INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
Performance study of parallel I/O systems

Tang, Jay, M.S.

Rice University, 1993
PLEASE NOTE

The original document received by U.M.I. contained poor print. Pages were filmed as received.

This reproduction is the best copy available.

University Microfilms International
RICE UNIVERSITY

Performance Study of Parallel I/O Systems

by

Jay Tang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
Master of Science

APPROVED, THESIS COMMITTEE:

[Signatures]

Dr. Peter J. Varman, Chairman
Associate Professor
Electrical and Computer Engineering

Dr. James B. Sinclair, Co-Chairman
Associate Professor
Electrical and Computer Engineering

Dr. Joseph R. Cavallaro
Assistant Professor
Electrical and Computer Engineering

Dr. Robert Jump
Professor
Electrical and Computer Engineering

Houston, Texas
April, 1993
Abstract

Performance Study of Parallel I/O Systems

Jay Tang

The use of parallelism in I/O systems is becoming increasingly important as the performance gap between processors and disks continues to widen. This thesis studies the performance of multiple external merge sorts in an I/O system with multiple disks. Specifically, we investigate the impact of data placement on I/O performance. For one intuitively good placement policy, a race develops among concurrent merge sorts, resulting in the serialization of job executions and significant performance degradation. We present a model of a system with two jobs performing concurrent I/O and analyze the model. Our analysis accurately predicts the development of the race condition. We also present methods to control the race based on data placement and disk scheduling policies, which are shown to be effective through simulations.
Acknowledgments

I would like to express my deep gratitude to my research advisors: Dr. Peter Varman and Dr. Bart Sinclair. Without their encouragement, advice, and patience, this thesis would not have been written. My thanks also go to Dr. Joseph Cavallaro and Dr. J Robert Jump, who agreed to be on my thesis committee and read my thesis in a very short period of time. Members of my committee have spent many hours to help me revise my thesis, and I thank them sincerely.

I also want to thank my colleagues at Electrical and Computer Engineering Department at Rice University. I am grateful to Jay Greenwood, Michael Wu, and Evan Speight for their helpful comments and discussion. Special thanks also go to Mark Nelson, Lakshmanamurthy Sridhar, Ramakrishnan Rajamony, and Roger Huang.

Finally, my deep appreciations go to my parents who have implanted in me the value of education since my childhood and made many sacrifices to help me obtain my education.
Contents

Abstract

Acknowledgments

List of Illustrations

1 Introduction

1.1 Motivation .................................................. 1
1.2 Thesis Contribution ....................................... 2
1.3 Related Work ............................................. 4
1.4 Outline of the Thesis .................................... 5

2 Data Placement Policies

2.1 External Merge Sort ....................................... 6
2.2 System Model ........................................... 9
2.3 Performance of Placement Policies .................... 12

3 Race Condition

3.1 Two-disk and Two-job System .......................... 20
3.2 Analytical Model of the Two-job System .............. 25
   3.2.1 Causes of the Race Condition ....................... 26
   3.2.2 Race Condition under Bounded Cache Size ....... 29
3.3 Solutions to the Race Condition Problem ............. 41
3.4 The Effect of Buffer Replacement ...................... 44
3.5 Summary ........................................................................................................ 56

4 Race Condition Solutions in Merge Sort ........................................ 58

5 Conclusions ..................................................................................................... 64

Bibliography ....................................................................................................... 67
## Illustrations

2.1 The run formation phase of external merge sort ........................................ 7
2.2 The merge phase of external merge sort ....................................................... 7
2.3 System model with CPUs, disks, and a disk cache ....................................... 10
2.4 Run placement policies for a light-load system with 2 jobs and 5 disks ........... 14
2.5 Completion time of all jobs for $1 \leq J \leq 5$ for the placement policies described in Figure 2.4 with 5 disks .......................................................... 14
2.6 Individual completion time for $1 \leq J \leq 5$ for the placement policy of Case 2 described in Figure 2.4 with 5 disks under FCFS scheduling ................. 18

3.1 Case one: two-disk and two-job system ....................................................... 21
3.2 Case two: two-disk and two-job system ....................................................... 21
3.3 Number of pending write requests at $d_1$ as time progresses for $b_w=1$ and $b_r=4$, 8 ................................................................. 22
3.4 Number of pending write requests at $d_1$ as time progresses for $b_w=2$ and $b_r=6$, 8 ................................................................. 23
3.5 Comparison of job completion time between simulation and total disk service demand ....................................................... 25
3.6 Number of pending write requests stays at a level dependent on cache sizes ................................................................. 30
3.7 The development of the race condition during the steady phase ................. 31
3.8 The breakdown of the total completion time according to our analytical model ........................................ 37

3.9 The percentage error between our analytical model and simulation for 
25 \leq \text{cache size} \leq 6000 and global buffer management policy .......... 39

3.10 The percentage error between our analytical model and simulation for 
25 \leq \text{cache size} \leq 400 and global buffer management policy .......... 39

3.11 The percentage error between our analytical model and simulation for 
25 \leq \text{cache size} \leq 6000 and local buffer management policy .......... 40

3.12 The percentage error between our analytical model and simulation for 
25 \leq \text{cache size} \leq 400 and local buffer management policy .......... 40

3.13 Completion time in a two-disk and two-job system with different 
scheduling policies and cache sizes ........................................... 43

3.14 State transition diagram for the buffer management policy that 
allows replacements ................................................................. 45

3.15 Completion time of different scheduling policies under an LRU buffer 
management policy for a two-disk and two-job system ....................... 47

3.16 Completion time for the two-disk and two-job system under the 
Adaptive policy and its two variants ............................................. 50

3.17 Completion time of different scheduling policies under a 
replacement-type of buffer management policy for the two-disk and 
two-job system ........................................................................ 52

3.18 Number of pending write requests in \(d_0\) and \(d_1\) for a cache size of 200 
blocks ......................................................................................... 52

4.1 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 ......................... 59
4.2 Completion time of 2 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy ............... 60
4.3 Completion time of 3 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy ............... 60
4.4 Completion time of 4 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy ............... 61
4.5 Completion time of 5 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy ............... 61
4.6 Completion time of all jobs for $2 \leq J \leq 5$ for the placement policy of Case 2 with 5 disks under the FCFS and the Adaptive scheduling policy with DPL ........................................ 62
Chapter 1

Introduction

Since the early 1980's, the speed of computer processors has been growing exponentially. In the last few years, the use of parallel processing technology has allowed orders-of-magnitude increase in computational capability by harnessing the power of multiple processors to work on parts of a single problem. Unfortunately, the performance of I/O subsystems has not kept pace with these improvements in processor performance. The speed to read from or to write to a disk has improved only marginally in the last decade. The maximum data rates possible from single disks are limited by physical constraints such as the rate of disk rotation and time for disk head movement, which are unlikely to increase dramatically due to the mechanical systems involved.

To reduce the speed mismatch between processors and I/O subsystems, there have been a number of proposals for using multiple disks to boost the performance of I/O systems [8, 16, 18], much the same way parallel processing technology is used to speed up compute-bound problem. The objective of this thesis is to study methods for effectively using multiple disks to provide better I/O system performance for an important database application, namely external merge sort.

1.1 Motivation

In this thesis, we will study the performance of external merge sorts in a parallel I/O system with multiple disks. External sorting of large files is important in many
database applications and has been studied extensively for several decades. The use of external sorting is necessitated by the limited size of computer memory. Merge sort is a well-known algorithm used for external sorting in database systems. The algorithm has two distinct phases. In the first phase, a memory-load of data is read from a disk, sorted, and written out to an external disk to form a run. A large number of unsorted data is thus broken into multiple runs; data within each run are sorted. These runs are merged in a single run, in a small number of merge passes during the second phase. The extensive use of secondary storage devices to store temporary runs indicates that the performance of external merge sort depends greatly on the performance of I/O systems.

The system assumed in this study consists of a set of shared disks that can be independently accessed by any executing job. Although our investigation is motivated by the study of concurrent merge jobs, the results also hold for other applications having similar data access patterns.

1.2 Thesis Contribution

This thesis describes various ways to improve the performance of independent and concurrent merge jobs through

- run placement strategies, and
- disk scheduling policies.

The problem of deciding on which disks the data of different jobs should be placed arises frequently in high-performance database management systems executing multiple external merge sorts. In our study, a single input run must be placement entirely on a single disk (we refer to this as run-level striping). This is the method commonly
employed by DBMS systems like SQL/DS and DB2. As a consequence, the input runs of a job may be distributed over multiple disks, but the sorted output run must go to a single file on a single disk. The asymmetry between reads and writes in such a I/O system suggests that the placement of sorted runs may have a significant impact on the performance of a merge sort. Intelligent run placement policies can markedly improve disk utilisations and, consequently, the completion time of jobs.

Our study of the run placement problem led to the discovery of an interesting phenomenon that has a significant impact on performance. For a certain placement strategy, despite its symmetry in both the run placement and the load on disks, we observed that concurrent merge jobs race against each other. One job gets ahead, and the rest come to a halt until that job terminates. Then, the race repeats again among the remaining jobs. This phenomenon results in a significant serialization of disk accesses and hence performance degradation. To mitigate the detrimental effect of the race condition, we propose different disk scheduling policies. Specifically, we investigate these policies:

- Round Robin
- Read Priority
- Adaptive.

Most disk caches (we use the term buffer and disk cache interchangeably throughout this thesis) allow a Least Recently Used (LRU) type of replacement. Although the strategies of Round Robin and Read Priority work quite well to combat the effect of the race condition with a sufficiently large disk cache, the effectiveness of each deteriorates rapidly as the size of the disk cache shrinks to the point of requiring a considerable amount of replacements. The Adaptive policy, a combination of Read
Priority and First-Come-First-Served (FCFS), is adopted to deal with the problem of replacement.

In our study, we assume that a single read fetches a chain of $N$ blocks to amortize the cost of seek time and rotational latency. With an extremely small disk cache size, as compared to the size of merge data, the Adaptive policy fails to avoid the replacement problem. We propose a prefetch policy, Dynamic Prefetch Length, that reduces the number of blocks in a read chain when the amount of free buffer space becomes scarce. This strategy successfully eliminates buffer replacements.

The unintuitive behavior of concurrent merge jobs under the placement policy that causes race condition motivates us to further investigate this phenomenon. We construct an analytical model for a simple two-disk and two-job system which enables us to show the effect of the race condition as a function of various I/O system parameters, including the per block read and write time and the number of blocks read or written at a time.

Database systems typically do not allow application processes to control file placement on disks and disk scheduling policies, functions normally reserved for the operating system. The results of this thesis demonstrate the importance and the relevance of providing this kind of capability to application processes. Specifically, the thesis shows that external merge sorting programs can achieve significant speedup through intelligent data layout schemes and discuss methods to avoid performance degradation caused by the race condition.

1.3 Related Work

Performance evaluation of different multiple-disk systems and associated management strategies have been studied in [19, 11, 17, 4, 5, 9]. A number of papers have dealt
specifically with the I/O performance during the merge phase of merge sort [3, 10, 20, 1, 6, 15, 14, 24]. New external sorting algorithms based on variants of merge sort [12] or on bucket sorting (see [23] for a randomized algorithm and [13] for a deterministic algorithm) for parallel disk systems have been recently proposed. Some of these algorithms achieve optimal asymptotic I/O performance, at the cost of a constant factor increase in the number of passes over the data, or in the number of block I/Os in any pass. All these papers deal with a single sort or merge job, and are not concerned with the interaction of multiple concurrent jobs. For other work on the I/O performance of different database algorithms like transitive closure see [2] and [22].

1.4 Outline of the Thesis

In Chapter 2, we describe the simulation model we used to study the performance of concurrent and independent merge jobs. We also present different data placement schemes and discuss their effects on the performance of multiple concurrent merge jobs. Chapter 3 presents an analytical model of a simple two-disk and two-job system which exhibits the race condition. We then discuss various scheduling strategies to combat the disastrous effect caused by the race condition under different buffer management policies. In Chapter 4, we demonstrate the effectiveness of different disk scheduling policies in suppressing the development of the race condition in merge sort. We summarize the thesis in Chapter 5.
Chapter 2

Data Placement Policies

In this chapter, we will first explain the external merge sort algorithm which motivates the research described in this thesis. The algorithm consists of two distinct phases: run formation phase and merge. The run formation phase breaks a large unsorted file into a number of smaller sorted files, referred to as runs. During the merge phase, these sorted runs are merged together into a single sorted run. Following a discussion of external merge sort, we present our simulation model for the merge phase. Finally, we discuss the advantages and disadvantages of different data placement policies and their respective performance.

2.1 External Merge Sort

External merge sort is an important algorithm for large database systems. The algorithm has two distinct phases: run formation and merge. The run formation phase breaks a large unsorted file into many smaller sorted runs. Figure 2.1 illustrates this phase. The unsorted file initially resides on a database disk or disks. Typically, the size of the file is too large to fit into the computer memory all at once. The algorithm first reads in a memory-load of data. These data are sorted in memory to form a run. The run is then written out to one of a set of disks which will be used in the merge phase. The above steps are repeated until the entire unsorted file has been broken into many smaller sorted runs residing on a number of disks.
Figure 2.1  The run formation phase of external merge sort

Figure 2.2  The merge phase of external merge sort
Figure 2.2 shows the process of the merge phase. In the general case, this phase consists of several passes. In each pass, some number \( k \) of sorted runs are merged together to form one larger sorted run. Each pass reduces the number of sorted runs by a factor of \( k \). The choice of \( k \) depends on the amount of computer memory available for the merge. We consider only a single-pass merge in this study. The sorted runs that are merged together in one pass will be referred to as input runs, and the resulting single sorted run as an output run. A disk containing an input run is referred to as an input disk, and the disk with the output run as an output disk. The merge process reads into the buffer \( N \) consecutive blocks from each run. It then merges the data and produces some number of merged blocks. Once some number \( M \) of merged blocks have been produced, they will be written to the output disk. When all the \( N \) data blocks of any run are consumed by the merge process, it reads in the next \( N \) consecutive data blocks from that particular run. The process continues until all data from all input runs are merged and written to the output disk.

Figure 2.2 shows the input and output disks for a typical merge. In this thesis, we consider the interaction of several identical, independent, and concurrent merges. We refer to the process executing a merge as a job. Each job has its own set of input runs and its own output run. Jobs share the disk subsystem on which input and output runs are placed. Thus, an input disk of one job might be the output disk of another job. The sharing of disks among different jobs causes multiple merge jobs to interact in quite interesting and nonobvious ways.

In this thesis, we limit our study of external merge sort to a single-pass of the merge phase only. There are many important issues that arises during the merge phase, such as data placement schemes, disk scheduling policies, buffer management policies, and prefetching strategies, that have significant effects on the performance of the merge phase. There issues are interrelated in a variety of ways. Achieving a
better understanding of how the performance of the merge phase is affected by these issues and their interactions, we develop methods to reduce the merge phase execution time by increasing the amount of I/O concurrency, and thus improve overall external merge sort performance.

2.2 System Model

The system model used for evaluating the performance of multiple concurrent merge jobs consists of a number of CPUs, an I/O subsystem, and a disk cache (Figure 2.3). The I/O subsystem comprises a number of independent disks that can be accessed simultaneously. To focus on the I/O performance, the experimental results reported in this thesis assume infinite-speed CPUs. Under this assumption, there is no contention for CPUs by different jobs, even if the number of jobs exceeds the number of CPUs. Since the actual number of CPUs is irrelevant, it is convenient to visualize the system as having one CPU dedicated to each job.

A block is the smallest unit of I/O operations. It consists of some number of bytes of data placed consecutively on a disk. A run consists of several consecutive blocks of data. In our model, blocks of a run are placed physically consecutively on disks.

Our work was carried out using the YACSIM simulation package [7]. The simulation follows the random-block-depletion model proposed by Kwan and Baer [10]. During the merge phase of an external merge sort, the CPU has some number of unmerged blocks from each sorted run in memory. It chooses a leading block of any of the runs with uniform probability and consumes it. In other words, data in that block have been depleted to generate a single sorted output block. Once a block is depleted, the CPU needs the next block from the same run to be present in memory before it can continue merging.
As mentioned above, a block is the smallest unit of I/O operations (often a 4Kbyte page in commercial systems). In applications like merging where successive blocks are laid out consecutively on a disk, I/O is done in a chain of $N$ blocks so that seek time and rotational latency can be amortized. $N$ is the read blocking factor. To increase I/O concurrency, we have adopted an anticipatory *intra-run prefetching* method where reads are always initiated before the blocks in the chain are needed. Specifically, say that reads have been initiated for a chain of $N$ blocks whose block numbers are $i$, $i+1$, $\cdots$, $i+T$, $\cdots$, $i+N-1$. $T$ is defined as the *prefetch trigger*. When block $i+T$ is depleted, a read for blocks $i+N$, $i+N+1$, $\cdots$, $i+N+T$, $\cdots$, $i+2N-1$ is initiated. Similarly, when block $i+N+T$ is being depleted, a read for blocks $i+2N$, $\cdots$, $i+3N-1$ is initiated. The depletion of the $T$'th block in
a read chain triggers the prefetch of the next \( N \) blocks. In our study, we employ a prefetching strategy where \( T = 0 \).

Initially, a CPU fetches \( N \) blocks from each run of a merge job. Since sorted runs may be stored on different disks, these I/O requests may be serviced by multiple disks simultaneously. When the first block from run \( i \) is chosen for depletion, a prefetch for the next \( N \) blocks from run \( i \) is issued. Meanwhile, the CPU requests a free disk cache block, generates a write block (a block of merged data to be written to a disk), and buffers it in the cache. If, for any reason (such as replacement or an I/O operation for the block is still in progress), the leading unmerged block of a run is not in the disk cache, the merge is suspended until an I/O fetch for that block is completed. When \( M \) write blocks have been buffered, a write operation is initiated. \( M \) is the write blocking factor.

The CPU continues the cycle of prefetching blocks, depleting blocks, and writing merged data out to a disk until it exhausts data from each sorted run. When a job requests a cache block to store a newly generated write block and no free block is available, its behavior depends on the disk cache management policy used. We study two such policies. They will be discussed in more detail later. The LRU Buffer Replacement policy allows the job to reclaim a block that has been prefetched but not yet used by the merge process. Under the Buffer Wait Policy, the job simply waits until a free block is released.

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. With multiple concurrent merge jobs, there is substantial amount of contention for disk service, and for cache blocks from read and write operations of different jobs. The interaction of these jobs poses some interesting problems which will be addressed in later chapters.
The I/O subsystem consists of a set of independent disks. Each disk has 7 platters, 1258 tracks/platter, 13 sectors/track, and a 4096-byte sector, with a total capacity of almost 500 megabytes. I/O requests from different jobs and of different types (read or write) are queued in different subqueues at each disk to allow the employment of various disk scheduling strategies such as Round Robin and Read Priority. A linear seek model with a seek time of 0.04 ms/track, an average rotational latency of 8.33 ms, and block transfer time of 1.024 ms/block was used. Unless noted otherwise, a job has 20 input runs, each of which contains 1000 blocks. The read and write blocking factors are $N = 12$ and $M = 40$, respectively.

2.3 Performance of Placement Policies

The performance of multiple-job, multiple-disk, external merging depends on the performance of the I/O system, which in turn is greatly influenced by data placement on the available disks. In this section, we describe various disk layout policies and compare their relative performance. We observe that the intuitively reasonable policy that simply distributes the static load evenly among the disks does not always achieve the best performance. The dynamic interaction of the multiple jobs can lead to "runaway" behavior (referred to as a race condition) that results in significant serialization of the disk usage, in an otherwise symmetric situation. An analytical model which predicts this condition will be presented in Chapter 3.

The input data for a job consists of a number of sorted runs, and the output of a job is a single sorted run. In this study we assume (in keeping with current high-performance database systems like DB2, SQL/DS, etc.) that each run is placed entirely on a single disk without striping.
We define $J$ as the number of jobs in the system, and $D$ as the number of disks. We study four possible run placement policies for the case when the number of jobs is no more than the number of disks (i.e., $J \leq D$), as listed below. These are shown in Figure 2.4 for the case of $J = 2$ and $D = 5$. Each job has some number of input runs (indicated by Read) and a single output run (indicated by Write). Disks containing only input runs are called read disks; disks containing only output runs are called write disks; disks with both input and output runs are called shared disks. 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.

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

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

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

  In this allocation, each job uses $D - 1$ disks for input and the remaining disk for output. Job $k$, $0 \leq k \leq J - 1$, uses disk $k$ for its output run, and its input runs are spread evenly among the remaining disks. Thus, there will be $D - J$ read disks and $J$ shared disks.

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

  This allocation is obtained by beginning with the allocation of Case 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 $(k - 1) \mod J$ to disk $k$. As in Case
Figure 2.4 Run placement policies for a light-load system with 2 jobs and 5 disks

Figure 2.5 Completion time of all jobs for $1 \leq J \leq 5$ for the placement policies described in Figure 2.4 with 5 disks
2, there will be \( J \) shared disks and \( D - J \) read disks. However, unlike Case 3, each shared disk also has input runs of the same job that read from that disk.

- **Case 4: Read from All Disks**

  In this allocation, each job uses all \( D \) disks for input and one disk for its output run. There are \( J \) shared disks and \( D - J \) read disks.

Case 1 is motivated by the recognition that the write of an output run will 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. The performance of this allocation is shown in Figure 2.5, as the number of jobs is varied from 1 to 5. For 1 and 2 jobs, the total time is essentially the same and 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. This allocation is desirable only if it is known that there will be a small number of jobs.

The allocation in Case 2 attempts to preserve the best-case performance of a single job (that of Case 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 arrivals of jobs. Figure 2.5 for Case 2 shows the performance of this allocation strategy as the number of jobs increases. An important fact to be noted in the figure is that the increase in the time as the number of jobs is increased cannot be accounted for merely by the increase in the load on each disk. In fact, even though there is perfect symmetry in the placement of runs for each job, in the dynamic situation, jobs progress at different rates, with a significant amount of serialization
among all jobs. The underlying reason is the difference in the rates of read and write service (since reads are spread across multiple disks). Intuitively, one job (say job A) succeeds in placing 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. Note that job A is not slowed down since it does not have any input runs 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 this job completes, the remaining jobs race against each other, and the same behavior continues with the remaining jobs. The allocation policy of Case 2 which leads to this race condition is analyzed in Chapter 3. Methods to control this situation are presented in Section 3.3 and 3.4.

Case 3 presents one method to control the runaway conditions inherent in the Case 2 layout. Essentially, the input runs of each job are permuted so that every write disk also contains input runs from the same job. This data placement scheme prevents a job from racing ahead, since the effect of its queued writes would slow down its own reads as well. Figure 2.5 shows the significant performance improvement of Case 3 over that of Case 2. For 5 jobs, the completion time for Case 2 is 105 seconds, while the time for Case 3 is 70 seconds. (This slowdown increases as the number of blocks per run increases.) One disadvantage of Case 3 over the previous two methods, however, is that the policy cannot be applied in the case of staggered arrivals of jobs; the number of jobs in the system must be known prior to laying out data on disks.

Case 4 is a straightforward run placement policy. All input runs are distributed evenly on the set of D disks. There is at most one output run on any disk. Because each disk has some number of input runs from each job, no race condition would
develop, as noted above. Figure 2.5 shows that the performance of Case 4 is reasonably good. The advantage of Case 4 is that the policy can accommodate staggered arrivals of jobs. 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.

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 $J < \frac{D}{2}$, Case 1 provides the best performance. The placement of input and of output runs is relatively balanced (see Figure 2.4). With prior knowledge of $J$ and $J \geq \frac{D}{2}$, the policy of Case 3 should be used for run placement. Case 4 is the most appropriate when there are staggered job arrivals in the system, as it consistently provides reasonable, albeit suboptimal, performance.

Figure 2.6 shows the individual completion time of each job for the run placement of Case 2 in Figure 2.4. The points along a vertical line with the same x-axis value are the completion times of individual jobs. For example, in Figure 2.6, the finish time for 4 jobs are 31, 56, 70, and 82 seconds. Data for 10 disks, $1 \leq J \leq 10$, and 5000 blocks per run have also been collected, and a similar pattern was noted. The line at the lower part of Figure 2.6 is the first job’s completion time, which is between 25 to 31 seconds, as the number of concurrent jobs running in the system varies from one to five. Increasing system load (more jobs) does not increase the first job completion time appreciably. This is only possible if all the other jobs that finishes later do not get much disk service and hence make very little progress during the interval. This serialization of the job executions results in the underutilization of disks and poor performance.

A major advantage of Case 2 and Case 4 placement policies is their ability to accommodate staggered job arrivals. Although the scheme of Case 3 performs better, its application is limited since it requires a priori knowledge of the number of jobs.
Figure 2.6  Individual completion time for $1 \leq J \leq 5$ for the placement policy of Case 2 described in Figure 2.4 with 5 disks under FCFS scheduling.

in the system and cannot accommodate staggered job arrivals. The performance gap between Case 3 and Case 4 appears to be small in Figure 2.5 for the I/O load employed. This gap, however, will increase if we increase the size of the input of each job and hence the I/O load in the system. If the race condition under Case 2 placement can be eliminated, this placement policy has the potential to provide even better performance than Case 4, while retaining the ability to accommodate future arrivals of jobs. In Chapter 3, we will analyze the causes of the race condition and present various solutions.
Chapter 3

Race Condition

In our study of the performance of multiple concurrent merge sort jobs, we have observed an interesting and, at first, surprising occurrence of a race condition: the serialization of job executions despite symmetry in both the run placement and the load on disks. The race among different jobs results in significant performance degradation. Finding the causes of the race condition and how it manifests itself allow us to gain a better understanding of the interactions of concurrent I/O tasks, and may help us discover other potential performance problems.

In this chapter, we present a simple two-disk and two-job system which may exhibit a race condition. The race condition causes the two jobs to execute serially, instead of in parallel, with one finishing far behind the other. We first show that the race condition occurs because there is a large number of pending write requests from one job queued together at a disk, effectively blocking any read requests from the other job to receive any service from that disk. Our analysis, under the assumption of infinite cache sizes, allows us to predict whether or not a race condition will occur. We then construct a model that describes how the race condition evolves with finite cache sizes. Initially, the number of pending writes from a job queued at one disk steadily increases until it reaches an upper ceiling imposed by the finite cache size. Then, that particular job races ahead of the other job until it completes. Based on the model, we are able to accurately predict the different extents to which the race
condition develops and the completion time of both jobs. The predictions of the analytic model agree well with the data obtained from simulations.

Based on our understanding of the causes of the race condition, we propose disk scheduling policies to prevent its occurrence. We present the RoundRobin and the ReadPriority disk scheduling schemes. Both of these schemes work fairly well to eliminate the race condition, assuming no replacement; that is, when a job requests \( N \) blocks of disk cache and they are not available, it simply gets blocked until its request can be satisfied. When we allow LRU replacement of input blocks in our buffer management scheme, both RoundRobin and ReadPriority scheduling policies perform poorly with small cache sizes; block replacements destroy their effectiveness completely. To combat the race condition under buffer replacement, we introduce the Adaptive disk scheduling policy, which is a combination of ReadPriority and FCFS. The policy dynamically switches between ReadPriority and FCFS, based on the number of pending write requests in the buffer, to reduce significantly the amount of buffer replacement. It shows good performance even with small cache sizes.

### 3.1 Two-disk and Two-job System

Consider a system consisting of two disks, labeled \( d_0 \) and \( d_1 \), and two jobs, \( j_0 \) and \( j_1 \). Initially, \( d_0 \) has a file A and \( d_1 \) has a file B stored on it. Job \( j_0 \) needs to make a copy of A, and job \( j_1 \) needs to make a copy of B. In the first case, \( j_0 \) makes a copy of file A on \( d_0 \) while \( j_1 \) makes a copy of file B on \( d_1 \) (see Figure 3.1). In the second case, \( j_0 \) makes a copy of file A on \( d_1 \), and \( j_1 \) makes a copy of file B on \( d_0 \) (see Figure 3.2). We want to compare the performance of the system in these two cases.

Scheduling at each disk is FCFS, and jobs block on reads but not on writes. A job reads \( b_r \) blocks from one disk and writes \( b_w \) blocks to the other disk during each
Figure 3.1 Case one: two-disk and two-job system

Figure 3.2 Case two: two-disk and two-job system
disk access. Unless specified otherwise, $b_r = 4$ and $b_w = 1$ throughout this chapter. The average time to read a single block is $r$ and the average time to write a single block is $w$. Since seek time and rotational latency are amortized over $b_r$ and $b_w$, we have that

$$b_r > b_w \Rightarrow r < w.$$  

To simplify the analysis, we assume fixed times for reads and writes that are functions of the number of blocks read or written at a time. We consider the case of infinite cache size to observe the development of the race condition. Since a job blocks on reads, there can be at most one read request pending at any disk queue.

From Figure 3.1, the completion time of $j_0$ ($j_1$) is just the total disk service demand (reads and writes) on $d_0$ ($d_1$), since the two jobs do not interact. From Figure 3.2,
Figure 3.4 Number of pending write requests at $d_1$ as time progresses for $b_w = 2$ and $b_r = 6, 8$

it is less clear how $j_0$ and $j_1$ would behave, as they share disks for reads and writes. Suppose $d_1$ has some pending write requests from $j_0$ followed by one pending read request from $j_1$. Job $j_1$ cannot make any progress until the first input block of its chained read request is completed. Meanwhile, the read requests of $j_0$ get serviced without interruption at $d_0$ (since $j_1$ is blocked), which permits $j_0$ to generate more write requests at $d_1$. Thus, any subsequent read requests at $d_1$ will wait longer before they reach the head of the disk queue. If the trend continues, the number of pending write requests at $d_1$ can grow without bound, eventually forcing $j_1$ to come to an effective halt. To validate our hypothesis, we have conducted a series of simulation experiments. Each job reads 10000 blocks from a disk and writes them to the other disk. The cache size is 20000 blocks. As the simulation progresses, we periodically
record the number of pending write requests at \( d_1 \). Figures 3.3 and 3.4 show how the number of pending write requests grows unbounded as time progresses for different values of \( b_w \) and \( b_r \). Multiplying \( b_w \) by the queue length gives the number of pending write blocks in the buffer. In Figure 3.3, the number of pending write requests for \( b_r = 8 \) increases much faster than for \( b_r = 4 \). At time = 14.0 seconds, there are approximately 2500 and 5000 writes for \( b_r = 4 \) and \( b_r = 8 \), respectively. The per block read or write time is just the sum of the total transfer time, seek time, and rotational latency divided by \( b_r \) and \( b_w \), respectively. The race condition causes \( j_0 \) and \( j_1 \) to execute serially. With only one job running, \( r < w \) means that the number of writes at a disk queue will grow steadily without bound. The slope of a line in Figures 3.3 and 3.4 is the ratio of the write service demand at a disk (which is proportional to queue length) to simulation time. A steep slope implies a faster rate at which the number of pending writes grows. Using the values of I/O parameters defined in Chapter 2 and \( b_w = 1 \), a single write takes an average time of 9.35 ms. A disk queue length of 5000 translates into 47.77 seconds of service demand. In Figure 3.3, the slope for \( b_r = 8 \) and \( b_w = 1 \) is 3.34, and \( \frac{w}{r} = 4.53 \); in Figure 3.4, the slope for \( b_r = 8 \) and \( b_w = 2 \) is 1.34, and \( \frac{w}{r} = 2.07 \). Thus, it appears that a larger ratio of \( w \) to \( r \) implies a faster rate at which write requests are accumulated.

The unlimited increase in one job's pending write requests at a disk effectively blocks the other job's chance of getting any read service, which results in the serialization of the execution of both jobs. For the two-disk and two-job system shown in Figure 3.2, one would normally expect the completion time to be the sum of total read and write service demand on any one disk. The effect of the race condition dramatically degrades performance. Figure 3.5 shows the difference between simulation time and total disk service demand. Notice that the total disk service demand is also a lower bound on completion time. It is interesting to note that the gap between the
two curves diminishes rapidly as \( b_w \) increases from 1 to \( b_r \) (which is 8). At \( b_w = 1 \), the simulation time is about 190 seconds, while the disk service demand is 115 seconds, a 60% difference. However, at \( b_w = 8 \), the curves coincide almost completely. In the next section, we shall explain the reasons of the convergence. The placement of Figure 3.1 in which the copy of a file is made on the same disk matches the low bound in Figure 3.5. The placement of the output relative to the input location of the files affects strongly the performance of the two-disk and two-job system.

3.2 Analytical Model of the Two-job System

In this section, we will present an analytical model of the the two-disk and two-job system described in Section 3.1 for the case where two jobs' executions get serialized.
First, we assume an infinite cache size and identify the conditions under which the race condition develops. We then study the evolution of the race condition when the cache size is bounded. An analytical model is constructed to predict the job completion time in this two-disk and two-job system, as cache size (relative to job size) varies from very small to very large. The results obtained from the model agree fairly well with those obtained through simulations.

3.2.1 Causes of the Race Condition

We define the following parameters for the I/O system model under consideration*:

\[ b_r = \text{read blocking factor} \]
\[ b_w = \text{write blocking factor} \]
\[ r = \text{read time per block} \]
\[ w = \text{write time per block} \]
\[ f = \frac{w}{r}. \]

We consider the setting of case two in Figure 3.2. Assume that at time \( t = 0 \), \( d_0 \) has only one read request pending, and \( d_1 \) has a read request and \( N_0 \) write service demand. Let \( T_1 \) be the time required for \( d_1 \) to service every request in the queue at \( t = 0 \). Then

\[ T_1 = N_0 + r b_r. \]

Let \( N_1 \) be the amount of write service demand arrived at \( d_1 \) during the interval \((0, T_1)\). From \( t = 0 \) to \( t = T_1 \), only one read request is serviced at \( d_1 \), the one that was in the queue at \( t = 0 \). As soon as \( j_1 \) has this read request serviced, it sends \( \frac{b_r}{b_w} \) write

---

*The unlimited buffer analysis is due to Sinclair [21]

†If \( b_r \) is not an integer multiple of \( b_w \), the exact term to use is \( \frac{b_r}{b_w} \). We use this approximation as the average in the long run.
requests to $d_0$. Assume that these write requests to $d_0$ are serviced before $T_1$. $j_0$ is completing reads from $d_0$ at a rate $\frac{1}{rb_r}$ during the entire interval $(0, T_1)$, except while $d_0$ is servicing the $\frac{b_r}{bw}$ write requests from $j_1$, which takes time $\frac{b_r}{bw} \cdot wb_r = wb_r$. Thus, during $(0, T_1)$, $d_0$ spends $[T_1 - wb_r]$ time to service reads from $j_0$, which translates into $\frac{T_1 - wb_r}{rb_r}$ number of read requests serviced. These newly read blocks are destined for $d_1$ as new write service demand. Therefore, the total write service demand $N_1$ at $d_1$ at time $T_1$ is

$$N_1 = [T_1 - wb_r] \frac{1}{rb_r} \cdot \frac{b_r}{bw} \cdot wb_r$$

$$= [T_1 - wb_r]f$$

$$= N_0 f + wb_r(1 - f).$$

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

$$N_1 = N_0 f + \alpha. \quad (3.1)$$

For the system to be stable, $N_1 \leq N_0$. If $N_1 > N_0$, write requests are arriving at a rate faster than they can be serviced. The system becomes unstable, as the queue at $d_1$ will grow without bound (unlimited cache size) until $j_0$ runs out of blocks to read. Note that for $f = 1$, $N_1 = N_0$. Notice that the state of both disks at time $T_1$ is exactly the same at time $T_0$, except the write service demand at $d_1$ is $N_1$ rather than $N_0$. We refer to the interval between $T_0$ and $T_1$ as a period. Let $N_i$, $i \geq 0$, be the write service demand at $d_1$ at time $T_i$, after $i$th period. From Equation 3.1, we get

$$N_0 > N_1 \iff N_0(1 - f) > wb_r(1 - f).$$

or

$$N_0 > N_1 \iff (f < 1 \text{ and } wb_r < N_0) \text{ or } (f > 1 \text{ and } wb_r > N_0). \quad (3.2)$$
Similarly,

\[ N_0 < N_1 \iff N_0 (1 - f) < wb_r (1 - f), \]

or

\[ N_0 < N_1 \iff (f < 1 \text{ and } wb_r > N_0) \text{ or } (f > 1 \text{ and } wb_r < N_0). \] (3.3)

\( wb_r \), the critical threshold for \( N_0 \), is the amount of write demand generated for each read completed. Equation 3.2 shows the conditions for stability, and Equation 3.3 shows the conditions for instability.

Since \( N_0 \) is arbitrary, the conditions in these two equations must hold for any \( N_i \), \( i \geq 0 \). From Equation 3.1, we rewrite \( N_i \) as

\[ N_i = wb_r + f^i (N_0 - wb_r). \] (3.4)

The condition \( f > 1 \text{ and } wb_r > N_0 \) in Equation 3.2 and \( f < 1 \text{ and } wb_r > N_0 \) in Equation 3.3 cannot hold for any \( N_i \), since the completion of a single read request at \( d_0 \) will put \( wb_r \) write demand at \( d_1 \). When \( wb_r < N_0 \), we show that \( f < 1 \) leads to a stable system and \( f > 1 \) leads to an unstable system. From Equation 3.4, we have that the term \( N_0 - wb_r \) is finite positive quantity and

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

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

The two-disk and two-job system is stable if \( f < 1 \text{ and } wb_r < N_0 \) and is unstable if \( f > 1 \text{ and } wb_r < N_0 \).

The race condition shown in this analysis is caused by a job \((j_0)\) sending writes to a disk \((d_1)\) which already has a backlog of writes. Since the amount of write service demand arriving at that disk is larger than the amount already queued, the backlog
will never be cleared. When the backlog becomes large enough, $j_1$ completes reads at $d_1$ at a rate which is insufficient to prevent $j_0$ from further increasing the backlog on $d_1$.

### 3.2.2 Race Condition under Bounded Cache Size

In Section 3.2.1, the number of pending write requests increases without bound. In reality, cache sizes must be bounded. We analyze the effect of limited cache sizes on the development of the race condition and present an analytical model that predicts the job completion time.

Throughout this section, we assume that $j_0$ is the job that races ahead and finishes first; the per block write time, $w$, is slower than the per block read time $r$. This implies that the completion time will be write-bound. We expect that the number of pending write requests eventually reaches an almost constant value, as shown in Figure 3.6. The job that races ahead would eventually run out of cache space and cannot make any progresses until some writes are written to a disk. The period during which the number of write requests increases steadily is defined as the transient phase; the period during which the number of write requests remains virtually constant is defined as the steady phase. Let

\[
N_k = \text{the amount of write demand generated at } d_1 \text{ during each period in the steady phase}
\]

\[
c = \text{cache size in blocks}
\]

\[
s = \text{size of a job in blocks}
\]

\[
k = \text{the total number of periods during the transient phase}
\]

\[
N_{\text{transient}} = \text{the amount of write service demand completed by } j_0 \text{ during the transient phase}
\]
We will first examine the evolution of the race condition during the steady phase. Assume that at the beginning of the steady phase, there is $N_k$ write demand for $j_0$ followed by one read request from $j_1$ at $d_1$. Let us define a period, $T_k$, as the time it takes to service all $N_k$ write demand and the one read request. Since $j_1$ has a pending read, the job blocks until the request is satisfied before it sends any write requests to $d_0$. Assume that the cache is full at this moment. Thus, the one read from $j_0$ cannot proceed even though $d_0$ is idle. As the writes are serviced at $d_1$, buffer space is gradually freed up. Each time there is a free disk cache block, it immediately claimed by the read process of $j_0$ at $d_0$ that was suspended for lack of buffer space. While $j_0$ has its read requests serviced, it piles up additional write demands behind the one read request of $j_1$ at $d_1$. At the end of $T_k$, a similar set of I/O requests will have been queued up at $d_1$ to be serviced during the period of $T_{k+1}$. Since $w > r$, the completion time for $j_0$ and $j_1$ is write bound. From time between zero to $j_0$'s
Figure 3.7  The development of the race condition during the steady phase

completion, \( d_1 \), the disk to which \( j_0 \) writes, is the performance bottleneck. Once \( j_0 \) completes, \( d_0 \), the disk to which \( j_1 \) writes, becomes the performance bottleneck. In order to predict the completion time of our system, it is sufficient to calculate the total service time at \( d_1 \) and \( d_0 \) during the interval when each is the performance bottlenecks.

Steady Phase Analysis

We assume here that the effect of the transient phase is negligible on the final completion time of both jobs. In particular, we assume all \( s \) blocks of \( j_0 \) were written during the steady phase. The assumption is valid when \( s \gg c \). In our two-job and two-disk simulation model, we always reserve \( 4 \times b_r \) blocks for reads; write requests
cannot use these blocks. Therefore, the write service demand $N_k$ queued during a steady phase period is

$$ N_k = (c - 4b_r) \cdot w = c' \cdot w. \quad (3.5) $$

The term $\frac{N_k}{w}$ gives the number of write blocks written in a single period during the steady phase. If the buffer is managed on a per job basis, (i.e. allocate each job half the buffer space) the value of $N_k$ would be $\frac{c'w}{2}$. Since there is a total of $s$ blocks per job, the total number of periods, $P_s$, in the steady phase is

$$ P_s = \frac{s}{\frac{N_k}{w}} = \frac{s w}{N_k}. $$

From Figure 3.7, we see that $j_1$ completes one read request during each period. The duration of each period is just $N_k + r b_r$, and the length of the steady phase, $T_s$ is

$$ T_s = (N_k + r b_r) \cdot \frac{s w}{N_k} = s w + r b_r \frac{s w}{N_k}. $$

The number of blocks $j_1$ has read and written when $j_0$ terminates depends on the number of periods in the steady phase, since $j_1$ does one read during each period. At the end of the steady phase, the number of blocks left for $j_1$ is

$$ BlocksLeft_{j_1} = s - P_s \cdot b_r = s - \frac{s w}{N_k} b_r. $$

Since all jobs are write bound, the additional time it takes for $j_1$ to finish is

$$ TimeToFinish_{j_1} = BlocksLeft_{j_1} \cdot w $$
Thus, the total time from the beginning of the steady phase to the completion of both jobs is

\[
CT_s = T_s + TimeToFinish_j h_r
\]

\[
= sw + rb_r \frac{sw}{N_k} + sw - \frac{sw}{N_k} wb_r,
\]

which reduces to

\[
CT_s = 2sw + (r - w) b_r \cdot \frac{sw}{N_k}, \quad (3.6)
\]

### Transient Phase Analysis

The analysis up to this point has been carried out without considering the effect of the transient phase. We assume that, at the start of the transient phase, \(d_1 \) initially has \( N_0 \) write service demand. Define a period \( T_i \) here similarly as in Section 3.2.2; that is, \( N_i \) write service demand is completed at \( d_1 \) during \( T_i \). The only difference is that, in succeeding periods, the amount of write demand increases until the finite-size cache allows no further increases. Let \( k \) be the number of periods in the transient phase and \( N_k \) be the amount of write demand at \( d_1 \) when we transition from the transient phase to the steady phase. From Equation 3.1, replacing \( \alpha \) by \( wb_r (1 - f) \), We have

\[
N_1 = N_0 f + \alpha
\]

\[
N_2 = N_1 f + \alpha
\]
\[ = (N_0f + \alpha)f + \alpha \]
\[ = N_0f^2 + f\alpha + \alpha \]
\[ N_3 = N_2f + \alpha \]
\[ = (N_0f^2 + f\alpha + \alpha)f + \alpha \]
\[ = N_0f^3 + f^2\alpha + f\alpha + \alpha \]

Therefore, it follows that

\[ N_k = N_0f^k + \alpha \sum_{j=0}^{k-1} f^j = N_0f^k + \alpha \frac{1 - f^k}{1 - f}. \quad (3.7) \]

Combining Equations 3.5 and 3.7 and replacing \( \alpha \) with \( wb_r(1 - f) \), we get

\[ N_0f^k + wb_r(1 - f) \frac{1 - f^k}{1 - f} = \epsilon'w \]
\[ N_0f^k + wb_r(1 - f^k) = \epsilon'w \]
\[ f^k(N_0 - wb_r) = \epsilon'w - wb_r \]
\[ f^k = \frac{\epsilon'w - wb_r}{N_0 - wb_r} \]

The value \( k \), the number of periods in the transient phase is

\[ k = \left\lfloor \frac{\log (\epsilon'w - wb_r) - \log (N_0 - wb_r)}{\log f} \right\rfloor. \quad (3.8) \]

Once we have calculated \( k \), it is straightforward to find \( N_{transient} \), the amount of write service that \( j_0 \) (the job that races ahead) has acquired during the transient phase. \( N_{transient} \) is just the sum of all \( N_i \), where \( 0 \leq i < k \). Applying Equation 3.7 and substituting \( \alpha \) with \( wb_r(1 - f) \), we have
\[ N_{\text{transient}} = \sum_{i=0}^{k-1} N_i \]
\[ = \sum_{i=0}^{k-1} \left( N_0 f^i + \alpha \frac{1 - f^i}{1 - f} \right) \]
\[ = \sum_{i=0}^{k-1} \left( N_0 f^i + \frac{\alpha}{1 - f} f^i \right) \]
\[ = \sum_{i=0}^{k-1} \left( N_0 f^i + w_b r + w_b f^i \right) \]
\[ = k \cdot w_b r + (N_0 - w_b r) \sum_{i=0}^{k-1} f^i. \]

The total amount of write demand done by \( j_0 \) during the transient phase is thus:

\[ N_{\text{transient}} = k \cdot w_b r + (N_0 - w_b r) \frac{1 - f^k}{1 - f}. \quad (3.9) \]

Since \( d_1 \) is the performance bottleneck (it always has a backlog of write requests), the length of the transient phase is just the amount of disk service provided by \( d_1 \), which is the sum of the write service demand from \( j_0 \) and the read service demand from \( j_1 \):

\[ Time_{\text{transient phase}} = N_{\text{transient}} + k \cdot b_r r. \quad (3.10) \]

Complete Model

Using Equations 3.8 and 3.9 for the transient phase, combined with our steady phase analysis, we now present a complete model for the race condition. Let \( s' \) be the number of blocks left for \( j_0 \) when the steady phase begins.

\[ s' = s - \frac{N_{\text{transient}}}{w}. \]

The number of periods during the steady phase would be:
\[ NewNumberPeriod_{\text{steady phase}} = \frac{s'w}{N_k} = \frac{sw - N_{\text{transient}}}{N_k}. \]

During each period of the steady phase, \(d_1\) would service \(N_k\) write service demand from \(j_0\) and \(rb_r\) read service demand from \(j_1\). The length of the steady phase is the product of the number of steady state periods and the duration of each period.

\[ Time_{\text{steady phase}} = \frac{s \cdot w - N_{\text{transient}}}{N_k} \cdot (N_k + rb_r) = (s \cdot w - N_{\text{transient}}) + \left(\frac{s \cdot w - N_{\text{transient}}}{N_k} \cdot rb_r\right) \quad (3.11) \]

By the end of the steady phase, the number of blocks read and written by \(j_1\) (the job that lags behind) is

\[ BlocksRead_{j_1} = BlocksRead_{\text{transient phase}} + BlocksRead_{\text{steady phase}} \]

\[ = NumberPeriod_{\text{transient phase}} \cdot b_r + NewNumberPeriod_{\text{steady phase}} \cdot b_r \]

\[ = k \cdot b_r + \frac{sw - N_{\text{transient}}}{N_k} \cdot b_r \]

\[ = b_r \left(k + \frac{sw - N_{\text{transient}}}{N_k}\right). \]

Therefore, \(j_1\) has \(s - BlocksRead_{j_1}\) blocks left when \(j_0\) terminates. The amount of time for \(j_1\) to finish is just the number of blocks multiplied by the per block write time, \(w\).

\[ Time_{j_1\text{finish}} = sw - wb_r \cdot \left(k + \frac{sw - N_{\text{transient}}}{N_k}\right) \quad (3.12) \]

The total completion time is the sum of the length of the transient and the steady phase, plus the additional time it takes for \(j_1\) to finish its remaining blocks.
Figure 3.8 The breakdown of the total completion time according to our analytical model

Figure 3.8 shows the breakdown of the total completion time. It is divided into the transient phase, the steady phase, and the time takes for \( j_1 \) to finish. Combining Equation 3.10, 3.11, and 3.12, we obtain

\[
CompletionTime = [N_{transient} + k \cdot b_r] \\
+ [(s \cdot w - N_{transient}) + vb_r \cdot \left( s \cdot w - N_{transient} \right) \frac{N_k}{N_k}] \\
+ [sw - wb_r \cdot (k + \frac{sw - N_{transient}}{N_k})]
\]

Simplifying the above formula, we have reached the final equation that would predict the completion time of a two-disk and two-job system under the race condition.

\[
CompletionTime = 2 \cdot sw + (r - w) \cdot (k + \frac{sw - N_{transient}}{N_k}) \cdot b_r \quad (3.13)
\]

Equation 3.13 can be interpreted as follows. If these two jobs were completely serialized, the total time would have been \( 2 \cdot sw \). If, from the beginning to the completion
of $j_0$, $j_1$ has received $rb_r \cdot (k + \frac{sw-N_{\text{transient}}}{N_k})$ amount of read service, add this number to $2 \cdot sw$. However, a corresponding amount of time for write demand should be subtracted since $j_1$ has less work to do during the final phase of the merge. Notice that, if we ignore the transient phase, by setting both $k$ and $N_{\text{transient}}$ to 0, Equation 3.13 reduces to Equation 3.6.

To verify our theoretical model, we compare the results obtained with Equation 3.13 with those from simulations. We ran simulations with two buffer management policies. The global buffer policy allows each job to request free blocks from a global buffer pool. The local buffer policy gives each job half the buffer space and allows it to request free blocks only from a local buffer pool assigned to itself. The size of each job is 3000 blocks. For each cache size, we ran eight trials. The simulation data have a 98% confidence interval width equal to 1.4% of the mean. The percentage error is

$$\frac{|T_{\text{simulation}} - T_{\text{model}}|}{T_{\text{simulation}}} \cdot 100.$$

We used a global buffer management policy for data collected for Figures 3.9 and 3.10; that is, both jobs request free blocks from a common global buffer pool. For Figures 3.11 and 3.12, we used a local buffer management policy; the buffer is equally divided into two halves between $j_0$ and $j_1$.

Figure 3.9 shows the percentage error between our analytical model and simulation for a cache size varying from 25 blocks to 6000 blocks. Since the total data size for both $j_0$ and $j_1$ is 6000 blocks, there is no need to investigate any cache size larger than 6000. The graph indicates that our analytical model predicts the job completion time quite well. Except for extremely small cache sizes, the percentage errors remain below 1%.
Figure 3.9  The percentage error between our analytical model and simulation for $25 \leq cache\ size \leq 6000$ and global buffer management policy

Figure 3.10  The percentage error between our analytical model and simulation for $25 \leq cache\ size \leq 400$ and global buffer management policy
Figure 3.11  The percentage error between our analytical model and simulation for \(25 \leq \text{cache size} \leq 6000\) and local buffer management policy

Figure 3.12  The percentage error between our analytical model and simulation for \(25 \leq \text{cache size} \leq 400\) and local buffer management policy
Figure 3.10 expands a section of Figure 3.9 for the cache size between 25 to 400 blocks. It is interesting to observe that the error is considerably larger (relative to the rest of the graph) when the cache size is below 40 blocks. In fact, at \( c = 26 \), the percentage error reaches about 4.8\%, an order of magnitude increase. Our analytical model does not adequately address the problems arising from extremely small cache size. One key assumption of our model is that the race condition will eventually reach a steady phase, imposed by the finite cache size. However, a very small cache size implies a very short queue length during the steady phase. Any statistical variation might disturb the equilibrium that is assumed in the steady phase as predicted by our model.

### 3.3 Solutions to the Race Condition Problem

The detrimental effect of the race condition on performance prompts us to find solutions that eliminate or suppress its development. When one job gets ahead, if its progress sufficiently slows down the other job that is contending for resources with it, the leading job would be able to move ahead further. To avoid this situation, the leading job must be prevented from monopolizing the resources.

Appropriate data placement policies can be used to eliminate the race condition. For the two-disk and two-job system, we can eliminate the race condition by simply switching the write disks of \( j_0 \) and \( j_1 \). In this case, there will be no interactions between the two jobs, and the race condition cannot develop. We have discussed various data placement policies in more detail in Section 2.3. However, it may not be always possible to use this method because of the need to know the number of jobs in the system before placing data on disks. In this section, we discuss two disk scheduling
policies that can prevent the development of the race condition. We present each of them in turn.

The first policy is RoundRobin. In the RoundRobin policy, I/O requests of each job are queued at separate queues at each disk. Requests at a disk are serviced in a RoundRobin fashion. The second policy, ReadPriority scheduling, gives read requests priority over writes. To validate our belief that these strategies would improve the performance, we ran simulations for the two-disk and two-job system. Recall that, when a job requests \( i \) blocks of buffer space and they are not available, the job gets suspended until its request can be satisfied. The various I/O parameters are as follows:

\[
\begin{align*}
b_r &= 4 \\
b_w &= 1 \\
r &= 3.11 \text{ milliseconds} \\
w &= 9.35 \text{ milliseconds} \\
s &= 3000 \text{ blocks} \\
c &= 24 \text{ to } 3000 \text{ blocks.}
\end{align*}
\]

Figure 3.13 shows the performance of these two scheduling policies and compares the results to that of FCFS for varying cache sizes. In Figure 3.13, the FCFS scheduling policy performs significantly worse than that of RoundRobin and ReadPriority. At \( c = 400 \), the completion time for FCFS is about 56 seconds, and both ReadPriority and RoundRobin finish in 40 seconds, a 40% improvement. Notice that the performance of ReadPriority and RoundRobin are both insensitive to cache sizes.

The RoundRobin scheduling policy alleviates the negative effect caused by the race condition in the following way. When one job gets ahead and accumulates write requests at a disk, the disk queue would give an equal share of service to requests
of each job, even to read requests arriving much later than existing write requests. Thus, these reads do not have to wait until all pending writes are serviced, preventing the leading job from getting further ahead. The ReadPriority scheduling policy works for a similar reason. When one job queues up write requests at a disk, read requests from the other job 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 of the other job.

Figure 3.13 also shows something quite unexpected; that is, increasing buffer size degrades performance significantly under the FCFS scheduling policy. When the buffer size is increased from 24 to 400 blocks, the completion time goes from about 40 to 56 seconds, a 40% difference. A small increase of 30 cache blocks at the low end of the cache size causes the completion time to increase dramatically, from 40 to 53 seconds. It is in this part of the curve where we have observed most of the growth of
the race condition. The phenomenon is contrary to our usual expectation that more buffer space (more resources) tends to improve performance. With small buffer sizes, when a job gets ahead, it can only go as far as the amount of buffer space allows. The limited buffer space forces the job to wait and permits the lagging job to catch up, resulting in increased disk parallelism and better performance. In Figure 3.13, increasing buffer space allows the race condition to grow more fully, and performance suffers correspondingly.

The disk scheduling policies described here provide a straightforward method to control the performance degradation caused by the race condition and is independent of any particular run placement policy. Simulation data suggest that the performance of both the RoundRobin and the ReadPriority schemes are insensitive buffer sizes.

3.4 The Effect of Buffer Replacement

In Section 3.3, we have discussed various mechanisms to control the race condition under a particular buffer management policy, namely Buffer Wait, that does not allow replacements. We now introduce a new buffer management policy, LRU Buffer Replacement. When a job needs blocks for write requests, it may replace buffer space occupied by read blocks in an LRU fashion.

Figure 3.14 illustrates the state transition diagram for the new buffer management policy. When a request for \( i \) free blocks comes, the buffer manager first checks to see whether \( i \) free blocks are available. If there are at least \( i \) free blocks, the request is satisfied immediately and the control is transferred back to the process. At any time, there are some number of blocks which have been prefetched into the buffer but not yet consumed by a job. These blocks are read blocks and candidates for replacements. If \( i \) free blocks are not available, the manager checks to see whether there are enough
Figure 3.14 State transition diagram for the buffer management policy that allows replacements

read blocks that can be replaced to satisfy the request. If the answer is positive, these blocks are replaced in an LRU fashion; otherwise, the requesting process blocks until i free blocks are available. The LRU replacement policy has a significant effect on the performance of jobs. Since read blocks come into the buffer one by one, leading blocks, which are needed first for a job to continue, are the first to be replaced. These blocks must be fetched again individually. In contrast, the previous policy of Buffer Wait does not incur this kind of performance penalty since the buffer manager suspends a process whose request for free blocks cannot be satisfied.

We now investigate the impact of the LRU Buffer Replacement policy on the development of the race condition and the effectiveness of various schemes to combat it.
We ran simulations under this new policy for FCFS, RoundRobin, and ReadPriority scheduling.

In Figure 3.13, which uses the Buffer Wait policy, the performance of RoundRobin and ReadPriority are both insensitive to cache sizes. However, Figure 3.15 shows that their performance is quite dependent on cache sizes under the LRU Buffer Replacement policy. Small cache sizes lead to significant performance degradation. For example, when the cache size is 400 blocks, the completion times for RoundRobin and ReadPriority are 60 and 68 seconds, respectively, much worse than the 40 seconds completion time observed in Figure 3.13. Another interesting observation is the completion time for the FCFS scheduling policy. We have shown, in Section 3.3, that small cache sizes suppress the development of the race condition. In Figure 3.15, the FCFS scheme performs consistently poorly. A careful examination of the simulation data suggests that there are many instances of block replacements occurring with small cache sizes. For the cache size of 400 blocks or less, there are, on the average, more than 3500 block replacements for a total data size of 6000 blocks. The number means that almost 60% of the data are read twice. When a read block has to be fetched the second time because of replacement, it is read as a single block; that particular I/O operation must therefore incur the full cost of seek time and rotational latency. As the cache size increases, we see a steady performance improvement for both the RoundRobin and the ReadPriority disk scheduling policies. When the cache size is 6000 blocks, at which point it can accommodate the entire data set of both jobs, the completion time is about 40 seconds, as would be expected. This matches the time observed in Figure 3.13. The finish time for FCFS stays at the same level even with large cache sizes, because the race condition starts to play an increasingly important role as the influence of replacement wanes.
Figure 3.15  Completion time of different scheduling policies under an LRU buffer management policy for a two-disk and two-job system

To deal with this problem, we need a mechanism that can deal with both the race condition and the large amount of replacement caused by the rapid growth of pending write requests.

We propose a new disk scheduling policy: Adaptive. Figure 3.15 shows that ReadPriority performs worse than RoundRobin. The performance gap of these two scheduling polices inspires the new Adaptive policy. As both jobs are running out of cache space, ReadPriority still gives read requests preference, despite the presence of a large number of write requests. RoundRobin, on the other hand, at least gives an equal share of service to both read and write requests. Since our two-disk and two-job system is write bound, the number of pending write requests would grow and take up more and more buffer space until stopped by the finite-size cache. We want a
mechanism that allows us to control the number of writes in the buffer. The proposed Adaptive policy is a combination of FCFS and ReadPriority.

Let job $i$ read from disk $i$ and write to disk $j$. We define disk $j$ as the feeding disk of disk $i$. Also, we define:

$$l_0 = \text{the number of pending write request at } d_0$$
$$l_1 = \text{the number of pending write request at } d_1$$
$$W_{\text{high}} = \text{the maximum number of writes at a disk queue before the scheduling policy is changed}$$
$$W_{\text{low}} = \text{the minimum number of writes at a disk queue before the scheduling policy is changed.}$$

In our two-disk and two-job system, $d_0$ is the feeding disk for $d_1$ and vice versa. Initially, when the disk queues are empty, the system begins with the ReadPriority scheduling policy at both $d_0$ and $d_1$. The purpose of ReadPriority is to prevent the development of the race condition. Write requests are being accumulated at both disk queues. When $d_0$ finds that $l_0$ exceeds an upper threshold, $W_{\text{high}}$, it changes its feeding disk, $d_1$, to the FCFS policy. By that time, $d_1$ would have some number of pending writes that have been queued at the disk longer than any read requests. The policy shift effectively gives write requests priority over read ones. It also slows down the reads at $d_1$ and the rate at which writes are being sent to $d_0$. Suppose both $d_0$ and $d_1$ have some number of writes and are under FCFS. When $l_0$ drops below a lower threshold, $W_{\text{low}}$, $d_0$ switches its feeding disk, $d_1$, to ReadPriority. $j_1$'s reads thus get serviced quickly at $d_1$, and $l_0$ starts to grow. When $l_0$ exceeds an upper threshold, $d_0$ will switch $d_1$ back to FCFS. At this point, both $l_0$ and $l_1$ are decreasing, since FCFS is equivalent to giving writes queued during the ReadPriority scheduling service priority. When the value of $l_1$ exceeds the upper threshold or drops below the
lower threshold, $d_1$ will switch the scheduling policy of its feeding disk, $d_u$, the same way $d_0$ switches it.

We have also experimented with two methods which are variants of the Adaptive policy that at first seemed promising. Method A uses the knowledge of total number of pending writes ($l_0 + l_1$) to decide at which point scheduling policies at disk queues are switched. In the second method (B), each disk changes *its own* scheduling policy, instead of its feeding disk's. For example, when $l_u$ exceeds $W_{high}$, $d_u$ changes its scheduling from ReadPriority to FCFS, hoping to flush out these writes accumulated under ReadPriority. Similarly, when $l_0$ drops below $W_{low}$, $d_0$ changes its scheduling from FCFS to ReadPriority.

Figure 3.16 shows the relative performance of the Adaptive policy and method A and B for the cache size from 40 to 6000 blocks. Adaptive performs well across the entire range of cache size. It was initially surprising to find that methods B had poor performance even at moderate cache sizes.

Method A, which uses the global number of pending writes, allows less control at individual disk queues. Suppose, due to statistical variations, $d_0$ has a few writes and $d_1$ has a large number of writes, but the sum has exceeded the global upper threshold. Both disks would be switched to FCFS in order to decrease the number of writes. Because $d_0$ has a shorter write queue, it will exhaust its write supply before its scheduling is switched to ReadPriority, since there may be enough pending writes queued up at $d_1$. Since $j_0$ has no service contention for its reads at $d_0$, it will, even under FCFS, pile up write requests at $d_1$ at a rate faster than they can be serviced. Thus, the number of pending writes at $d_1$ will never drops below the global lower threshold, and $d_0$ and $d_1$ will remain under FCFS until $j_0$ exhausts its input. The situation just described shows how method A could break down. FCFS scheduling in this two-disk and two-job system causes a large number of replacements. When
Figure 3.16 Completion time for the two-disk and two-job system under the Adaptive policy and its two variants

we increase the cache size, statistical disturbance is less likely to cause the above scenario to happen. A larger cache means there could be more pending writes queued at disks; and it is less likely that one disk has very few and the other has many writes queued up. Consequently, the performance of method A is comparable to that of the Adaptive policy under reasonably large cache sizes.

The policy of method B, which allows each disk to switch its own scheduling policy, has an element of instability. Assume both $d_0$ and $d_1$ are under FCFS. When $l_0$ drops below the lower threshold, $d_0$ changes its own scheduling to ReadPriority, which will pile up writes at $d_1$. Because $d_0$ can service reads faster than $d_1$ can process them (since $r < w$), the number of writes at $d_1$ will eventually exceed the upper threshold, keeping $d_1$ under FCFS and slowing down the reads of $j_1$. Nevertheless, the reads of $j_1$ still get a small amount of service under FCFS and will slowly increase the number
of writes at \(d_0\). When \(d_0\) has a sufficient number of pending writes, it switches from ReadPriority to FCFS and stops sending more writes to \(d_1\). During the interval, the number of writes at \(d_1\) has exceeded the upper threshold. This method works if the difference in the size of queues at each disk is close (which is the true at large cache sizes). With larger caches, the above scenario no longer poses any problems. A large cache is more likely to be able to absorb the growing number of pending writes, even as they temporarily exceed the upper threshold, as \(d_0\) switches its scheduling from FCFS to ReadPriority and back to FCFS.

Figure 3.17 compares the job completion time of the Adaptive scheduling policy with that of RoundRobin and ReadPriority over a wide range of cache size. The performance of Adaptive is insensitive to cache sizes. In order to better understand the interactions between \(d_0\) and \(d_1\), we periodically recorded the number of pending write requests at \(d_0\) and \(d_1\) with a cache size of 200 blocks. Figure 3.18 plots the number of pending write requests at \(d_0\) and \(d_1\) at fixed intervals between time 5 and 10 seconds. The two horizontal and two vertical lines give the theoretical values of \(l_0\) and \(l_1\) when the scheduling policy is changed from ReadPriority to FCFS or vice versa. As long as the number of writes at both disk queues remains in the central rectangular region, \(W_{low} \leq l_0, l_1 \leq W_{high}\), the system will behave as described. The graph demonstrates that the Adaptive policy indeed controls the growth of the number of pending write requests in the disk cache to avoid replacement. At the same time, no job is allowed to lag continuously, since its reads are periodically given priority.

When the Adaptive scheme is employed to deal with the replacement problem, there are two parameters associated with it: the upper and the lower threshold that decide when disk scheduling policies are switched. Up to \(2b_r\) blocks may be used for reads at any time by each job. The amount of cache space that can be safely used
Figure 3.17  Completion time of different scheduling policies under a replacement-type of buffer management policy for the two-disk and two-job system

Figure 3.18  Number of pending write requests in $d_0$ and $d_1$ for a cache size of 200 blocks
for writes is \( c - 4 \cdot b_r \). Let us define the maximum amount of cache space that each job should use, \( c' \), as

\[
c' = \frac{c - 4 \cdot b_r}{2}.
\]

We express the upper and the lower thresholds as a fraction of the buffer space available for writes of each job. We have \( \lambda_{\text{high}} = \frac{W_{\text{req}}}{c'} \) and \( \lambda_{\text{low}} = \frac{W_{\text{req}}}{c'} \), respectively. We would like to study the sensitivity of the Adaptive policy to these two parameters. Thus, we ran simulations for different cache sizes from 40 to 6000 blocks. For each size, we varied \( \lambda_{\text{high}} \) and \( \lambda_{\text{low}} \). We also tried large values of \( \lambda_{\text{high}} \) and small values of \( \lambda_{\text{low}} \).

<table>
<thead>
<tr>
<th>Cache Size (blocks)</th>
<th>Completion Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \lambda_{\text{high}} = 0.96 )</td>
</tr>
<tr>
<td></td>
<td>( \lambda_{\text{low}} = 0.76 )</td>
</tr>
<tr>
<td>40</td>
<td>38.83</td>
</tr>
<tr>
<td>80</td>
<td>38.23</td>
</tr>
<tr>
<td>100</td>
<td>38.57</td>
</tr>
<tr>
<td>200</td>
<td>38.60</td>
</tr>
<tr>
<td>400</td>
<td>37.90</td>
</tr>
<tr>
<td>1000</td>
<td>37.86</td>
</tr>
<tr>
<td>2000</td>
<td>38.34</td>
</tr>
<tr>
<td>3000</td>
<td>38.54</td>
</tr>
<tr>
<td>4000</td>
<td>38.63</td>
</tr>
<tr>
<td>5000</td>
<td>38.81</td>
</tr>
<tr>
<td>6000</td>
<td>39.32</td>
</tr>
</tbody>
</table>

*Table 3.1* Completion time with the Adaptive policy and different \( \lambda_{\text{high}} \) and \( \lambda_{\text{low}} \).

The data shown in Table 3.1 suggest that the the Adaptive policy performs quite well for a wide range of \( \lambda_{\text{high}} \) and \( \lambda_{\text{low}} \). In fact, there is no replacement at all for the
set of parameters tested in this table. Keeping $\lambda_{low}$ constant, increasing $\lambda_{high}$ (assume they are not close to 1) merely accumulates more pending write requests in the buffer before they get flushed out. Increasing the gap between $\lambda_{high}$ and $\lambda_{low}$ simply allows more writes requests to be written to disks before the scheduling policy reverts back to ReadPriority.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Number of Blocks Replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\lambda_{high} = 0.84$</td>
</tr>
<tr>
<td>30</td>
<td>382</td>
</tr>
<tr>
<td>32</td>
<td>53</td>
</tr>
<tr>
<td>34</td>
<td>32</td>
</tr>
<tr>
<td>36</td>
<td>17</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
</tr>
</tbody>
</table>

| Table 3.2 Completion time under the Adaptive policy with small caches and large $\lambda_{high}$ |

However, the data in Table 3.2 convey another message. At very small cache sizes, large values of $\lambda_{high}$ (close to 1.0) causes a moderate amount of replacement. For $\lambda_{high} = 0.94$ and cache size = 30 blocks, there are 974 read blocks replaced. The time it takes just to refetch these blocks would add more than 9 seconds to the completion time. Figure 3.18 helps explain this phenomenon. The value of $\lambda_{high}$ in the graph is 0.75, and, periodically, the number of pending write requests at a disk would reach the limit set by $\lambda_{high}$, which is $0.75 \cdot c'$. Larger values of $\lambda_{high}$ allows the writes to occupy the amount of buffer space whose size is very close to $c'$. When we switch from ReadPriority to FCFS scheduling, it takes some time before the number of writes starts to drop. During that interval, the inertial of ReadPriority may cause the write requests to occupy more buffer space than $c - 4b_r$, the maximum amount
of buffer writes may use without causing replacements. Table 3.2 shows that there is a significant amounts of replacement under these circumstances.

Table 3.3 shows the number of block replacements under very small cache sizes and small values of $\lambda_{low}$. These values are carefully selected so that the number of pending write requests at the low threshold at each disk queue,

$$l_{floor} = \lambda_{low} \cdot c'$$

is set to 1, 2, and 3. The switch takes place when the number of writes becomes 0, 1, and 2. The data in this table indicate that there are substantial amounts of replacements for small values of $l_{floor}$. Suppose $l_{floor} = 1$. Assume $l_0 = 1$ and $l_1$ be a finite number, and both disks are under FCFS. $d_0$ will change the scheduling policy at $d_1$ to ReadPriority when $l_0$ drops to 0. Since there is no write requests competing for service at $d_0$, $j_0$’s reads will get serviced quickly and be changed into pending writes at $d_1$. At this point, both $d_0$ and $d_1$ are piling up write requests on each other. $l_1$ might exceed the upper threshold due to the lack of service contention for $j_0$’s reads at $d_0$. Eventually, the ReadPriority policy at $d_1$ will generate a sufficiently large number of writes at $d_0$. Consequently, reads of $j_0$ cannot get serviced quickly, and $l_0$ exceeds the upper threshold. $d_0$ then switches scheduling at $d_1$ to FCFS. With small cache sizes, there might be too many writes queued up at $d_1$ during the interval. Hence, replacement becomes inevitable. In the above scenario, larger caches are better able to accommodate writes during their build-ups at $d_1$. Since the Adaptive policy is inherently stable, these writes will be flushed out eventually. The data on the last two rows in Table 3.3 show that large caches do not have the replacement problem even as $\lambda_{low}$ becomes very small.
<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Number of Blocks Replaced</th>
<th>( \lambda_{\text{high}} = 0.75 )</th>
<th>( \lambda_{\text{low}} = 0.75 )</th>
<th>( \lambda_{\text{low}} = 0.75 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>( \lambda_{\text{low}} = 0.15 )</td>
<td>3089</td>
<td>2299</td>
<td>1967</td>
</tr>
<tr>
<td>32</td>
<td>( \lambda_{\text{low}} = 0.14 )</td>
<td>2905</td>
<td>1336</td>
<td>305</td>
</tr>
<tr>
<td>34</td>
<td>( \lambda_{\text{low}} = 0.14 )</td>
<td>1719</td>
<td>1146</td>
<td>142</td>
</tr>
<tr>
<td>36</td>
<td>( \lambda_{\text{low}} = 0.10 )</td>
<td>1292</td>
<td>490</td>
<td>38</td>
</tr>
<tr>
<td>38</td>
<td>( \lambda_{\text{low}} = 0.10 )</td>
<td>1250</td>
<td>265</td>
<td>19</td>
</tr>
<tr>
<td>40</td>
<td>( \lambda_{\text{low}} = 0.10 )</td>
<td>213</td>
<td>144</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>( \lambda_{\text{low}} = 0.10 )</td>
<td>10</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.3  Completion time under the Adaptive policy with small caches and small \( \lambda_{\text{low}} \)

### 3.5 Summary

In this chapter, we studied a two-disk and two-job system. Each job had to make a copy of a file stored on a disk. The two jobs raced against each other when they read a file from one disk and copy it to the other disk. Under the assumption of an infinite cache size, we analyzed the conditions under which the race condition develops. Our analysis indicated that, when \( f > 1 \) (or \( r < w \)), the race condition caused the two jobs to execute serially and lead to significant performance degradation. Next, we assumed a bounded cache size and constructed an analytical model that predicted the completion time of both jobs. These predictions agreed well with the data obtained from simulations.
We studied methods to prevent the performance degradation caused by the race condition under two buffer management policies: LRU Buffer Replacement and Buffer Wait. We described the RoundRobin and the ReadPriority disk scheduling policies that prevent the occurrence of the race condition under the Buffer Wait policy. However, under the LRU Buffer Replacement policy, both RoundRobin and ReadPriority lost their effectiveness at small cache sizes due to replacement. We proposed and studied the Adaptive scheduling policy that dynamically controlled the number of pending write requests in the buffer and hence the amount of replacement. This method worked well in improving performance by preventing the development of the race condition and block replacements at small cache sizes.
Chapter 4

Race Condition Solutions in Merge Sort

For the two-disk and two-job system, we presented various mechanisms to control the race condition in Section 3.3 and 3.4. In this chapter, we present simulation data on the effect of the race condition on multiple external merge sorts. The data show the effectiveness of these mechanisms in alleviating the detrimental impact of the race condition.

We use the Buffer Wait management policy that allows no replacement in the experiments whose results are presented in Figure 4.1. The values of the various I/O parameters that we used in these experiments are defined in Section 2.2. Figure 4.1 suggests that both the ReadPriority and the RoundRobin scheduling policies successfully control the development of the race condition. For 5 jobs, FCFS finishes at 105 seconds, and both ReadPriority and RoundRobin complete around 70 seconds, a 50\% improvement. In fact, the completion times of all jobs using either RoundRobin or ReadPriority disk scheduling is comparable to that achieved using the placement of Case 3 (see Figure 2.5).

When we experimented with the LRU Buffer Replacement policy and the Adaptive policy, we discovered that the mechanism broke down under very small cache sizes. Further examination of our simulation data indicates that there is substantial number of replacements. This is only possible when reads are much faster than writes; read blocks are being added to the cache at a rate faster than write ones are written out even under the Adaptive scheduling policy. The I/O model of the merge sort in a five-
Figure 4.1 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

disk system uses four read disks and one write disk for each job. In other words, each write disk has four disks feeding it. We proposed the strategy of Dynamic Prefetch Length, DPL, to slow down the reads and hence to reduce or eliminate the problem of replacement.

DPL works as follows. When the number of pending write requests exceeds an upper threshold, we reduce the number of read blocks in a prefetching chain by a half. When the number of writes drops below a lower threshold, we revert back to full-length prefetches.

Figures 4.2, 4.3, 4.4, and 4.5 show the performance of the Adaptive policy with and without DPL for 2, 3, 4, and 5 jobs. There are five independent disks in the system. The layout policy is that of Case 2, and the size of the cache varies from 2000
Figure 4.2  Completion time of 2 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy

Figure 4.3  Completion time of 3 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy
Figure 4.4  Completion time of 4 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy

Figure 4.5  Completion time of 5 jobs for the placement policy of Case 2 with 5 disks and the Adaptive scheduling policy
Figure 4.6  Completion time of all jobs for $2 \leq J \leq 5$ for the placement policy of Case 2 with 5 disks under the FCFS and the Adaptive scheduling policy with DPL.

To 24000 blocks. With small cache sizes, the Adaptive policy alone permits severe performance degradation. At a cache size of 2000 blocks, the merge sort finishes at 79 seconds; at a cache size of 8000 blocks, it finishes at 48 seconds, a 65% difference. With the aid of DPL, the Adaptive policy performs consistently well across the full range of cache sizes under consideration. Notice that there is a slight performance penalty at very small cache sizes. In Figure 4.3, the completion times are 50 and 46 seconds for the cache size of 2000 and 8000 blocks, respectively. The DPL scheme causes the difference in the following way. With a small cache, a job spends more time doing half-length prefetches, which makes the per block read time larger. As defined in Chapter 2, a prefetching chain of length 12 translates into 1.72 milliseconds of per block read time, assuming the average rotational latency of 8.33 milliseconds and the transfer rate of 1.02 milliseconds per block. A half-length prefetch means a
per block read time of 2.41 milliseconds, a 40% difference. As the cache size increases, a job spends more time doing full-length prefetches, and correspondingly less time on half-length prefetches. Thus, the completion time curve eventually flattens out.

In this chapter, we showed that both the RoundRobin and the ReadPriority scheduling policies are effective in controlling the development of the race condition in multiple merge sorts using the Buffer Wait policy. It was surprising to find that the Adaptive scheduling broke down at small cache sizes under the LRU Buffer Replacement policy due to block replacements. We proposed the Dynamic Prefetch Length scheme to slow down the reads of merge jobs. The combined use of the Adaptive scheduling and DPL proved to be a good strategy to prevent the race condition under the LRU Buffer Replacement policy with small to large cache sizes.
Chapter 5

Conclusions

The thesis presented the results of a systematic study of the performance of multiple and concurrent external merge sorts in a parallel I/O system with multiple disks. Specifically, we concentrate on the merge phase of external merge sort.

One important result is the discovery of an interesting phenomenon we call a race condition. Despite a balanced load and symmetry in the data placement, jobs "race" against each other, resulting in a significant serialization of disk accesses, which translates into performance degradation. In order to better understand the nature and the causes of the race condition, we studied the phenomenon in detail using a simple two-disk and two-job system with a buffer management policy that allows no replacements. We analyzed the race condition under assumptions of both finite and infinite cache sizes. Our theoretical model provides accurate predictions of the completion time of the two-job system under consideration with finite cache sizes. The data obtained from the model agree with those from simulation runs; the percentage errors are consistently less than 1% except at very small cache sizes where our assumptions of how the race condition behaves are not entirely valid.

We then studied disk scheduling policies to mitigate the detrimental effects of the race condition. Both the RoundRobin and the ReadPriority scheduling work fairly well to control the development of the race condition. However, when we switch to a new buffer management policy that allows LRU replacement, both scheduling strategies degrade performance due to an excessive number of buffer replacements.
To deal with the problem, we presented the Adaptive scheduling policy, which is a combination of FCFS and ReadPriority. The strategy is intended to control the growth of pending write requests in the buffer, and to guarantee enough buffer space left for read requests. It solves the replacement problem successfully.

We also studied the effect of data placement on the performance of multiple concurrent merge jobs. The layout policy of Case 2 results in a race among the jobs despite their complete symmetry. Consequently, we observed a severe performance degradation due to the serialization of job execution in this parallel I/O system. We then show that both the RoundRobin and the ReadPriority scheduling policies are quite effective in suppressing the development of the race condition in concurrent merge jobs under the Buffer Wait policy. Under the LRU Buffer Replacement policy, we demonstrate that the Adaptive policy works fairly well. Nevertheless, the scheme breaks down for very small cache sizes due to replacements. The strategy of Dynamic Prefetch Length is used in conjunction with the Adaptive scheme to solve the replacement problem with very small cache sizes.

The run placement (Case 2 policy) that leads to the race condition has an advantage; it can accommodate staggered arrivals of jobs. To avoid its inherent unstable behavior, we suggest the use of a RoundRobin or ReadPriority scheduling, or, under the LRU Buffer Replacement policy, the Adaptive policy in conjunction with the Dynamic Prefetch Length strategy. In the absence of such mechanisms, a policy that places the input runs equally among all disks (Case 4) is shown to achieve reasonable performance consistently for varying numbers of jobs.

Our study of multiple concurrent merge jobs assumes the absence of other kinds of I/O activity in the system. For future work, we would like to examine the effect on the performance of merge jobs when transaction-type I/O activities are present in the system. We also want to investigate the performance implications when different
merge jobs are in their run formation and merge phases and their interactions on each other.
Bibliography


