesort: Simulation and Analysis," Proc. 8th Intl. Conference on Data Engineering, 1992, pp. 273–282.
- [14] Lee, K., and Varman, P.J., "Prefetching and I/O Parallelism in Multiple Disk Systems," Proc. Int. Conference on Parallel Processing, August 1995.
- [15] Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.