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.
Performance analysis of parallel I/O models for external mergesort

Pai, Vinay Sadananda, M.S.
Rice University, 1991
RICE UNIVERSITY

Performance Analysis of Parallel I/O Models for External Mergesort

by

Vinay Sadananda Pai

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

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

Dr. James Bartlett Sinclair
Associate Professor
Electrical and Computer Engineering

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

Houston, Texas
April, 1991
Performance Analysis of Parallel I/O Models for External Mergesort

Vinay Sadananda Pai

Abstract

Since the I/O subsystem is the bottleneck in external mergesort, I/O parallelism can result in substantial performance improvements. Concurrency can be introduced by overlapping I/O requests at different disks, and the average service time can be reduced by reading several blocks on each I/O operation. However, a RAM-based cache is required to support this prefetching.

A prefetching strategy is proposed in which one block from each run is read on every I/O operation, and a Markov model is developed to analyze the dynamic behavior of the system. The steady-state probabilities and a closed-form expression for the average parallelism are derived as functions of the cache size and the number of runs.

Several related prefetching strategies are also investigated for both single-disk and multiple-disk cases. For each strategy, the total execution time and the cache behavior are studied by simulating the merge and deriving analytic expressions for the average service time.
For Mama, Papa, and Mamama
Doing continually all actions whatsoever, taking refuge in Me, he reaches by My grace the eternal, undying abode.

Surrendering in thought all actions to Me, regarding Me as the Supreme and resorting to steadfastness in understanding, do thou fix thy thought constantly on Me.

Fixing thy thought on Me, thou shalt, by My grace, cross over all difficulties; but if, from self-conceit, thou wilt not listen (to Me), thou shalt perish.

—The Lord Krishna to Arjuna, from the Bhagavadgītā
Acknowledgments

I would like to thank Dr. Peter Varman, my research advisor, for his guidance, collaboration, and patience. My research topic evolved from his research in parallel merging and research interests in external algorithms. I am particularly grateful for his intensely thorough scrutiny of my thesis drafts. I would also like to thank the members of my thesis committee, Dr. J. Bartlett Sinclair and Dr. Joseph R. Cavallaro.

I would like to thank my parents for their support, guidance, and understanding; I owe my accomplishments entirely to their efforts as parents. I would also like to thank my grandmother and my brothers, Vivek and Vijay. Vivek arduously (and patiently) prepared all of the Matlab files necessary to graph the results of my simulations. Vijay painstakingly proofread the final drafts of my thesis, and yes, Vijay, you were correct about the proper punctuation of citations. Last, but not least, I would like to thank my fiancéé, Aarti, for her support and patience while I was writing my thesis.

During the “final three weeks,” I owe my sustenance to three factors: Vivek’s virtually limitless supply of snack food, the lunches packed by my grandmother, and my mother’s efforts to maintain a never-ending supply of caffeinated diet drinks.

This research was partially funded by a National Science Foundation Graduate Research Fellowship and a Rice University graduate fellowship. I am grateful to both of these institutions for their funding.

I would also like to thank the following individuals for their assistance: Nariankadu D. Hemkumar for answering my LaTÉX, Matlab, and Idraw questions and for con-
vicing me to choose these media for crafting my thesis; Kishore Kota for answering miscellaneous \LaTeX\ questions; Jim Carson for his \LaTeX\ and Unix knowledge; Mike Ingram for introducing me to this beast known as \LaTeX; Dr. J. Bartlett Sinclair and Bill Dawkins for answering miscellaneous CSIM questions throughout my research; Dr. Dave Johnson for his assistance in editing the thesis document-style; and the others I may have inadvertently omitted from this list.

I would also like to acknowledge the following musical artists who provided inspiration during my long hours at a workstation: Jim Morrison and the Doors, Pandit Ravi Shankar, Ustad Alla Rakha, Rush, the Police, INXS, Guns and Roses, A Flock of Seagulls, Ozzy Osbourne, Boston, the Cult, and the Talking Heads.
Contents

Abstract ii
Acknowledgments v
List of Tables x
List of Illustrations xi

1 Introduction

1.1 Disk Storage Devices 2
1.2 External Mergesort 4
1.3 Analyzing the Merge Phase 5

2 Survey of Related Work

2.1 Parallel I/O Architectures 8
2.1.1 Byte-Interleaving 8
2.1.2 Block-Interleaving 9
2.1.3 Hybrid Organizations 10
2.1.4 Performance of I/O Architectures 10
2.2 Prefetching 13
2.3 Previous Analysis of Merging 16
2.3.1 Merging from a Single Disk 16
2.3.2 Analysis of Data Placement 18
2.3.3 Data Models for Depletion 19
3 A Markov Model for a Parallel Disk Cache

3.1 Model Definition ........................................... 21
3.2 Categorizing States ........................................... 28
3.3 State Probabilities ........................................... 37
3.4 Counting States ........................................... 48
  3.4.1 Useful Summations ........................................... 48
  3.4.2 Theory of Partitions ........................................... 49
3.5 Average Parallelism on Each I/O Operation ................. 50
3.6 Simulation Results ........................................... 56
3.7 Conclusions ........................................... 59

4 Analysis of Prefetching Strategies ................................. 63

4.1 Introduction ........................................... 63
4.2 Overview of Prefetching Strategies ......................... 65
4.3 Simulation Model ........................................... 67
  4.3.1 Simulation Parameters ........................................... 68
4.4 Explanation of Simulation Statistics ......................... 69
4.5 One-Disk Results ........................................... 72
  4.5.1 Demand Run Only ........................................... 72
  4.5.2 “Demand Run Only”: Prefetching from a Single Run .... 74
  4.5.3 “All Runs”: Fetching from All Runs ....................... 77
  4.5.4 Single-Disk: Finite-Speed CPU .......................... 80
4.6 Multiple-Disk Results ........................................... 81
  4.6.1 Estimating the Total Execution Time ....................... 81
  4.6.2 D-Disk, “Demand Run Only” Strategy ....................... 85
  4.6.3 D-Disk, “Demand Disk, All Runs” Strategy ............... 87
4.6.4 D-Disk, “All Disks, One Run” Strategy ....................... 88
4.6.5 D-Disk, “All Disks, All Runs” Strategy ....................... 91
4.7 Summary ....................................................... 94

5 Conclusions .................................................... 103

Bibliography ..................................................... 105
Tables

3.1  The relative steady-state probability of each state category      39

4.1  The average time to fetch one block.      70
4.2  Simulation results for one disk.      73
4.3  Merging 5 runs using 5 disks. Each run contains 1000 blocks. A large
cache size was used to ensure that all prefetches were successful.      83
4.4  Merging 25 runs using 5 disks. Each run contains 1000 blocks. A
large cache is used to ensure that all prefetches are successful.      84
Illustrations

1.1 In order to determine the next record to be written to the final run, one block from each sorted run is required. .................................................. 5

3.1 The system model consists of $D$ disks, a RAM cache, and an infinitely fast CPU. Separate I/O subsystems exist for read and write traffic. ........................................... 22

3.2 The system model of the merge operation. .................................................. 24

3.3 A sample cache state for a system with $D = 5$ disks. .................................. 25

3.4 The sample cache state for $D = 5$ disks after a d-transition. ...................... 26

3.5 The sample cache state for $D = 5$ disks after an f-transition. ...................... 27

3.6 The sample cache state for $D = 5$ disks after an r-transition. ...................... 27

3.7 The entire Markov chain for the case of three disks and seven cache blocks. ................................................................. 29

3.8 $I_1$: Interior state with exactly one part equal to 1. .............................. 41

3.9 $I_i$: Interior state with $i$ parts equal to 1. ............................................ 41

3.10 $I_D$: Central interior state with all $D$ parts equal to 1. .......................... 42

3.11 An edge state with no free cache blocks. Exactly one part can be equal to 1. ................................................................. 42

3.12 A state in $E_{f,1}$ with $f \geq 1$ free cache blocks and 1 part which is 1. .... 44

3.13 A state in $E_{f,k}$ with $f \geq 1$ free cache blocks and $k \geq 2$ parts which are 1. 44
3.14 A state in $E_{f,f+1}$ with $f \geq 1$ free cache blocks and $f + 1$ parts which are 1. ................................................................. 46
3.15 Predicted $\varphi$ for small values of $D$. .......................... 57
3.16 Predicted $\varphi$ for large values of $D$. ......................... 58
3.17 Predicted and simulated $\varphi$ for $D = 3$ and $D = 5$. .......... 60
3.18 Predicted and simulated $\varphi$ for $D = 10$ and $D = 25$. ....... 61
3.19 Predicted average parallelism. ............................... 62

4.1 The effect of fetching $N$ blocks on the total time for the single-disk case. In both graphs, the cache is large enough to ensure that all prefetches are successful, i.e., the success ration is 1. ............... 76
4.2 Single-disk "All Runs": Cache performance as a function of $N$. .... 79
4.3 The performance of single-disk prefetching strategies when a finite-speed CPU is used. The cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1. ............... 81
4.4 Cache requirements for the "demand disk all runs" strategy, as $N$ varies. 25 runs, 5 disks. ................................. 89
4.5 Cache requirements for "all disks one run", D=5 disks ............ 92
4.6 All Disks All Runs: The effect of the cache size on the success ratio. 25 runs, 5 disks. ................................. 95
4.7 The effect of a finite-speed CPU on the D-disk prefetching strategies. In all cases, the cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1. ............... 97
4.8 The performance of the $D$-disk strategies when fetching $N$ blocks. The cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1. ............... 98
4.9 Comparison of prefetching strategies for $D = 5$ disks, $k = 5$ runs.

The cache is large enough to ensure that all prefetches are successful,

i.e., the success ratio is 1. .................................................. 100

4.10 Total time as a function of cache size. $N = 1$ and $N = 5$. $k = 25$ runs. 101

4.11 Total time as a function of cache size. $N = 10$. $k = 25$ runs. .......... 102
Chapter 1

Introduction

In several applications sorting comprises a significant, if not dominant, portion of the total execution time. For example, relational database systems [20] often need to sort the intermediate data during a query operation in order to extract those records which match given selection criteria. Sorting is also dominant in I/O-intensive and compute-intensive relational database operations such as join, which uses the sort-merge-join algorithm. A third need for sorting in database systems arises in determining the union and intersection of indexes (record pointers) in queries involving several predicates. The performance of the sorting step in this case is crucial to the performance of the entire query operation.

One particular sorting algorithm, mergesort [11], sorts a list of records by first creating several lists of sorted elements and then merging elements from the lists into one final list. A record is a logical unit of data which contains one or more fields: one of these fields is termed the key. Sorting a list of records involves rearranging these records according to some specified ordering of the key field. For notational convenience, the records will be assumed to be sorted by increasing order of keys. When the data size is small enough such that all records can fit into main memory, the records can be sorted internally. However, if the total data size exceeds the main memory size, the records must be moved repeatedly between I/O devices and main memory, and an external sort must be performed. With currently available hard-disk drive technology, the I/O subsytem is the bottleneck in an external sort.
Measures that aim to increase the speed of external sorting should necessarily address the problem of reducing the total time spent in servicing I/O operations.

Although a substantial body of existing work analyzes parallel techniques for internal sorting (e.g., refer to [21] for a discussion of recent approaches), the analysis of I/O parallelism for external mergesort has received little attention.

1.1 Disk Storage Devices

Even though internal and external mergesort strategies are conceptually similar, the disparity in access times between memory and disk storage results in certain important differences. For instance, two-way merges are suitable for internal strategies but possibly disastrous for external sorts, since data must be moved several times between main memory and the I/O devices. Unless the I/O operations are efficiently managed, the time spent servicing I/O requests can seriously degrade the performance of the entire mergesort.

Currently, disk storage is used as the predominant means for external storage, for reasons of performance and cost. Random access to data is supported with current disk technology, but the access times are not necessarily uniform.

Although main memory and disk storage can be viewed conceptually as a large array of bytes, the implementation of disks differs considerably from this conceptual view. A disk drive consists of a number of circular platters which are coated with magnetic media on both surfaces. A number of these platters are stacked on a central spindle, and the spindle is spun continuously at a constant rate [7]. Associated with each surface is a head which can read or write data to that surface only. These heads are all attached to an arm which is driven by a common servo actuator.
Each media surface consists of concentric, circular tracks. A cylinder $i$ consists of the corresponding track $i$ from each surface. Each track is logically partitioned into equal-sized portions known as blocks. The block is the smallest allowed unit of data transfer, and each block holds a fixed quantity of data, usually ranging from 512 to 4096 bytes, depending on the particular implementation [10] [18]. Since a block is typically larger than the size of one record, several records will be placed contiguously in each block.

The service time required to read or write a particular block consists of three access delays. First, the heads must seek to the appropriate cylinder. This seek time can be approximated by a linear function of the seek distance in cylinders [13] [18]. If the heads are already positioned over the correct track, no seek is required. The second access delay is the rotational latency. Since the platters are constantly spinning, the heads must wait until the appropriate block spins under the heads before beginning a read or write. If a request is issued for $n$ consecutive blocks from a track, the first block may encounter a rotational latency, each of the next $n - 1$ blocks can be read sequentially without any additional rotational latency. The final access delay, the transfer time, is the time required to read or write the block which is positioned under the heads. The transfer time is simply the product of the transfer rate and block size, and this transfer rate, in turn, is the product of the rotational velocity of the platters and the data density of the media surface [7].

Disk access times are substantially longer than the ~100 nanosecond access times of main memory [17]. Average seek times range from 2 to 50 milliseconds, depending on the particular implementation. The average rotational latency is typically 8.3 ms, since 3600 rpm is a common speed for the spindle motors, and transfer rates range from 1 to 3 megabytes per second [7] [10] [18].
Since an access to internal memory can be serviced in constant time, internal sorting algorithms need only minimize the number of memory accesses. On the other hand, the access time for a disk depends on the previous position of the disk heads, the previous block which passed under the heads, and the interval of time in which the disks were last accessed (since the platters are continuously spinning). As a result, an efficient external algorithm should attempt to minimize the average access time for an I/O request, as well as the number of I/O requests.

1.2 External Mergesort

When sorting external data which exceeds the size of main memory, the disk will be used to store intermediate data. Portions of the entire data will be retrieved into memory, sorted internally, and written to disk as a run. Each run consists of a sequence of sorted records and occupies a range of consecutive blocks on disk. Within a run, block \( i \) contains records with smaller key values than those records in block \( i + 1 \). After all of the records have been processed, a number of independently sorted runs will exist on disk.

These runs must now be collectively merged into one final sorted run. Figure 1.1 depicts this final merge step. In order to form one final sorted run from these individual runs, the next block from each run must be resident in memory. Records are extracted in sorted order from these memory-resident blocks and placed into the output blocks which will be written to disk as the final sorted run. After a record with key value \( x \) is extracted, every subsequently extracted record will have a key value \( y \) such that \( y \geq x \).

As the records are extracted, all of the records in one block will eventually be depleted. When a block is depleted, the next block from that run must be fetched
Figure 1.1: In order to determine the next record to be written to the final run, one block from each sorted run is required.

before the merge can continue. In order to determine the next record to place in the final run, the smallest unprocessed record from each of the $D$ runs is required. Therefore, each run must have at least one block resident in memory.

In this manner, the merge continues until all records from all runs have been retrieved, processed, and written to the final sorted run. The merge order determines the number of runs that are merged in one pass. If the number of runs is very large, several passes of this merge process will be required to obtain the final sorted run.

1.3 Analyzing the Merge Phase

The first portion of the mergesort which creates the individual sorted runs can be implemented in a straightforward manner. Records are read sequentially from the original input, sorted in memory, and written sequentially to the final output. Optimal algorithms for internal sorting have been well understood; the I/O requirements are straightforward, since all accesses are sequential in nature. However, the merge phase
which creates one sorted run from the independent runs does not exhibit this sequential access pattern. As a result, different strategies for this final merge phase can be compared by analyzing their relative I/O performance.

Through efficient parallel partitioning techniques, optimal internal merging algorithms have been developed for multiprocessors [21]. Parallelism can be introduced in the I/O subsystem by developing architectures with multiple disk units. In order to effectively utilize these new parallel architectures, data must be optimally partitioned for the external merge problem. Since a greater I/O bandwidth is now available, blocks which will be required in the near future can be fetched along with each I/O request. With the introduction of this prefetching, however, a memory-based cache is required to store these prefetched blocks until they are required. A successful integration of these approaches will simultaneously allow concurrent I/O operations and reduce the average service time for an I/O operation.

The remainder of this thesis will be organized as follows. In Chapter 2, current work on parallel I/O architectures will be summarized, and a suitable architecture and data placement scheme will be developed for external merging. In Chapter 3, an analytic model will be developed for a particular I/O model and prefetching strategy. Finally, in Chapter 4, additional prefetching strategies will be developed, and the I/O performance of these strategies will be analyzed through simulation.

The performance of the merge phase of external mergesort will be analyzed for these parallel I/O models. In particular, the behavior of these models for various cache and data sizes will be analyzed. A Markov model will be developed and analyzed for some of these prefetching strategies. This model will predict the concurrency in I/O that is achievable for a given cache size. For the other prefetching strategies, the cache requirements will be determined experimentally through simulation. For all prefetching strategies, analytic expressions for an I/O request's average service
time will be developed. These expressions will be used, in turn, to predict the total execution time.

Simulation will be performed using CSIM [4], a discrete-event or process-oriented simulation package. CSIM supports theoretic queuing models and concurrent processes. By modeling a CPU, cache, and disk system, the I/O requirements of the merge phase can be simulated at the block level. The interaction of architectures, interleaving schemes, and prefetching strategies can be studied by monitoring each I/O operation.
Chapter 2

Survey of Related Work

2.1 Parallel I/O Architectures

In order to reduce the average I/O service time, two approaches have been studied. The first approach groups all available disks into a tightly coupled array which is subsequently accessed as one logical disk unit. Data is byte-interleaved across the disks in the array to provide increased I/O bandwidth. The second approach introduces concurrency by dividing multiple-block requests into single-block requests at different disks. The data is block-interleaved across the disks in order to reduce the variance in disk queue lengths. Both of these strategies differ from a traditional system in which a file resides on only one disk.

2.1.1 Byte-Interleaving

Kim [10] and Gibson [5] [6] propose an arrangement of disks in which data is byte-interleaved across \( n \) disks. In this scheme, byte \( i \) of a block is located on disk \( i \mod n \). All disks in the array are used to service a request for a single block, and access to the individual disks in the array is not permitted. Since the \( n \) disks can transfer data simultaneously, the transfer rate of the array is now \( n \) times that of a single disk. The seek time and rotational latency, however, are unaffected. Through this reduction in transfer time, disk arrays attempt to provide faster response times than conventional disk systems. Additionally, the disk array is viewed as a logical unit by both the operating system and the user.
Kim [10] refers to a group of disks arranged in this manner as *synchronized disks*. The *degree of synchronization* (ds) is defined as the number of disks in the array. Gibson refers to such an array which incorporates data redundancy as a *redundant array of inexpensive disks*, or RAID for short.

Other means of exploiting greater transfer rates from a single disk have not been entirely successful. One such attempt was a track-parallel disk in which parallel reads were allowed from all tracks of a cylinder, but aligning multiple read/write heads concurrently [10] was not possible. Future improvements in a single disk's performance do not invalidate any of the interleaving schemes presented, since these schemes build configurations with standard disks. Any improvements to basic disk technology can be used in conjunction with the benefits provided by these parallel configurations of disks.

Byte-parallel interleaving can provide an enhanced transfer rate and simultaneously be implemented in the controller hardware [10]. The disk controller must now support conversion of the data and synchronization of the motors. Conversion is required to assemble the parallel data streams from the disks into a serial data stream and vice versa. The rotation of all disks must be synchronized so that bytes can be written to the same location on each disk. Each disk's spindle will have its own motor, but all the motors will controlled by a central clock with a feedback control loop. The spindles will spin synchronously and the arms will all seek together. Now, these synchronized disks can be treated as a single logical disk unit.

### 2.1.2 Block-Interleaving

In a *declustered* system, a file is interleaved blockwise across the disks [16]. Contiguous blocks are interleaved in a round-robin fashion across all $D$ disks such that block $b$ will be found on disk $b \mod D$. Since the unit of interleaving is a file block rather
than a byte, this organization allows multiple single-block requests to be serviced concurrently. The degree of declustering (dd) is defined as the number of concurrent accesses that can be made to a file. Since the data is interleaved, the request rates at the disks will generally be balanced for multiple-block requests.

In traditional systems, a request for consecutive blocks incurs a single seek and rotational latency for the entire request. With the declustered system, however, the multiple-block request is decomposed into a separate request at each declustered disk. Now, each request will incur a seek and rotational latency. The hope with the declustered system, however, is that the variance in queueing times found in traditional systems can be reduced substantially.

2.1.3 Hybrid Organizations

Although more disks can be added to an array to increase the array's bandwidth, the controller will eventually become the bottleneck. Since RAID systems as proposed have only a single controller, they do not support multiple, simultaneous large transfers at peak speed as is possible with multiple controllers. Therefore, a single RAID may not be an appropriate solution in all circumstances.

Reddy and Bannerjee [18] consider a hybrid system composed of several declustered units in which each unit is possibly an array of synchronized disks. In this hybrid arrangement, each file is block-interleaved across these synchronized units. These systems will be referred to as declustered-synchronized systems. Each disk organization is classified by \((dd,ds)\), \(dd \geq 1, ds \geq 1\).

2.1.4 Performance of I/O Architectures

Reddy and Bannerjee [18] studied two "model" workloads which attempt to characterize the typical request. Consisting of many small requests, the first workload
characterizes usage for a multi-user file system and transaction-based system. The second considers single-user operation of a multiprocessor executing scientific applications which issue a few requests for large files. In the first workload, the average response time is the performance measure of interest, whereas in the second workload, the total execution time is the desired metric.

In the transaction workload, the I/O arrival rates are assumed to be distributed exponentially, and the request rate is varied until the utilization of all disks approaches one, i.e., the disks are saturated.

For the scientific workload, the I/O request patterns for matrix-matrix multiplication, matrix-vector multiplication, and the fast Fourier transform were considered. For these computationally intensive algorithms, the I/O transfers were characterized by large, sequential transfers which occurred infrequently. Since the results of the scientific workload depend on the specific I/O patterns of these particular problems, the results did not conclusively indicate which approach would be best in general for external merging. With multiway merging, the data is read only once, and I/O is requested frequently, rather than after long intervals of computation. Additionally, since blocks are being merged from runs which are spread across the disks, the transfers are not sequential. For mergesort, the access patterns more closely resemble those of the transaction workload.

Kim [10] studies the performance of a fixed arrangement of byte-interleaved disks while varying the transfer size. As the block size increases, the transfer time becomes the dominant component of the service time. The reduction in transfer time promised by synchronized disks becomes more noticeable as the block size increases. However, Kim does not consider any declustered arrangements.

Reddy and Bannerjee [18] varied both \textit{dd} and \textit{ds} between 1 and 16 to consider various hybrid organizations.
At low request rates, the \((dd = 16, ds = 1)\) system performs better than \((2,1), (4,1),\) and \((8,1)\), since it has the most parallelism available to serve a given request. At higher request rates, the \((16,1)\) performs worse than the other three systems, because the disk utilization increases. As the degree of declustering is decreased, the additional seek and rotational latency penalties are reduced, but the amount of parallelism available for serving a request is correspondingly reduced. At low request rates, the parallelism dominates since the utilizations are low. At higher request rates, however, the higher disk utilizations result in higher waiting times, resulting in relatively poor performance of \((16,1)\). Note that as the utilization increases, the waiting times will increase.

These seek time and rotational latency penalties are present because a single request for \(m\) contiguous blocks is divided into \(m\) separate requests for one block each. When the \(m\) blocks are processed as a single request, only one seek and rotational latency are required, but the \(m\) separate requests will each require a seek and rotational latency.

When the unit of declustering is smaller than a block, the system is said to employ sub-block declustering. Sub-block declustering is shown to be useful for larger block sizes [18], implying that smaller block sizes are useful for declustering. Maintaining a small block size may effectively eliminate the need for sub-block declustering.

Synchronized systems \((1,16), (1,8),\) and \((1,4)\) perform better than other systems at very low request rates, since even single-block requests can exploit the full parallelism of the disk array. In the declustered organizations, only multiple-block requests can utilize the parallelism. As a result, the synchronized systems perform better than declustered systems at lower request rates.
At low request rates, the organizations that use both synchronization and declustering perform worse than their purely synchronized counterparts and better than their purely declustered counterparts.

When the interarrival time is less than 20 ms— the case of high request rates—the synchronized systems are so heavily loaded that the response time is far worse than the response times for other systems. The synchronized systems suffer from greater queue lengths, and consequently longer service times. Declustered systems fare better than their synchronized counterparts at these high workloads.

Because external merging is an I/O bound problem, the interarrival time of disk requests should be quite small. Since declustered systems are more resilient to high workloads, declustering will be the more favorable approach for external merging. Additionally, the data can be interleaved at the run level in which all the blocks from a run reside entirely on one disk. In this manner, contiguous blocks from a run can be serviced as one request, and blocks from different runs can still be fetched in parallel.

2.2 Prefetching

A study of the effects of prefetching in a parallel environment by Kotz and Ellis [12] shows that:

- The disk cache's hit ratio is not an adequate measure by itself in determining the performance of parallel file access patterns.

- Caching with prefetching does significantly improve the hit ratio and the average time for I/O operations.

- In most cases, the overall execution time is improved, but in a few cases, the time increases.
Kotz and Ellis [12] assert that synchronized disk systems have limited prefetching capability and that parallelism in the I/O is better achieved by using multiple disks and possibly multiple controllers. One proposed parallel disk architecture involves multiple conventional disks which are addressed independently, since the disks are attached to separate processors. A file may be spread over several disks, but since the disks are connected to separate processors, a bottleneck at the channel is avoided.

Prefetching is shown to be useful on supercomputers and multiprocessors, since the I/O for scientific applications is characterized by sequential accesses and a need for minimum latency [17]. However, the file access patterns for these systems differ from uniprocessors, since concurrent processes may be accessing disjoint portions of data [12]. In fact, these file access patterns play a role in determining the prefetching strategy.

The effectiveness of prefetching is measured by two metrics: the overall execution time and the average time to read one block. When a prefetch can overlap with computation, that block is added to the cache. The average time to read one block is reduced as more blocks are read from the cache, since a greater portion of the I/O requests is overlapped with computation.

In this study, file access patterns for multiprocessors are divided into two broad categories: random and sequential. Random patterns, however, are not amenable to prefetching, and no strategies are proposed for these patterns. All sequential patterns consist of a sequence of accesses to sequential portions or runs. In some cases, however, the pattern of accesses may only seem sequential from a global perspective.

Additionally, the sets of data accessed by the processors may or may not intersect. As a result, these access patterns will be further classified as overlapped or disjoint. In addition, the length of sequential portions may be regular, thereby allowing the end of the portion to be calculated. The distance between the last block of one portion
and the first block of the next portion may also be regular (a stride). In both cases, the exact bounds of prefetching can be determined.

The external merge falls into this category of sequential accesses with fixed-length portions, since blocks can be prefetched from each run. For this category, prefetching is shown to be quite effective. The average block read time is reduced an average of 48% by prefetching in all cases studied [12]. Furthermore, the hit ratio was greatly improved in all prefetching experiments. In all cases, it was over 0.69, and in more than half of the cases it was over 0.86. Due to the sequential nature of the experiments, the hit ratio was nearly zero in all experiments without prefetching; in other words, blocks were almost never used more than once.

If a demand fetch is a cache hit, the data in the cache may still not be complete; if the I/O for the prefetch has not finished, the demand fetch will have to wait. This waiting period is called the hit-wait time; hits with a positive hit-wait time are termed unready buffer hits, whereas ready buffer hits have a zero hit-wait time.

The number of requests serviced by each disk does not increase with prefetching. The requests are simply issued in a smaller interval of time. Even when the execution time does not decrease, the disk reads tend to be issued nonuniformly in time, creating periods of high disk contention. Without ensuring that the prefetching benefits are evenly distributed, however, any reduction in the average block read time does not guarantee a reduction in the total execution time.

As the CPU speed increases, however, the degree of overlap of the I/O and computation decreases. The asymptotic case of an infinitely fast CPU will reveal the lower bound on the I/O time for a given prefetching strategy.
2.3 Previous Analysis of Merging

Considerable work pertaining to internal merging and sorting exists [2] [21], but little work has focused on parallelism in the I/O. Most work which analyzes the performance of external mergesort has treated only the single-disk case [13] [19].

Recent work shows that a one-pass mergesort meets the lower bound in the number of I/O operations required for sorting [1]. Aggarwal and Vitter [1] show that mergesort is an optimal external sorting method (up to a constant factor in the number of parallel I/O’s) if the next $P \geq 1$ blocks which will be merged can be prefetched concurrently. In order to determine which blocks will be required next, each block includes $P$ end-markers which indicate the largest record in the next $P$ blocks in the same run. By comparing the end-markers from each block, the next block can be determined. However, their model assumes that the I/O system can always transfer an arbitrary $P$ blocks in one operation. The placement of data across the disks and the degree of declustering are not considered by this study.

2.3.1 Merging from a Single Disk

Kwan and Baer [13] study the multiway merge of $k$ sorted runs which uses a single disk. A block-depletion model is assumed in which blocks are depleted randomly from one run at a time. Newly fetched blocks and older blocks have equal probability of being depleted.

In each merge pass, $k$ runs are merged into a larger run. One buffer of memory is allocated for each run, and runs are read into these buffers on a block-by-block basis. To eliminate contention between the read and write traffic, two storage devices—one for reads and one for writes—are assumed to be available [13] [19]. Therefore, the read and write traffic can take place concurrently. Since the writes are sequential,
seek time will not be an issue with the writes. Therefore, the merge activity will be analyzed by studying the read access time.

Kwan and Baer [13] show that the cost of the merging phase increases with the degree of merging, $k$. Although the number of merge passes decreases as $k$ increases, the cost of a merge pass increases with $k$. Reading $k$ contiguous segments in random order is shown to require $(k^2)/3 + O(k)$ moves, where a move is defined as the cost of travelling the length of one segment. In other words, an average of $\frac{k}{3}$ segments are skipped between successive read operations.

Tag sort is proposed as a more efficient alternative to mergesort when the size of each key is small in comparison to the record size. Tag sort consists of three phases: extracting the keys from all records, sorting the keys, and constructing the sorted output file. Since the key size is small in comparison to the record size, sorting the key file should be much less expensive than sorting the entire file.

The analysis which leads to this conclusion centers on the assumptions made about the disk service time. Seek time and rotational latency are assumed to dominate the service time, but no measures to reduce these components are investigated. If these access delays can be reduced, the transfer time will dominate the service time.

The fact that the seek time grows linearly with $k$ is crucial to this analysis [13]. As will be shown in Chapter 4, the seek time can be made independent of $k$ if one block from each run is fetched on each I/O operation. Also, strategies which reduce the average service time even further are discussed in Chapter 4.

McCulloch [14] also proposes an alternative to mergesort. The same assumption about the dominance of access delays is used to assert that multiple passes which reduce these delays can minimize the total execution time.

As shown in more recent work [1], mergesort is an optimal strategy in the number of I/O operations. As a result, these alternative merging schemes will not be
considered. Salzberg [19] shows that a merging method which makes only one pass of the data is optimal if the transfer time is the dominant component of the service time. The transfer time component grows linearly with the block size, but the seek time and rotational latency component are unaffected. With larger block sizes, the transfer time component becomes the significant component. This study, however, analyzes the performance when only a single disk is used to store all of the runs.

2.3.2 Analysis of Data Placement

As Kwan and Baer show for the single-disk case [13], an average of $\frac{km}{3}$ cylinders will be skipped on each fetch, where $m$ is the length of each run in cylinders. When $D$ disks are used and data is block-interleaved, each run is uniformly distributed across all $D$ disks, and, as a result, each run occupies only $\frac{m}{D}$ cylinders. The length of each run on a disk has been reduced by a factor of $D$. Therefore, an average of $\frac{km}{3D}$ cylinders will be skipped on each fetch. Using this block-declustered arrangement, the total seek distance is reduced by a factor of $D$.

When the data is block-interleaved in this manner, each disk contains an equal number of blocks from each of the $k$ runs. Fetching the next block from each run may cause contention for the same disk.

Placing all of the blocks from a run on a single disk seems more promising. In other words, blocks are interleaved at the run level. Run $r$ will reside entirely on disk $r \mod D$. All blocks in a run will occupy contiguous blocks on its disk. If the total number of runs $k = D$, then no random seeks are ever required. As a result, each run can be read sequentially. The only seeks required are those which cross a cylinder boundary when reading a run sequentially.

When $k > D$, the average seek distance matches that of the block-interleaving case. Each disk contains $\frac{k}{D}$ runs, each run occupying $m$ cylinders. The average seek
distance is simply \( \frac{kD}{3D} \) cylinders. If one block from each run must be accessed, the \( k \) I/O operations can be completed in a total of \( \lceil \frac{k}{D} \rceil \) steps, since \( D \) operations can be processed concurrently.

Since synchronized disks can be viewed logically as a single disk with an increased transfer rate, the effects of synchronization will not be considered as a separate case. Any parallelism inherent to synchronization is orthogonal to the parallelism introduced by run-level interleaving. Since interleaving at the run level shows more promise than block interleaving, data will be assumed to be interleaved at the run level in all of the prefetching strategies studied in this thesis.

### 2.3.3 Data Models for Depletion

In order to analyze the I/O operations during messsort, a data model must first be chosen. Two different models are often used. In the first model, blocks are assumed to be depleted randomly from the runs. This model will be termed the block-depletion model. The second model assumes that records are depleted randomly from the runs; the order of block depletion will, therefore, depend on the how the records themselves are depleted. This model will be termed the record-depletion model.

1. With the block-depletion model, introduced by Kwan and Baer [13], newly fetched blocks and partially depleted blocks have equal probability of being depleted. As a result, this model tends to overestimate the cache requirements. With this model, the probability that older blocks remain in the cache is higher than it should be. As a result, a larger cache size is expected. However, this model is still realistic, since data across the runs are usually not distributed in an entirely random fashion.
2. The record-depletion model, implicitly used in the work by Salzberg [19], tends to underestimate the cache requirements. In this model, all records are equally likely to be depleted, since the data are assumed to be randomly distributed. Therefore, blocks from all runs should deplete at nearly the same rate. When a block is depleted, the blocks being processed from the other runs will also be nearly depleted. After the fetch, these partially depleted blocks will more likely be depleted before the newly fetched block. In other words, older cache blocks have a higher probability of being depleted than do newly fetched cache blocks. This model, however, is too optimistic, since it implies that a cache size of $O(k)$ blocks will suffice in all cases, in which $k$ is the merge order.
Chapter 3

A Markov Model for a Parallel Disk Cache

3.1 Model Definition

In studying the dynamic behavior of the merging of several runs, an analytic model can be developed to study the cache. In this particular approach, the system model will consist of an infinitely fast CPU, a RAM-based disk cache with a capacity of $C$ blocks, and $D$ disks which contain exactly one run each. On a single I/O operation, $D$ blocks, one from each disk, can be read and transferred in parallel. Figure 3.1 shows the architecture of this system model. This study will concentrate on the I/O performance of the parallel merge. Since CPU time required to process one block is typically two orders of magnitude faster than the I/O time required to fetch one block, CPU activity does not drastically affect the total execution time of the merge phase. Therefore, CPU activity can be ignored without rendering the model unrealistic.

Whenever a block is depleted from a run, the cache will be checked to see if the next block from that run resides in the cache. If that block is not found in the cache, an I/O operation is required. The block that is required will be referred to as the demand-fetch block. In order to exploit the parallelism possible with the $D$ disks, the next block from each of the other $D - 1$ runs will be fetched along with the demand-fetch block. A read of up to $D$ blocks, one from each disk, will be charged unit cost. Note that this model neglects the reduction in seek time that arises when the fixed number of runs are spread across the $D$ disks. In general, the reduction in
seek time, coupled with the parallel access of the disks, could result in superlinear speedup. This effect is discussed in Chapter 4.

On a fetch, if the cache has enough room to accommodate all $D$ blocks, they will be added to the cache. Otherwise, only the demand-fetch block will be added to the cache.\footnote{Even though more than one cache block may be available, only one block is fetched. In more recent work with A. Schaffer and P. Varman, we show that adding only one block, rather than adding as many as possible, results in greater parallelism.} In this manner, if the full parallelism of the $D$ disks can be used effectively, the I/O cost of the merge operation will be reduced by a factor of $D$ over the single-disk case. In the ideal case, the total I/O cost is now identical to the time required to read one run from start to finish. Under this system model, a speedup of $D$ is the best that can be obtained.

In order to study the dynamic behavior of the cache, a random model of block depletion is assumed. For each run, the number of blocks which are currently resident in the cache will be tracked. At any stage of the system, at least one block from each of the $D$ runs will be present in the cache.
The leading block from each run is a potential candidate for being depleted next. In this model, any one of these \( D \) leading blocks is chosen with equal probability \( \frac{1}{D} \). At any stage, a cached block from a randomly chosen run is depleted. If that run no longer has any cached blocks, an I/O operation will be required to retrieve the next block from that sorted run. Note that at least one block from each run must be present in the cache to determine the next merged record. By tracking the number of cache blocks allocated to each run, a Markov model to compute the effective I/O parallelism is developed.

The program listed in Figure 3.2 describes the system model. This system can be modeled mathematically by a Markov chain. Each state shall represent the number of cache blocks allocated to each run at the start of each iteration of the loop in Figure 3.2. Each execution of the loop causes a transition to a different state. A state \( s \) can be represented by a \( D \)-tuple, \( s = [s_1, \ldots, s_j, \ldots, s_D] \), in which \( s_j, 1 \leq j \leq D \), represents the number of blocks from run \( j \) present in the cache. Figure 3.3 shows a sample cache state for \( D = 5 \) disks. By inspecting the program from Figure 3.2, one can easily observe that \( s_j \geq 1 \) for \( 1 \leq j \leq D \). Each \( s_j \) will be referred to as a part\(^2\).

In order to study the Markov chain in greater detail, the following notation will be useful.

**Definition 3.1** Let \( s = [s_1, \ldots, s_i, \ldots, s_D] \) be a system state. Given a cache size\(^3\) of \( C, C \geq D + (D - 1) = 2D - 1 \), define \( \delta(s) = (C - \sum_{i=1}^{D} s_i) \) to be the number of free cache blocks in state \( s \). Define \( \gamma(s) \) to be equal to the number of \( s_i, 1 \leq i \leq D \), which are 1.

\(^2\)The notation of part is borrowed from partition theory. In subsequent sections, partition theory will become useful in analyzing the model.

\(^3\)Since \( C \geq 2D - 1 \), all three categories of transitions are possible.
Initial State:

Add the first block from each run to the cache.
\[ a[i] = 1, \quad \text{for } i=1,\ldots,D \]

\* a[i] represents the number of blocks
\* from run i which reside in the cache.
\*
\num_free_cache = C-D;

Do forever
{
    /* RECORD STATE HERE */

    Randomly choose a run, j, from which to deplete a block.
    \[ a[j] = a[j]-1; \]
    if (a[j]==0)  /* A run has emptied */
        if (num_free_cache >= (D-1)) {
            Fetch D blocks.
            for(i=1,\ldots,D) a[i] = a[i]+1;
            num_free_cache = num_free_cache - (D-1);
            /* Note that only (D-1) new prefetches have
            * been added. The demand-fetch replaces
            * the cache block depleted from run j.
            */
        }
    else    {
        Fetch 1 block from run j;
        a[j] = a[j]+1;
        /* The demand-fetch replaces the cache block
        * depleted from run j. */
    }
}
else    { /* run not emptied */
    num_free_cache++;
    /* depletion without any fetch */
}
}  

Figure 3.2: The system model of the merge operation.
The transitions from state $s = [s_1, \ldots, s_i, \ldots, s_D]$ to state $t$ can be characterized as follows:

1. **Depletion (d-transition):** A block is depleted from run $i$, and all runs still have at least one cached block. This transition is enabled only if $s_i > 1$. Following the transition, $t = [s_1, \ldots, s_i - 1, \ldots, s_D]$.

2. **Fetch (f-transition):** A depletion from run $i$ occurs, and run $i$ no longer has any cached blocks. A fetch is required, and all $D$ blocks are cached. This transition is enabled only if $s_i = 1$ and $\delta(s) \geq D - 1$. Following the transition, $t = [s_1 + 1, \ldots, s_{i-1} + 1, 1, s_{i+1} + 1, \ldots, s_D + 1]$.

3. **Replenish (r-transition):** A depletion from run $i$ occurs, and run $i$ no longer has any cached blocks. However, the cache does not contain enough free blocks for a fetch of $D$ blocks. Instead, only one block from run $i$ is fetched. This transition is enabled only if $s_i = 1$ and $\delta(s) < D - 1$. Following the transition, $t = s$. In other words, this transition is a "self-loop" which returns to the same state.

Figure 3.3: A sample cache state for a system with $D = 5$ disks.
Figures 3.4 through 3.6 show the resulting cache state after a transition from the original state shown in Figure 3.3. Figure 3.4 shows a d-transition which occurs after a block is depleted from the first run. After the depletion, the first run still contains additional cached blocks; as a result, no fetch is required.

If a depletion were to occur from the second run in the cache state from Figure 3.3, an I/O operation would be required to fetch the next block. If the cache contains enough free blocks to fetch \( D - 1 \) additional prefetches, the next block from each of the \( D \) runs can be fetched. Figure 3.5 shows this f-transition.

On the other hand, if the cache did not have enough free blocks to fetch \( D \) blocks on an I/O operation, only the demand-fetch block would be fetched. This r-transition occurs after depleting a block from the second run in the cache state from Figure 3.3. Figure 3.6 shows the resulting state after this r-transition.

Every state will have exactly \( D \) possible transitions to the next state. Each of these transitions is equiprobable with probability \( \frac{1}{D} \), since the transition from the current state is determined uniquely by the choice of the run which is depleted.

![Diagram](image)

**Cache state:** \([5, 1, 4, 2, 5]\)

**Figure 3.4:** The sample cache state for \( D = 5 \) disks after a d-transition.
Figure 3.5: The sample cache state for $D = 5$ disks after an f-transition.

Figure 3.6: The sample cache state for $D = 5$ disks after an r-transition.
The steady state probability of each state and the number of states will determine the cache’s steady-state behavior. Initially the cache will be loaded with one block from each run, i.e., the initial state is assumed to be $[1,1,\ldots,1]$.

The $D$ independent disks will be able to fetch blocks from independent disk locations, but they will be assumed to initiate and complete I/O operations as a group. This assumption relaxes any temporal transients without invalidating the cache behavior being studied.

Figure 3.7 shows the entire Markov chain for the case of three disks and seven cache blocks. The transitions are explicitly shown, and three invalid states are also included. These invalid states will be explained subsequently in Section 3.2. The Markov chain for three disks lies in three mutually orthogonal planes, each plane representing depletions from a different run.

### 3.2 Categorizing States

In this section, the state space of the Markov model which was discussed earlier will be characterized.

**Definition 3.2** A valid state is one that can be reached by some sequence of transitions from the initial state $[1,1,\ldots,1]$.

**Invariant 3.1** A valid state $s = [s_1,\ldots,s_i,\ldots,s_D]$ in this Markov model must satisfy\(^4\) $0 \leq \delta(s) \leq C - D$, $\gamma(s) \geq 1$, and all parts\(^5\) $s_i \geq 1$ for $1 \leq i \leq D$.

States which have at least $D-1$ free cache blocks can be distinguished from those which have fewer than $D-1$. States of the first type will be known as interior states,

---

\(^4\)Note, however, that not all states which satisfy these three conditions are valid.

\(^5\)Since each part $s_i \geq 1$ for $1 \leq i \leq D$, $\delta(s) \leq C - D$. 
Figure 3.7: The entire Markov chain for the case of three disks and seven cache blocks.
and those of the second type will be known as edge states. In an interior state, full
$D$-way parallelism is achieved by choosing an f-transition, whereas in edge states, the
only transitions that cause an I/O operation are r-transitions which replenish only
the depleted block.

To precisely characterize the set of valid states, the state $s$ must be further qualified
by $\delta(s)$ and $\gamma(s)$. Such a classification is necessary, because the number of parts equal
to one (i.e., $\gamma(s)$) determines the probability of an r-transition or f-transition.

**Definition 3.3** The following classes of states are defined:

**Interior:** $I_k = \{s \mid D - 1 \leq \delta(s) \leq C - D, k = \gamma(s)\}$. This set describes
those interior states with exactly $k$ parts which are 1. A state in $I_k$
has $k$ f-transitions, $(D - k)$ d-transitions, and no r-transitions.

**Edge:** $E_{f,k} = \{s \mid 0 \leq \delta(s) < D - 1, f = \delta(s), k = \gamma(s)\}$. This set
describes those edge states with $f$ free cache blocks and $k$ parts which
are 1. A state in $E_{f,k}$ has $k$ r-transitions, $(D - k)$ d-transitions, and
no f-transitions.

As will be shown later, not all of these classes contain valid states, and the steady-
state probability of an edge state depends on both $f$ and $k$.

**Claim 3.1** If $s$ is reachable from $r$ by a d-transition, then $\delta(s) = \delta(r)+1$,
and either $\gamma(s) = \gamma(r)$ or $\gamma(s) = \gamma(r) + 1$. If $s$ is reachable from $r$ by an
f-transition, then $\delta(s) = \delta(r) - (D - 1) = \delta(r) - D + 1$.

**Lemma 3.1** Any state $s$ with $\gamma(s) \geq 2$ is not reachable from any state
by an f-transition.
Proof. An f-transition from state r to state s increases all but one part of r by 1. Since $\gamma(s) \geq 2$, at least one part of r must be 0. Such a state is not possible.

Lemma 3.2. Any state s with $\delta(s) = 0$ and $\gamma(s) \geq 2$ is unreachable.

Proof. As shown by Lemma 3.1, s could not have been reached by an f-transition since $\gamma(s) \geq 2$. Therefore, s could only have been reached by a d-transition from state r. By Claim 3.1, $\delta(r) = \delta(s) - 1 = -1$, an impossibility.

Lemma 3.3. Any state s with $\gamma(s) \geq \delta(s) + 2$ is not reachable from any other state.

Proof. Since $\gamma(s) \geq 2$, s can not be reached by an f-transition, as shown by Lemma 3.1. Therefore, s could only have been reached by a d-transition from a state t with either $\gamma(t) = \gamma(s) - 1$ or $\gamma(t) = \gamma(s)$. Since $\delta(t) = \delta(s) - 1$, as given by Claim 3.1, two possibilities exist. In the first case,

$$\gamma(t) = \gamma(s) - 1$$

(3.1)

$$\gamma(t) \geq (\delta(s) + 2) - 1$$

(3.2)

Equation 3.2 is derived by substituting $\gamma(s) \geq \delta(s) + 2$ into equation 3.1. Now $\delta(s) = \delta(t) + 1$ is substituted into equation 3.2, resulting in:

$$\gamma(t) \geq (\delta(t) + 1 + 2) - 1 = \delta(t) + 2$$

(3.3)

In the second case,

$$\gamma(t) = \gamma(s)$$

$$\gamma(t) \geq \delta(s) + 2 = (\delta(t) + 1) + 2 = \delta(t) + 3$$

(3.4)
In either case, $\gamma(t) \geq \delta(t) + 2$. The base case holds for $\delta(s) = 0$, as shown by Lemma 3.2. Now assume inductively that this lemma holds for all states $w$ where $\delta(w) < \delta(s)$. As shown by equations 3.3 and 3.4, any state $t$ which reaches $s$ satisfies $\gamma(t) \geq \delta(t) + 2$. Also, $\delta(t) < \delta(s)$, by Claim 3.1. Hence, $t$ satisfies the induction hypothesis, and, as a result, $t$ is unreachable. Hence, $s$ is unreachable. Therefore, the lemma holds for $s$ as well, and the lemma is proven for all $w$, $\delta(w) \leq \delta(s)$. □

**Lemma 3.4** The set of valid states is a subset of $\{I \cup E\}$, in which $I = \{\cup I_k \mid 1 \leq k \leq D\}$ and $E = \{\cup E_{f,k} \mid 0 \leq f < D - 1, 1 \leq k \leq f + 1\}$.

**Proof** By Invariant 3.1, the set of all valid states is a subset of $U = \{s \mid 0 \leq \delta(s) \leq C - D, 1 \leq \gamma(s) \leq D\}$. The set $U$ can be partitioned into subsets, $I$, $E$, and $F$, that are pairwise disjoint and collectively exhaust $U$.

\[
I = \{s \mid D - 1 \leq \delta(s) \leq C - D, 1 \leq \gamma(s) \leq D\} \\
E = \{s \mid 0 \leq \delta(s) < D - 1, 1 \leq \gamma(s) \leq \delta(s) + 1\} \\
F = \{s \mid 0 \leq \delta(s) < D - 1, \delta(s) + 2 \leq \gamma(s) \leq D\}
\]

By Lemma 3.3, all states in $F$ are unreachable and, hence, invalid. Therefore, the set of valid states is a subset of $\{I \cup E\}$. □

**Definition 3.4** The following notation will be used to represent the different types of transitions:

$d_i :$ a $d$-transition by depleting part $s_i$ in $s$.

$f_i :$ an $f$-transition by depleting part $s_i = 1$ in $s$.

A sequence of transitions will be denoted $(x^1 x^2 \ldots x^n)^p$ in which each $x^i$ is either $d_i$ or $f_i$, for $1 \leq i \leq D$, $1 \leq j \leq n$, and the sequence of transitions
\((x^1x^2\ldots x^n)\) is applied \(p\) times beginning with some state \(s\). Within the sequence \((x^1x^2\ldots x^n)\), the transitions \(x^i\) are taken in the order written—from left to right. The final state \(t\) is given by \(t = (x^1x^2\ldots x^n)^p s\). For sequences \(V\) and \(P\), if \(t = Vs\) and \(u = Pt\), \(V\) and \(P\) can be combined such that \(u = VP\)

Let \(I = \{\bigcup I_k \mid 1 \leq k \leq D\}\) and \(E = \{\bigcup E_{f,k} \mid 0 \leq f < D - 1, 1 \leq k \leq f + 1\}\) be sets of states as defined in Lemma 3.4. Now, all states \(c \in \{I \cup E\}\) will be shown to be valid by showing that \(c\) is reachable from \([1,1,\ldots,1]\). This reachability will be shown by describing a sequence of transitions starting from \([1,1,\ldots,1]\) that reaches \(c\) in which all intermediate states belong to \(\{I \cup E\}\). First, a few lemmas regarding this sequence of transitions will be established.

**Lemma 3.5** If \(s \in \{I \cup E\}\), then \(s' = d_is\), for some \(i, 1 \leq i \leq D\). Also satisfies \(s' \in \{I \cup E\}\).

**Proof** In order for the \(d\)-transition to be enabled, \(s_i \geq 2\) must hold initially. Since \(s \in \{I \cup E\}\), \(\gamma(s) \leq \delta(s) + 1\). By Claim 3.1, \(\delta(s') = \delta(s) + 1\) and either \(\gamma(s') = \gamma(s)\) or \(\gamma(s') = \gamma(s) + 1\). When \(\gamma(s') = \gamma(s)\),

\[
\gamma(s') \leq \delta(s) + 1 = \delta(s')
\]

When \(\gamma(s') = \gamma(s) + 1\),

\[
\gamma(s') - 1 \leq \delta(s) + 1 = \delta(s')
\]

Therefore, \(\gamma(s') \leq \delta(s') + 1\). Since \(\delta(s') = \delta(s) + 1\), and \(\delta(s) \geq 0, \delta(s') \geq 1\). Therefore, \(s' \in \{I \cup E\}\). \(\Box\)
Lemma 3.6 For a state \( s \in \{ I \cup E \} \), if \( s_1 = 1 \), and \( \delta(s) \geq (D - 1) + \alpha, \alpha \geq 1 \), then \( t = (f_1d_2, \ldots, d_{j-1}, d_{j+1}, \ldots, d_D)s \) satisfies the following conditions: each intermediate state and \( t \) belong to \( \{ I \cup E \} \), \( \delta(t) \geq (D - 1) + (\alpha - 1) \), and \( t = [1, s_2, \ldots, s_{j-1}, (s_j) + 1, s_{j+1}, \ldots, s_D] \).

Proof Let \( s' = f_1s \). As shown by Claim 3.1, \( \delta(s') = \delta(s) - (D - 1) \). Since \( \delta(s) \geq (D - 1) + \alpha, \delta(s') \geq \alpha \). Since \( \gamma(s') = 1 \) and \( \alpha \geq 1 \), \( \gamma(s') \leq \delta(s') + 1 \), and, hence, \( s' \in \{ I \cup E \} \), as shown by Lemma 3.3.

Let \( s_{i+1} = d_is_i \), where \( s_i^2 = s' \). Since each \( s_i^2 \geq 2, 2 \leq i \leq D \), as shown by Lemma 3.1, each d-transition \( d_i \) is permitted. Lemma 3.5 shows if \( s_i^1 \in \{ I \cup E \} \), each \( s_{i+1} = d_is_i \) belongs to \( \{ I \cup E \} \). Since \( s_i^2 \in \{ I \cup E \} \), as shown above, each intermediate state in the sequence of transitions \( (f_1d_2, \ldots, d_{j-1}, d_{j+1}, \ldots, d_D)s \) also belongs to \( \{ I \cup E \} \).

As defined earlier, \( t = (f_1d_2, \ldots, d_{j-1}, d_{j+1}, \ldots, d_D)s \). Now, \( t \) must be shown to satisfy \( \delta(t) \geq (D - 1) + (\alpha - 1) \). As shown by Claim 3.1, \( \delta(s') = \delta(s) - D + 1 \), and \( \delta(s^{i+1}) = \delta(s^i) + 1 \). Since the sequence contains a total of \( D - 2 \) d-transitions and 1 f-transition,

\[
\delta(t) = \delta(s) - D + 1 + (D - 2) = \delta(s) - 1
\]

\[
\geq (D - 1) + (\alpha - 1)
\]

To show that \( t = [1, s_2, \ldots, s_{j-1}, (s_j) + 1, s_{j+1}, \ldots, s_D] \), note that \( s' = [1, (s_2) + 1, \ldots, (s_{j-1}) + 1, (s_j) + 1, (s_{j+1}) + 1, \ldots, (s_D) + 1] \) and that each element \( i, 2 \leq i \leq D, i \neq j \), is depleted exactly once, resulting in the desired \( t \). \( \Box \)

Lemma 3.7 For a state \( s \in \{ I \cup E \} \), if \( s_1 = 1 \), and \( \delta(s) \geq (D - 1) + n_j \), \( n_j \geq 1 \), then the state \( u = (f_1d_2, \ldots, d_{j-1}, d_{j+1}, \ldots, d_D)^n_s \) satisfies \( \delta(u) \geq \)
\[ D - 1, \ u = [1, s_2, \ldots, s_{j-1}, (s_j) + n_j, s_{j+1}, \ldots, s_D], \] and every intermediate state belongs to \( \{I \cup E\} \).

**Proof**  The lemma is proven by direct application of Lemma 3.6 \( n_j \) times.  \( \Box \)

**Lemma 3.8** Every state \( c \in \{I \cup E\} \) is reachable from \([1,1,\ldots,1]\) and can reach \([1,1,\ldots,1]\).

**Proof** In the following proof, the components of \( c \) will be assumed to have been re-arranged in a non-decreasing order. This reordering is used purely for notational convenience; the reordering does not restrict \( c \) in any manner. That is, \( c = [1, \ldots, 1, c_{k+1}, c_{k+2}, \ldots, c_D] \) such that \( c_i \leq c_{i+1} \) for \( 1 \leq i \leq D \). As a result of this reordering, \( c_i = 1 \) for \( 1 \leq i \leq k, \ k = \gamma(c), \) and \( c_i \geq 2 \) for \( k + 1 \leq i \leq D \). Note that since \( c \in \{I \cup E\} \), \( k = \gamma(c) \leq \delta(c) + 1 \). Rearranging this expression,

\[
\delta(c) \geq k - 1 \tag{3.5}
\]

A sequence of transitions required to reach \( c \) from \( s \) is given by the following pseudo-code.

\[
s = [1,1,\ldots,1] \\
\text{For } j = k + 1, \ldots, D \\
\quad t = ((f_1d_2d_3 \ldots d_{j-1}d_{j+1} \ldots d_D)^{c_j-2})s \\
\quad s = t \\
\quad u = (f_1d_2d_3 \ldots d_k)s
\]

In order to reach \( c \) from \([1,1,\ldots,1]\), induct on \( j \). For a given iteration on \( j \), \( t \in \{I \cup E\} \) is reached from \( s \) by the sequence of transitions

\[
t = ((f_1d_2d_3 \ldots d_{j-1}d_{j+1} \ldots d_D)^{c_j-2})s
\]
Each intermediate state in this sequence must be shown to belong to \( \{I \cup E\} \). Assume inductively that at the start of iteration \( j \), \( s = [1, \ldots, 1, c_{k+1}-1, \ldots, c_{j-1}-1, 1, \ldots, 1] \), \( s \in \{I \cup E\} \), \( \delta(s) \geq D - 1 \). After the sequence of transitions, \( t \) will be shown to be \( t = [1, \ldots, 1, c_{k+1} - 1, \ldots, c_{j-1} - 1, (c_{j}) - 1, 1, \ldots, 1] \), \( t \in \{I \cup E\} \), and \( \delta(t) \geq D - 1 \).

First, \( \delta(s) \geq (D - 1) + (c_{j} - 2) \) will be shown. In order to compute \( \delta(s) \), the difference between \( s_{i} \) and \( c_{i} \), \( 1 \leq i \leq D \) will be determined. For \( j \leq i \leq D \), since \( s_{i} = 1 \) and \( c_{i} \geq 2 \), \( s_{i} \leq (c_{j} - 1) \).

\[
\sum_{i=j}^{D} (c_{i} - s_{i}) \geq (c_{j} - 1) + \sum_{i=j}^{D} 1
\]
\[
\geq (c_{j} - 1) + (D - j)
\]

Note that \( k = \gamma(c) \leq \delta(c) + 1 \). By rearranging this expression, \( \delta(c) \geq k - 1 \). Since \( s_{i} = c_{i} - 1 \) for \( k + 1 \leq i \leq j - 1 \),

\[
\delta(s) = \delta(c) + (j - k - 1) + \sum_{i=j}^{D} (c_{i} - 1)
\]
\[
\geq \delta(c) + (j - k - 1) + (c_{j} - 1) + (D - j)
\]
\[
\geq \delta(c) + (D - 1) + (c_{j} - k - 1)
\]
\[
\geq (D - 1) + (c_{j} - 2), \text{ since } k \geq 1
\]

Since \( \delta(s) \geq (D - 1) + (c_{j} - 2) \), Lemma 3.7 shows that \( t = [1, s_{2}, \ldots, s_{j-1}, s_{j} + (c_{j} - 2), s_{j+1}, \ldots, s_{D}] \) and \( \delta(t) \geq D - 1 \).

After the sequence of transitions, \( s_{j} + (c_{j} - 2) = c_{j} - 1 \), since \( s_{j} = 1 \) initially. Since \( \delta(t) \geq D - 1 \), the induction step is shown to hold. The base case— \( s = [1, 1, \ldots, 1], j = k + 1 — satisfies^6 \delta(s) \geq (D - 1) \) and \( \delta(s) \in \{I \cup E\} \).

Let \( u = (f_{1}d_{2}, d_{3}, \ldots, d_{k})s \) and \( s' = f_{1}s \), in which \( s = [1, \ldots, 1, (c_{k+1})-1, \ldots, (c_{D})-1] \). Since \( \delta(s) \geq D - 1 \), \( \delta(s') \geq 0 \), \( \gamma(s') = 1 \), and \( s' = [1, 2, \ldots, 2, c_{k+1}, \ldots, c_{D}] \). Therefore,

---

^6Note that \( C \geq 2D - 1 \) as defined earlier.
\( s' \in \{I \cup E\} \). Since \( s'_i \geq 2, 2 \leq i \leq k \), each d-transition \( d_2 \ldots d_k \) is enabled. By Lemma 3.5, every state reached after any such d-transition belongs to \( \{I \cup E\} \). After the sequence \((d_2d_3 \ldots d_{j-1}d_{j+1} \ldots d_D)\) is applied to \( s' \), \( u_i = s'_i - 1 = c_i = 1, 2 \leq i \leq k \).

As a result, \( u = c = [1, \ldots, 1, c_{k+1}, \ldots, c_D] \).

In order to reach \([1, 1, \ldots, 1]\) from \( c = [1, \ldots, 1, c_{k+1}, c_{k+2}, \ldots, c_D] \), the following sequence of transitions is sufficient:

\[
\begin{align*}
s &= c \\
&\text{For } i = k + 1, \ldots, D \\
&t = (d_i)^{c_i - 1}s \\
&s = t
\end{align*}
\]

Each part \( s_i \) in \( s \) is depleted in turn, until the state \([1, 1, \ldots, 1]\) is reached. Since \( s_i = c_i \), the sequence of \((c_i - 1)\) d-transitions is permitted. Each d-transition results in a state which belongs to \( \{I \cup E\} \), as shown by Lemma 3.5. After parts \( s_{k+1}, \ldots, s_D \) have been depleted, the state \([1, 1, \ldots, 1]\) is reached. \( \square \)

**Theorem 3.1** The set of all valid states is given by exactly \( \{I_k, 1 \leq k \leq D\} \cup \{E_f, k, 0 \leq f < D - 1, 1 \leq k \leq f + 1\} \). Furthermore, every valid state can reach \([1, 1, \ldots, 1]\).

**Proof** This theorem is proven directly from Lemmas 3.4 and 3.8. \( \square \)

### 3.3 State Probabilities

In order to determine the probability of caching all \( D \) blocks on any given I/O operation, the steady-state probabilities for each state in the Markov model must be determined. Since the absolute probability of each state depends on the sum of all
state probabilities, the relative probability of each state can be determined much more easily. The absolute probability will simply be this relative probability scaled by a constant, $A$. A state $s$ with $\delta(s) = 0, \gamma(s) = 1$ will be given a relative probability of 1, i.e., an absolute probability of $A$; as will be shown later, all such states have the same probability. All other relative probabilities will be expressed relative to this probability.

**Theorem 3.2**  The *ergodic theorem* [15] for a discrete-time Markov chain states that if a Markov chain is irreducible, recurrent nonnull, and aperiodic (i.e., it is ergodic), there exists a unique limiting distribution for the probability of being in a state $k$ denoted as $\pi_k$, independent of the initial state. These probabilities are called the *steady-state* or equilibrium probabilities.

**Lemma 3.9**  Each valid state $s$ has a unique steady-state probability.

**Proof**  Note that Theorem 3.1 has shown that the set of valid states is given by exactly $\{I_k, 1 \leq k \leq D\} \cup \{E_{f,k}, 0 \leq f < D - 1, 1 \leq k \leq f + 1\}$.

The Markov chain is irreducible since all states are reachable from all other states. Any valid state $s$ can reach any state $t$, since $s$ can reach $[1, 1, \ldots, 1]$ and then reach $t$ from $[1, 1, \ldots, 1]$, as shown by Lemma 3.8.

The Markov chain is recurrent nonnull since the mean-time to return to any state is finite. Each state is reachable from any arbitrary state, and each state transition has $1/D$ probability. Each state is recurrent since a given sequence of transitions always exists in order to return to that state. The model clearly contains a finite number of states for any $C$ and $D$. Hence, the mean-time to return to any state will always be a finite value.
In order for the Markov chain to be aperiodic, for each state there must exist some number \( k \) such that returning to the state can be accomplished in \( k, k+1, k+2, \ldots, \infty \) transitions. As shown by Lemma 3.8, an arbitrary valid state \( s \) can reach a state \( c \in E_{f,k} \) in \( \alpha \) transitions, for some \( \alpha \). Returning to \( s \) from \( c \) can also be accomplished in \( \beta \) transitions, for some \( \beta \).

Since \( c \in E_{f,k} \), \( c \) has an \( r \)-transition. Taking this \( r \)-transition from \( c \) will not lead to a new state. Therefore, taking \( r \geq 0 \) such \( r \)-transitions from \( c \) will not change the state. As a result, returning to \( s \) from \( c \) can be accomplished in \( \beta + r, r \geq 0 \), transitions.

Hence, returning to \( s \) after leaving \( s \) can be accomplished in \( \alpha + \beta, \alpha + \beta + 1, \alpha + \beta + 2, \ldots, \infty \) transitions. Therefore, each state \( s \) is aperiodic, and, hence, the entire Markov chain is aperiodic.

The Markov chain satisfies all requirements of Theorem 3.2, and, hence, each state has a unique steady-state probability.

Table 3.1 lists all state classifications and their associated relative probabilities. To verify that these probabilities are correct, the probabilities of all incoming transitions to a state will be summed. This sum should equal the probability for the state. In particular, the left-hand side of the equation \( \pi P = \pi \) will be computed and shown.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Relative Steady-State Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_k ) ( 1 \leq k \leq D )</td>
<td>Interior state ( s ) with ( k = \gamma(s) ) parts equal to 1.</td>
<td>( (D - 1) )</td>
</tr>
<tr>
<td>( E_{f,k} ) ( 0 \leq f &lt; D - 1 ) ( 1 \leq k \leq f + 1 )</td>
<td>An edge state ( s ) with ( f = \delta(s) ) free cache blocks and ( k = \gamma(s) ) parts which are equal to 1.</td>
<td>( \frac{(f + 1)!}{(f - k + 1)!(D - 2)!} )</td>
</tr>
</tbody>
</table>
to equal the right-hand side. The vector $\pi$ contains the state probabilities, and the matrix $P$ is the single-step transition probability matrix [15]. The values of $\pi$ will be those probabilities given in Table 3.1, and $P$ will be given by the transition rules defined earlier.

Figures 3.8 through 3.14 show the transitions associated with different state classes. In each of these figures, each transition has $\frac{1}{D}$ probability of being taken, except for those transitions labeled as having $n$ occurrences which have $\frac{n}{D}$ probability.

Lemma 3.10 All interior states have relative probability $(D - 1)$.

Proof Figure 3.8 shows the incoming transitions to an interior state state $s$ with $\gamma(s) = 1$. State $s$ can be reached by $D - 1$ d-transitions or an f-transition. The f-transition must be from an interior state, and all interior states have relative steady-state probability $D - 1$. The d-transition must be either from another interior state or from an edge state $u$ with $\delta(u) = D - 2$ and $\gamma(u) = 1$. The steady-state probability of this edge state is also $D - 1$. Summing the probabilities on the input edges is straightforward:

$$\frac{1}{D}(D)(D - 1) = (D - 1)$$

Figures 3.9 and 3.10 show an interior state $s$ with $2 \leq \gamma(s) \leq D$. Such a state cannot be reached by any f-transitions, as shown by Lemma 3.1. All incoming transitions are d-transitions, each having probability $\frac{1}{D}$. These d-transitions originate either from an interior state or from an edge state $u$, where $\delta(u) = D - 2$, and $\gamma(u) = \gamma(s)$ or $\gamma(u) = \gamma(s) + 1$. All of these states have relative steady-state probability $(D - 1)$. As a result, $s$ also has relative steady-state probability $(D - 1)$.
Figure 3.8: $I_1$: Interior state with exactly one part equal to 1.

Figure 3.9: $I_i$: Interior state with $i$ parts equal to 1.
Figure 3.10: \( I_D \): Central interior state with all \( D \) parts equal to 1.

Figure 3.11: An edge state with no free cache blocks. Exactly one part can be equal to 1.
Lemma 3.11 A state $s \in E_{f,1}$ with $f = \delta(s)$ and $\gamma(s) = 1$ has relative steady-state probability $(f+1)$.

Proof Figure 3.11 shows the incoming transitions to a state $s \in E_{0,1}$. The sum of the probabilities of the incoming transitions is:

$$\frac{1}{D}(f+1) + \frac{1}{D}(D-1) = 1$$

which satisfies the lemma, since $f = 0$.

For $f \geq 1$, Figure 3.12 shows the relevant incoming transitions. Any d-transition reaching $s$ must be from an edge state $u$, with $\delta(u) = f-1$ and $\gamma(s) = 1$. Summing the probabilities of the incoming transitions results in:

$$\frac{1}{D}(D-1)f + \frac{1}{D}(D-1) + \frac{1}{D}(f+1) = \frac{1}{D}(Df + D) = (f+1)$$

Lemma 3.12 A state $s \in E_{f,k}$ with the restrictions that $f = \delta(s)$, $f \geq 1$, and $k = \gamma(s)$, $2 \leq k \leq f$, has relative steady-state probability

$$\frac{(f+1)!}{(f-k+1)!(D-2)!}$$

Proof Figure 3.13 shows the case of an edge state with $2 \leq k \leq f$. The state $s$, with $f = \delta(s) \geq 2$ and $k = \gamma(s)$, $2 \leq k \leq f$, has a total of $D$ incoming d-transitions; $k$ of these incoming d-transitions are from states in $E_{f-1,k}$, and the other $D-k$ incoming d-transitions are from states in $E_{f-1,k-1}$. Since $\gamma(s) \geq 2$, $s$ can not be reached from any state by an f-transition, as given by Lemma 3.3. Since $s$ has $k = \gamma(s)$ parts equal to 1, $s$ has $k$ r-transitions. If $B_{f,k}$ is defined to be the probability of a state $s$ with $f = \delta(s)$ and $k = \gamma(s)$, then the relative probability of $s$ is given by the following
Figure 3.12: A state in $E_{f,1}$ with $f \geq 1$ free cache blocks and 1 part which is 1.

Figure 3.13: A state in $E_{f,k}$ with $f \geq 1$ free cache blocks and $k \geq 2$ parts which are 1.
equation:

\[ B_{f,k} = \frac{k}{D} (B_{f,k}) + \frac{k}{D} [B_{f-1,k-1}] + \frac{D-k}{D} [B_{f-1,k}] \]  

(3.6)

By rearranging equation 3.6,

\[ B_{f,k} = \frac{k[B_{f-1,k-1}] + (D-k)[B_{f-1,k}]}{D-k} \]  

(3.7)

To prove the probability given by the lemma, the probability will be substituted into equation 3.7.

\[ B_{f,k} = \frac{(f+1)!(D-k-1)!}{(f-k+1)!(D-2)!} \]  

(3.8)

\[ B_{f-1,k-1} = \frac{f!(D-k)!}{(f-k+1)!(D-2)!} \]  

(3.9)

\[ B_{f-1,k} = \frac{f!(D-k-1)!}{(f-k)!(D-2)!} \]  

(3.10)

Now the expressions given by equations 3.9 and 3.10 will be substituted into equation 3.7. The property \( x! = x(x-1)! \) will be used in the following derivations.

\[ B_{f,k} = \frac{1}{D-k} \left[ k \frac{f!(D-k)!}{(f-k+1)!(D-2)!} + (D-k) \frac{f!(D-k-1)!}{(f-k)!(D-2)!} \right] \]

\[ = \frac{1}{D-k} \left[ k \frac{f!(D-k)!}{(f-k+1)!(D-2)!} + \frac{f!(D-k)!}{(f-k)!(D-2)!} \right] \]

\[ = \frac{1}{D-k} \left[ \frac{(D-k)!}{(D-2)!} \left[ \frac{k f!}{(f-k+1)!} + \frac{f!}{(f-k)!} \right] \right] \]

\[ = \frac{(D-k-1)!}{(D-2)!} \left[ \frac{k f!}{(f-k+1)!} + \frac{f!(f-k+1)}{(f-k+1)!} \right] \]

\[ = \frac{(D-k-1)!}{(D-2)!} \left[ \frac{k f! + (f+1)f! - k f!}{(f-k+1)!} \right] \]

\[ = \frac{(f+1)!(D-k-1)!}{(f-k+1)!(D-2)!} \]

Since equation 3.7 is satisfied, the proof is complete. \( \square \)
Figure 3.14: A state in $E_{f,f+1}$ with $f \geq 1$ free cache blocks and $f + 1$ parts which are 1.

**Lemma 3.13** A state $s \in E_{f,f+1}$ with $f = \delta(s)$, $\gamma(s) = f + 1$ has relative steady-state probability

$$\frac{(f + 1)! (D - f - 2)!}{(D - 2)!}$$

**Proof** Although the probability given by this lemma satisfies the probability given by Lemma 3.12 for $k = f + 1$, the derivation differs, since the incoming state transitions are quite different. Figure 3.14 shows the edge state $s$ with $f = \delta(s)$ and $\gamma(s) = \delta(s) + 1 = f + 1$. Such a state has $f + 1$ incoming d-transitions from states in $E_{f-1,f}$. State $s$ can not be reached by a d-transition from any state $u$ with $\gamma(u) = \gamma(s)$, since $u$ would have $\gamma(u) = \gamma(s) = f + 1$ and $\delta(u) = \delta(s) - 1 = f - 1$. Since $\gamma(u) = \delta(u) + 2$, $u$ is invalid, as shown by Lemma 3.3. Since $s$ has $\gamma(s) = f + 1$ parts equal to 1, $s$ also has $f + 1$ r-transitions. If $B_{f,f+1}$ denotes the probability of the state $s$ with $f = \delta(s)$ and $\gamma(s) = \delta(s) + 1 = f + 1$, then the following equation must hold:

$$B_{f,f+1} = \frac{f + 1}{D}[B_{f-1,f}] + \frac{f + 1}{D}[B_{f,f+1}]$$
\[ D \cdot B_{f,f+1} = (f+1)[B_{f-1,f}] + (f+1)B_{f,f+1} \]
\[ B_{f,f+1} = \frac{(f+1)[B_{f-1,f}]}{D - f - 1} \]  \hspace{2cm} (3.11)

To prove the probability given by the lemma, the probability will be substituted into equation 3.11.
\[ B_{f,f+1} = \frac{(f+1)!((D-f-2)!}{(D-2)!} \]
\[ B_{f-1,f} = \frac{f!((D-f-1)!}{(D-2)!} \]  \hspace{2cm} (3.12)

Now \( B_{f-1,f} \) given by equation 3.12 will be substituted into equation 3.11.
\[ B_{f,f+1} = \frac{1}{D - f - 1} \left[ \frac{(f+1)!((D-f-1)!}{(D-2)!} \right] \]
\[ = \frac{1}{D - f - 1} \left[ \frac{(f+1)!((D-f-1)!}{(D-2)!} \right] \]
\[ = \frac{(f+1)!((D-f-2)!}{(D-2)!} \]

Since equation 3.11 is satisfied, the proof is complete. \( \square \)

**Theorem 3.3** The relative steady-state probabilities of all valid states in the Markov chain are given by Table 3.1.

**Proof** For \( k = f + 1 \), the probability given by Lemma 3.13 reduces to that given by Lemma 3.12. For \( k = 1 \), the probability given by Lemma 3.11 reduces to that given by Lemma 3.12. Three separate proofs are required, however, since the state transitions are rather different. Lemma 3.10 proves the probability for the interior state. \( \square \)
3.4 Counting States

3.4.1 Useful Summations

The following summations [3] [8] [9] will be useful in simplifying some of the expressions encountered in the later sections.

\[
1 + m + \frac{m(m+1)}{2!} + \cdots + \frac{m(m+1) \cdots (m+r-1)}{r!} = \frac{(m+r)!}{m!r!} = \binom{m+r}{r} = \binom{m+r}{m}
\]  \hspace{1cm} (3.13)

where \( r \) and \( m \) are positive integers

Gaussian polynomials [3] are defined for the complex variable \( q \) as follows:

\[
(q)_n = (1-q^n)(1-q^{n-1}) \cdots (1-q)
\]

The notation \( \left[ \begin{array}{c} n \\ m \end{array} \right] \) is defined as:

\[
\left[ \begin{array}{c} n \\ m \end{array} \right] = \begin{cases} (q)_n(q)_{m-1}(q)_{n-m} & \text{if } 0 \leq m \leq n \\ 0 & \text{otherwise} \end{cases}
\]

\[
\sum_{k=0}^{h} \left[ \begin{array}{c} n \\ k \end{array} \right] \left[ \begin{array}{c} m \\ h-k \end{array} \right] q^{(n-k)(h-k)} = \left[ \begin{array}{c} m+n \\ h \end{array} \right]
\]

\[
\lim_{q \rightarrow 1} \left[ \begin{array}{c} n \\ m \end{array} \right] = \binom{n}{m} = \frac{n!}{m!(n-m)!}
\]

As \( q \rightarrow 1 \),

\[
\sum_{k=0}^{h} \binom{n}{k} \binom{m}{h-k} = \binom{m+n}{h}
\]  \hspace{1cm} (3.14)

A similar expression, known to Chu Shih-Chieh in 1303 [8], will be useful:

\[
\sum_{k=-m}^{n} \binom{r}{m+k} \binom{s}{n-k} = \binom{r+s}{m+n}
\]  \hspace{1cm} (3.15)
3.4.2 Theory of Partitions

In counting the different classes of states in the Markov model, partition theory will be useful.

**Definition 3.5** [3] A *partition* of a positive integer \( n \) is defined to be a finite nonincreasing sequence of positive integers \( \lambda_1, \lambda_2, \ldots, \lambda_r \) such that \( \sum_{i=1}^{r} \lambda_i = n \). The \( \lambda_i \) are called parts of the partition.

**Definition 3.6** [3] A *composition* is a partition in which the order of the summands is considered. The compositions of \( n \) with exactly \( m \) parts will be denoted \( c(m, n) \). The number of compositions of \( n \) into exactly \( m \) parts, each part \( \geq k \), will be denoted by \( c_k(m, n) \).

In the Markov model, a state with \( S \) used cache blocks corresponds to a composition of \( S \) with \( D \) parts, each part \( \geq 1 \). The order of the parts is important in the Markov model since different orderings represent different cache states. As shown in Andrews, [3],

\[
c(m, n) = \binom{n-1}{m-1} = \frac{(n-1)!}{(m-1)!(n-m)!}
\]

\[
c_k(m, n) = \binom{n-(k-1)m-1}{m-1}
\]  \( \quad (3.16) \)

For \( k = 2 \), \( c_2(m, n) \) is the number of compositions in which no part equals 1.

\[
c_2(m, n) = \binom{n-m-1}{m-1}
\]

**Definition 3.7** Let \( \hat{n}(D, S) \) be the number of states in which the \( D \) parts sum to \( S \) and no part equals 1. Then,

\[
\hat{n}(D, S) = c_2(D, S) = \binom{S-D-1}{D-1}
\]
3.5 Average Parallelism on Each I/O Operation

Since an I/O operation occurs on an f-transition or an r-transition, the probability of these transitions will determine the average I/O parallelism.

**Definition 3.8** Let $\varphi_f$ be the probability that an f-transition is taken, and let $\varphi_r$ be the probability that an r-transition is taken. Define

$$\varphi = \frac{\varphi_f}{\varphi_f + \varphi_r}$$

Therefore, $\varphi$ represents the probability of reading from all $D$ disks on each I/O operation. Then, the average parallelism of an I/O operation is given by $D\varphi + (1 - \varphi)$.

In order to determine $\varphi$, the function $\tilde{n}(D, S, k)$ is introduced.

**Definition 3.9** Let $\tilde{n}(D, S, k)$ be the number of states in which the $D$ parts sum to $S$, and exactly $k$ parts are equal to 1.

**Lemma 3.14**

$$\tilde{n}(D, S, k) = \binom{D}{k} \binom{S - D - 1}{D - k - 1}$$

**Proof**

$$\tilde{n}(D, S, k) = \binom{D}{k} \hat{n}(D - k, S - k)$$

$$= \binom{D}{k} \binom{(S - k) - (D - k) - 1}{(D - k) - 1}$$

$$= \binom{D}{k} \binom{S - D - 1}{D - k - 1}$$

Equation 3.18 sums the various ways of constructing a state with exactly $k$ parts equal to 1. The other $D - k$ parts must not equal to 1, and they must sum to exactly $S - k$. $\hat{n}(D - k, S - k)$ gives the number of such possible combinations. Finally, the $k$ parts which are equal to 1 can be chosen in $\binom{D}{k}$ ways.
Lemma 3.15  The relative probability of an f-transition being taken is given by:

$$\varphi_s = (D - 1) \binom{C - D}{D - 1}$$

Proof  Note that an f-transition can be taken only from an interior state. Each interior state has relative steady-state probability $D^{-1}$, as given by Theorem 3.3. An interior state $s$ with $k$ parts equal to 1 has $k$ such f-transitions. Since each transition has probability $\frac{1}{D}$ of being taken, the probability of an f-transition being taken from $s$ is $k \frac{1}{D} (D - 1)$. This probability must now be summed for all interior states.

The number of parts, $k$, which equal 1 will depend on $S$, the number of used cache blocks. All interior states satisfy $D \leq S \leq C - D + 1$, but those with $D + 1 \leq S \leq 2D - 2$ can not have all possible values of $k$. For this range of $S$, the minimum value of $k$ can be achieved by having as many non-one parts as possible. Since each $s_i \geq 1$ for $1 \leq i \leq D$, only $S - D$ parts can be greater than 1. Therefore, only $D - (S - D) = 2D - S$ parts can be equal to 1. For a state $s$ with $D + 1 \leq S \leq 2D - 2$, $k$ must be in the range $2D - S \leq k \leq D - 1$, since the number of used cache blocks is not sufficient to allow $1 \leq k \leq 2D - S - 1$; otherwise, the sum of the parts cannot be as large as $S$.

For $k = D$, only one state, the central interior state, is possible, and all transitions from this state are f-transitions. A state $s$ with $2D - 1 \leq S \leq C - D + 1$ can have $1 \leq k \leq D - 1$ parts equal to 1.

Over the range of $S$ and $k$ described above, $\tilde{n}(D, S, k)$ gives the number of interior states with $k$ f-transitions. The probability of an f-transition for a given value of $k$ and $S$ is given by:

$$\frac{k}{D^2} \frac{(D - 1)}{D} \tilde{n}(D, S, k) = (D - 1) \frac{k}{D} \binom{D}{k} (S - D - 1) \binom{D - k - 1}{D - k - 1}$$
\begin{equation}
\varphi_f = (D - 1) \sum_{S=D+1}^{2D-2} \left[ \sum_{k=2D-S}^{D-1} \binom{D-1}{k-1} \binom{S-D-1}{D-k-1} \right] + (D - 1) \sum_{S=2D-1}^{C-D+1} \left[ \sum_{k=1}^{D-1} \binom{D-1}{k-1} \binom{S-D-1}{D-k-1} \right] + (D - 1)
\end{equation}

To compute \( \varphi_f \), the probability from equation 3.19 is summed over \( k \) and \( S \) for interior states.

\begin{equation}
\varphi_f = (D - 1) \sum_{S=D+1}^{2D-2} \sum_{k=2D-S}^{D-1} \binom{D-1}{k-1} \binom{S-D-1}{D-k-1} + (D - 1) \sum_{S=2D-1}^{C-D+1} \sum_{k=1}^{D-1} \binom{D-1}{k-1} \binom{S-D-1}{D-k-1} + (D - 1)
\end{equation}

In order to bring the summations in equation 3.20 into the closed form given by equation 3.14, the substitution \( l = k - 1 \) will be made, resulting in:

\begin{equation}
\varphi_f = (D - 1) \sum_{S=D+1}^{2D-2} \sum_{l=2D-S-1}^{D-2} \binom{D-1}{l} \binom{S-D-1}{D-2-l} + (D - 1) \sum_{S=2D-1}^{C-D+1} \sum_{l=0}^{D-2} \binom{D-1}{l} \binom{S-D-1}{D-2-l} + (D - 1)
\end{equation}

The innermost sum can be simplified by the Gaussian summation, given by equation 3.14, with \( n = D - 1, k = l, m = S - D - 1, h = D - 2 \).

\begin{equation}
\varphi_f = (D - 1) \sum_{S=D+1}^{2D-2} \left[ \binom{S-2}{D-2} - \sum_{l=0}^{2D-S-2} \binom{D-1}{l} \binom{S-D-1}{D-2-l} \right] + (D - 1) \sum_{S=2D-1}^{C-D+1} \binom{S-2}{D-2} + (D - 1)
\end{equation}

\begin{equation}
= (D - 1) \sum_{S=D+1}^{2D-2} \binom{S-2}{D-2} + (D - 1) \sum_{S=2D-1}^{C-D+1} \binom{S-2}{D-2} - (D - 1) \sum_{S=D+1}^{2D-2} \sum_{l=0}^{2D-S-2} \binom{D-1}{l} \binom{S-D-1}{D-2-l} + (D - 1)
\end{equation}

By noting the ranges of summation in equation 3.22, \( S - D \leq D - 2 - l \leq D - 2 \).

Since \( \binom{a}{b} = 0 \) for \( a < b \) and \( (S - D - 1) < (D - 2 - l) \), the last summation over \( S \) and \( l \) in equation 3.22 is equal to zero. Therefore, equation 3.22 reduces to:

\begin{equation}
= (D - 1) \sum_{S=D+1}^{2D-2} \binom{S-2}{D-2} + (D - 1) \sum_{S=2D-1}^{C-D+1} \binom{S-2}{D-2} + (D - 1)
\end{equation}
The first two summations in equation 3.23 can be combined, since the range of $S$ is contiguous.

\[
(D - 1) \sum_{S=D+1}^{C-D+1} \frac{(S-2)}{D-2} + (D - 1)
\]  

(3.24)

The summation in equation 3.24 can be expanded in order to achieve the summable form given by equation 3.13.

\[
= (D - 1) + (D - 1) \left[ \frac{(D - 1)!}{(D - 2)!1!} + \frac{(D)!}{(D - 2)!2!} + \cdots + \frac{(C - D - 1)!}{(D - 2)!(C - 2D + 1)!} \right]
\]

\[
= (D - 1) + (D - 1) \left[ \frac{D - 1}{2!} + \frac{(D - 1)(D)}{3!} + \cdots + \frac{(D - 1)(D)(D+1)}{(C - 2D + 1)!} \right] + \cdots
\]

(3.25)

Equation 3.25 can be simplified by using the summation given in equation 3.13, with $m = (D - 1), r = (C - 2D + 1)$. Following the substitution, the expression simplifies to $(D - 1)\binom{C-D}{D-1}$.

\[\square\]

**Lemma 3.16** The relative probability of an r-transition being taken is given by:

\[
\varphi_r = (D-1)\binom{C-D}{D-1} \left[ (1 - D) + (C - D + 1) \left[ H(C - D) - H(C - 2D + 1) \right] \right]
\]

in which $H(n) = \sum_{i=1}^{n} \frac{1}{i}$.

**Proof** The probability of an edge state $s$ with $f = \delta(s)$, and $k = \gamma(s)$ is $\frac{(f+1)!(D-k-1)!}{(f-k+1)!(D-2)!}$, as shown by Theorem 3.3. Since each edge state has $k$ r-transitions and each transition has probability $\frac{1}{D}$, the probability of an r-transition being taken is $\frac{k}{D} \left( \frac{(f+1)!(D-k-1)!}{(f-k+1)!(D-2)!} \right)$.

Each edge state $s$ will satisfy $f = \delta(s), 0 \leq f \leq D - 2$, and $k = \gamma(s), 1 \leq k \leq f + 1$, as shown by Lemma 3.3. Since $S = C - f$, $\tilde{n}(D, C - f, k)$ gives the number of edge states with $k$ r-transitions. To compute $\varphi_r$, the probability of an r-transition is
summed over all edge states.

$$
\varphi_r = \sum_{f=0}^{D-2} \sum_{k=1}^{f+1} \left[ \frac{k}{D} \frac{(f+1)!(D-k-1)!}{(f-k+1)!(D-2)!} \binom{D}{k} \binom{C-f-D-1}{D-k-1} \right] 
$$

(3.26)

Equation 3.26 can be simplified by rearranging and combining terms, resulting in:

$$
\varphi_r = \sum_{f=0}^{D-2} \sum_{k=1}^{f+1} \left[ \frac{f+1}(D-k) \binom{f}{k-1} \binom{C-f-D-1}{D-k-1} \right] 
$$

(3.27)

The \((D-k)\) term in equation 3.27 can now be combined with \(\binom{C-f-D-1}{D-k-1}\) by noting that \(\frac{1}{a+1}(a+1)! = a!\).

$$
\varphi_r = (D-1) \sum_{f=0}^{D-2} \left[ \frac{f+1}{(D-k)} \sum_{k=1}^{f+1} \left( \frac{f}{k-1} \frac{1}{C-f-D} \binom{C-f-D}{D-k-1} \right) \right] 
$$

$$
= (D-1) \sum_{f=0}^{D-2} \left[ \frac{f+1}{C-f-D} \sum_{k=1}^{f+1} \left( \binom{f}{k-1} \binom{C-f-D}{D-k} \right) \right] 
$$

(3.28)

The inner sum in equation 3.28 can be simplified by using Shih-Chieh's summation, given by equation 3.15, with \(m = -1, r = f, s = C-f-D, n = D\). The resulting equation is given by:

$$
\varphi_r = (D-1) \sum_{f=0}^{D-2} \left[ \frac{f+1}{C-f-D} \left( \binom{C-D}{D-1} - \sum_{k=f+2}^{D} \frac{f}{k-1} \binom{C-f-D}{D-k} \right) \right] 
$$

By noting that \(f+1 \leq k-1 \leq D-1\) and, as a result, \(k-1 > f\), the last summation equals zero, since \(\binom{f}{k-1} = 0\). The resulting expression is:

$$
\varphi_r = (D-1) \binom{C-D}{D-1} \sum_{f=0}^{D-2} \frac{f+1}{C-f-D} 
$$

(3.29)

In order to simplify equation 3.29, the summation will be expanded; for clarity, let \(x = C-D\).

$$
\sum_{f=0}^{D-2} \frac{f+1}{x-f} = \frac{1}{x} + \frac{2}{x-1} + \frac{3}{x-2} + \cdots + \frac{D-1}{x-(D-2)} 
$$

(3.30)
Now, each term in equation 3.30 will be rewritten as the sum of $-1$ and $1 + a$, given that the original term has value $a$.

\[
\sum_{f=0}^{D-2} \frac{f+1}{x-f} = \left( -\frac{x}{x} + \frac{x+1}{x} \right) + \left( -\frac{x-1}{x-1} + \frac{(x-1)+2}{x-1} \right) + \left( -\frac{(x-2)}{x-2} + \frac{(x-2)+3}{x-2} \right) + \cdots
\]

\[
+ \left( -\frac{(x-(D-2))}{x-(D-2)} + \frac{x-(D-2)+(D-1)}{x-(D-2)} \right)
\]

\[
= \left( -1 + \frac{x+1}{x} \right) + \left( -1 + \frac{x+1}{x-1} \right) + \left( -1 + \frac{x+1}{x-2} \right) + \cdots + \left( -1 + \frac{x+1}{x-(D-2)} \right)
\]

\[
= -(D-1) + (x+1) \left( \frac{1}{x} + \frac{1}{x-1} + \frac{1}{x-2} + \cdots + \frac{1}{x-(D-2)} \right)
\]  

(3.31)

The goal is to express the terms from equation 3.31 in terms of $H(n) = \sum_{i=1}^{n} (1/i)$.

First, $x = C - D$ will be substituted into equation 3.31.

\[
\sum_{f=0}^{D-2} \frac{f+1}{C-D-f} = -(D-1) + (C-D+1) \left( \frac{1}{C-D} + \frac{1}{C-D-1} + \frac{1}{C-D-2} + \cdots + \frac{1}{C-2D+2} \right)
\]

\[
= (1-D) + (C-D+1) [H(C-D) - H(C-2D+1)]
\]  

(3.32)

By substituting equation 3.32 into equation 3.29, the resulting expression gives the probability shown by the lemma.

\[\Box\]

**Theorem 3.4**

\[
\varphi = \frac{1}{2-D+(C-D+1)[H(C-D)-H(C-2D+1)]}
\]

in which $H(n) = \sum_{i=1}^{n} (1/i)$. 
Proof    Lemmas 3.15 and 3.16 show that \( \varphi_i = (D - 1)\left(\frac{C-D}{D-1}\right) \) and \( \varphi_r = (D - 1)\left(\frac{C-D}{D-1}\right) \left[(1 - D) + (C - D + 1) (H(C - D) - H(C - 2D + 1))\right] \). As given by Definition 3.8, \( \varphi = \frac{\varphi_i}{\varphi_i + \varphi_r} \). By substituting the expressions derived for \( \varphi_i \) and \( \varphi_r \), the theorem is proven.

\[ \square \]

3.6 Simulation Results

Figure 3.15 shows the behavior\(^7\) of \( \varphi \) for small values of \( D \). As the results show, \( \varphi \) asymptotically approaches 1 as \( C \) increases. Also, the growth in \( C \) required to maintain a given \( \varphi \) is not linear in \( D \). In fact, the difference in \( C \) required to maintain a given \( \varphi \) becomes more noticeable as \( \varphi \to 1 \). For example, for \( D = 5 \), a cache size of 100 is required for \( \varphi = 0.90 \), but the for \( D = 25 \), a cache size of nearly 2700 is required.

Figure 3.16 shows the behavior of \( \varphi \) for large values of \( D \). As can be seen from these plots, for \( D > 50 \), the cache size required to maintain \( \varphi \geq 0.90 \) becomes unreasonably large. In fact for \( \varphi > 0.97 \), \( D = 25 \) seems a realistic upper bound on \( D \).

A small C-based simulation program was developed to simulate the program model listed in Figure 3.2. By explicitly tracking the various transitions, \( \varphi \) was determined experimentally for various values of \( C \) and \( D \). Figures 3.17 and 3.18 contain these experimental results. The simulated values and values given by Theorem 3.4 are plotted against the cache size, \( C \), for a given number of disks, \( D \).

The simulations were conducted for 1000 and 10,000 total fetches. If \( n \) represents the total number of fetches, then \( pn \) f-transitions and \( (1 - \varphi)n \) r-transitions were taken. Therefore, the total data size in blocks is given by \( \varphi nD + (1 - \varphi)n \). As the number of fetches increases, the behavior of the simulations should approach the

\(^7\)The graphs all list \( \varphi \) as the success ratio. This term will be used again in Chapter 4.
Figure 3.15: Predicted $\varphi$ for small values of $D$. 
Figure 3.16: Predicted $\varphi$ for large values of $D$. 

---

**Cache Performance Predicted by Markov Analysis**

- **Success Ratio** vs **Cache size (blocks)**
- Lines represent different values of $D$: $D=5$, $D=10$, $D=15$, $D=20$.
steady-state behavior predicted by the Markov model. For small data sizes (i.e., 1000 fetches), the simulation value of \( \varphi \) should be higher than the predicted value of \( \varphi \), since the data size is within an order of magnitude of the cache size. The initial transient behavior of the system will tend to favor interior states over the edge states, but as the data size increases, the effect of this initial transient diminishes. As shown in the graphs, the steady-state behavior of the Markov model is a good prediction of the simulated behavior.

The average parallelism, \( D\varphi + (1 - \varphi) \), is plotted in Figure 3.19. As defined, the average parallelism approaches \( D \) as \( \varphi \to 1 \), indicating that all I/O operations are able to fetch \( D \) blocks.

### 3.7 Conclusions

As shown by the simulation results, the steady-state analysis of this Markov model accurately models the simulated behavior of the program model listed in Figure 3.2. In the process of analyzing this Markov model, the relative-steady state probability of each state was determined. By summing this probability over all states, an expression was derived which expresses the average parallelism of an I/O operation in terms of \( C \) and \( D \). This expression indicates the cache size requirements for a desired level of I/O parallelism.

Although the quantity \( D \) was given as the number of disks, \( D \) can be viewed as the number of I/O streams available. Each stream contains exactly one run, and each I/O operation must read one block from each stream. The system, however, must still wait for all I/O operations initiated together to complete before initiating the next I/O request.
Figure 3.17: Predicted and simulated $\phi$ for $D = 3$ and $D = 5$. 
Figure 3.18: Predicted and simulated $\rho$ for $D = 10$ and $D = 25$. 
The I/O streams can thus be mapped to physical devices without affecting the predicted behavior of the cache. For example, this Markov model correctly predicts the cache behavior when more than one run is stored on each disk or when all runs are stored on one disk. These two cases will be simulated in Chapter 4, and the simulation results will match the behavior predicted by this Markov analysis.
Chapter 4

Analysis of Prefetching Strategies

In the previous chapter, a Markov model for prefetching was developed and analyzed; in that model, one block from each of the $D$ I/O streams was fetched on an I/O operation. By analyzing this Markov model, the average parallelism was derived as a function of the cache size and the number of streams, $D$.

In this chapter several related prefetching models will be studied. These strategies will be divided into single-disk and multiple-disk approaches, and then classified by the degree of prefetching. By analyzing the average I/O time and observing the simulated behavior, the I/O performance and cache requirements for these new strategies will be determined.

4.1 Introduction

When all runs are stored on one disk, all I/O requests must be serviced sequentially. The first strategy will fetch only the demand-fetch block on an I/O operation. This strategy requires enough memory to store only one block from each of the $k$ runs. Analysis has shown that the average seek distance in this case depends on $k$ [13].

By increasing the size of the cache and by prefetching several blocks on every I/O operation, the I/O time can be reduced. Since all $k$ runs reside on a single disk, the I/O requests are processed sequentially. As a result, the best one can achieve is a reduction in the average seek time and rotational latency, by amortizing these costs over a fetch of several blocks. Two strategies will be studied to accomplish this
goal. In the first strategy, \( N \) blocks will be fetched from a single run on each I/O operation. In the second strategy, the next block from each run will be fetched on each I/O operation.

When \( D > 1 \) disks are available, two more factors can contribute to a decrease in the total I/O time. First, the possibility for concurrency is introduced since operations can proceed simultaneously at each of the \( D \) disks. Secondly, the average seek distance can be reduced or eliminated entirely, since the runs are distributed evenly across \( D \) disks.

If all \( D \) disks are accessed simultaneously, I/O requests at different disks will overlap. In the ideal case, this overlap can result in a reduction in time by a factor of \( D \). When the variance in the rotational latency and seek time is considered, however, the measured speedup may be less than this ideal factor of \( D \).

Another method of obtaining the overlapped operation of \( D \) disks is to fetch \( N \) contiguous blocks from a disk on each request. The computation can continue to proceed as soon as the first of the \( N \) blocks is available. If the next request is issued to a different disk, this second group of \( N \) operations and the remaining \((N - 1)\) operations from the first batch can be serviced concurrently. In addition to this overlap, the average time for an I/O operation is reduced, since only one seek and one rotational latency are required to fetch the \( N \) blocks.

Each prefetching strategy can be implemented using either synchronized or unsynchronized prefetches. With synchronized prefetching, after initiating a group of prefetches, the CPU will wait until all prefetches complete before continuing. With unsynchronized prefetching, the CPU does not wait for any prefetches to complete; as soon as the demand fetch completes, the CPU will continue. Therefore, the remaining prefetches can actually overlap in time with demand fetches being serviced at other disks.
4.2 Overview of Prefetching Strategies

The prefetching experiments will be divided into single-disk and multiple-disk cases. For the single disk case, two strategies will be discussed. The first, studied by Kwan and Baer [13], fetches only the demand-fetch block. The second strategy will fetch the next block from each run on every I/O operation in order to make the seek distance independent of the merge order, $k$. Even though all runs are stored on one disk, the Markov analysis from Chapter 3 applies to this strategy as well. Additionally, the effect of fetching $N > 1$ blocks on both of these strategies will be studied.

When $D > 1$ disks are available, the $k$ runs will be distributed evenly across the $D$ disks. As a result, two special cases arise— one$^8$ in which $k = D$ and the other in which $k > D$.

As a logical extension of the two single-disk strategies, four prefetching strategies can be considered in the multiple-disk case. The single-disk strategies focused on amortizing the seek and rotational latency. With $D$ disks, however, a new element to be considered is the potential concurrency available. Four strategies arise by combining the goals of possible concurrent operations and reduction in the seek-time.

**Demand run only:** In this strategy, only the demand-fetch block is retrieved on each I/O operation: no prefetches are issued. Even though the average seek distance is still a function of $k$, this distance is still lower than the distance from the single-disk case, since each disk contains $\frac{k}{D}$ runs.

**Demand disk, all runs:** This strategy is analogous to the single-disk “all runs” strategy. The average seek distance is made independent of $k$ by reading the next block from each run in order. However, only the runs on the same disk

---

$^8$The case of $k < D$ can be analyzed as the $k = D$ case.
as the demand-fetch block are accessed. When the prefetched are synchronized, no concurrency is possible, but if the prefetched are unsynchronized, operations may overlap at different disks. However, the degree of concurrency should be less than $D$.

**All disks, all runs:** In the ideal case, this strategy can potentially exhibit an overlap of $D$ operations, since the next block from each run on every disk is fetched on every I/O operation. The average seek distance is made independent of $k$ by accessing the runs in order on each disk. Concurrency is possible since $D$ requests, one at each disk, can be serviced simultaneously.

**All disks, one run:** This strategy attempts to exploit the possible concurrency of the "all disks all runs" strategy but does not address the reduction of seek distance. On each I/O operation, one run from each disk is accessed: the demand-fetch block and one other block from each of the remaining disks. If no seek is required to fetch the demand-fetch block, the next block will be fetched from the previously accessed run on each disk; otherwise, the next run on each disk will be accessed. When compared with the "all disks all runs" strategy, this strategy should exhibit the same concurrency but require a smaller cache. However, the average seek depends on $k$ with this strategy, whereas the average seek is independent of $k$ for the "all disks all runs" strategy.

For simplicity, $k$ will be assumed to be a multiple of $D$; if not, $\lceil \frac{k}{D} \rceil$ should be substituted for $\frac{k}{D}$ in the subsequent analysis. Fetching $N$ blocks on each access to a run can be used in conjunction with each of these four approaches. In subsequent sections, each prefetching strategy will be analyzed independently. The average I/O time will be derived and used to predict the total execution time for each strategy. In addition, the memory requirements for each strategy will be discussed.
4.3 Simulation Model

The Rice C Simulation Package (CSIM) [4] is used to construct a process-oriented simulation model of the merge phase of external mergesort. The initial phase which creates the sorted runs on disk is not analyzed for the reasons discussed in Chapter 1. The simulation model consists of a CPU, a cache, a disk subsystem, and blocks of data. Initially, sorted runs are generated such that subsequent depletions will follow the block-depletion model.\(^9\) Only the largest key value in each block is maintained.

Once the runs have been created, the merge phase can be simulated. At each step, the CPU depletes one of the memory-resident blocks, simulating the merging of records to create the final sorted run. The next block required from the run from which a block was depleted will be termed the demand-fetch block. If the demand-fetch block exists in the cache, it is retrieved from the cache without initiating any I/O operation. If the demand-fetch block is not found in the cache, an I/O request must be initiated. If the cache contains enough room for prefetches, a request for these prefetches will be initiated along with the request for the demand fetch; otherwise, only the request for the demand-fetch block is issued.

Each request for a block—demand fetch or prefetch—will be queued at the appropriate disk as an individual request. A separate process is allocated for each request, and this process suspends\(^{10}\) while that block is in the appropriate disk queue. In this manner, each block can be serviced independently. Since each block is treated separately, prefetches can be processed concurrently while the CPU continues with the merge. In order to synchronize prefetches, a mechanism which allows the CPU

\(^9\)At each time step, one block will be depleted from a randomly chosen run. This data modeling scheme is discussed previously in Section 2.3.3 of Chapter 2.

\(^{10}\)In CSIM, whenever a process waits in a queue, it is suspended until it receives its service allocation. If several elements must be added consecutively to a queue, a separate process must be allocated to add each element.
to wait on a prefetch is implemented. This general framework for the I/O provides the generality necessary to simulate several prefetching strategies.

4.3.1 Simulation Parameters

Each run is comprised of exactly 1000 blocks and occupies \( m = \frac{1000}{150} \approx 6.667 \) cylinders. Each disk block is 4096 bytes in length and holds 64 records. The total data size consists of 320,000 records when \( k = 5 \) and 1.6 million records when \( k = 25 \).

In analytically studying the performance of each prefetching strategy, the effects of the three components of the I/O cost will be studied. These three components of the disk's access time—seek time, rotational latency, and transfer time—are explained in Section 1.1 of Chapter 1. To aid in the analysis, the following quantities are defined:

\( S \) : The seek time per cylinder. The seek time for an I/O request will be the product of the seek distance and \( S \).

\( R \) : The average rotational latency. This quantity is shown to be half of the time taken for one full revolution of the platters [13].

\( T \) : The transfer time per block is exactly \( T \).

\( m \) : The length of each run in cylinders.

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

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

\( \Pi \) : The average time to fetch one block. This quantity is the sum of the average seek time, average rotational latency, and average transfer time.
The disk model used by the simulator has the following parameters: $T = 2.048$ ms, $\mathcal{R} = 8.333$ ms, and $S = 0.5$ ms/cylinder, 10 blocks/track, 150 blocks/cylinder, and 150K blocks/disk. However, the disk model establishes a minimum seek time of 2.0 ms and a maximum seek time of 25.0 ms. This "clipping" of the seek time ensures that the disk model stays within reasonable limits of actual disk behavior. The acceleration and deceleration which occur in actual head movement is difficult to model analytically. Table 4.1 shows a summary of the expected average I/O time to fetch one block for each strategy.

4.4 Explanation of Simulation Statistics

In order to understand the performance of each prefetching strategy, the simulation program can collect several statistics. These statistics can then be used to determine the average I/O time, and the cache behavior. The following terminology will be used in exploring the simulation results:

**Cache size:** This parameter can be adjusted to fix the maximum capacity, expressed in blocks, of the cache.

**Blocks Cached:** The total number of blocks added to the cache during the course of the simulation.

**Blocks Prefetched:** The total number of blocks which were fetched before they were needed. These "prefetched" blocks are added to the cache. Any demand fetch which is issued along with other prefetches is also added to the cache, but does not increment this "blocks prefetched" count.

**Synchronized:** After initiating a group of prefetches, the CPU will wait until all prefetches complete before continuing.
Table 4.1: The average time to fetch one block.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Seek Time</th>
<th>Rotational Latency</th>
<th>Transfer Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Disk, D=1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demand Run Only</td>
<td>$m(kS^{kN})$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td>All Runs</td>
<td>$m\left(\frac{k}{kN}\right)S$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td><strong>D Disks, k&gt;D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Additional overlap possible if N &gt; 1.</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demand Run Only</td>
<td>$m\left(\frac{k}{3N}D\right)S$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td>Demand Disk, All Runs</td>
<td>$m\left(\frac{k}{kN}D\right)S$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td>All Disks, All Runs</td>
<td>$m\left(\frac{k}{kN}D\right)S$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$D$ operations can occur concurrently.</td>
</tr>
<tr>
<td>All Disks, 1 Run(^{11})</td>
<td>$m\left(\frac{k}{3ND}D\right)S$</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$D$ operations can occur concurrently.</td>
</tr>
<tr>
<td><strong>D Disks, k=D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Additional overlap possible if N &gt; 1.</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demand Run Only</td>
<td>0</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td>All Disks, 1 Run</td>
<td>0</td>
<td>$\frac{1}{N}R$</td>
<td>$T$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$D$ operations can occur concurrently.</td>
</tr>
</tbody>
</table>

**Unsynchronized:** The CPU does not wait for any prefetches to complete; as soon as the demand fetch completes, the CPU will continue. Therefore, the remaining prefetches can overlap in time with demand fetches being serviced at other disks.

**Wait Ratio:** If a demand-fetch block is in the process of being prefetched due to a previous prefetch request, the system will wait for this prefetch to complete, instead of issuing a new I/O request. If the demand-fetch block is found in the cache, no wait is required. The wait ratio is the fraction of prefetches which required the CPU to wait for the I/O to complete.

\(^{11}\)The seek time for the "all disks one run" strategy is shown for only the demand disk.
Success Ratio: This statistic is the probability that a given prefetch can be initiated because the cache has enough room to accommodate all blocks. The success ratio is equivalent to the derived quantity \( \varphi \) from Chapter 3.

Total Blocks Accessed: This figure represents the total size of the data. If the cache does not have enough room to accommodate all of the prefetches, only the demand fetch is retrieved. Additionally, no cache blocks are ever replaced, because an older cached block from a run will always be depleted before a new prefetch from that same run. This is so because each run contains records sorted by increasing order of key values. Consequently, each block is read exactly once.

Total Seeks: The sum of the number of seeks on each disk.

Average Non-Zero Seek: The average seek distance, given in cylinders, and the average seek time, given in milliseconds, can be determined for each seek. This average figure is computed over all seeks, and not over all accesses, because each access may not necessarily require a seek.

Average All Seek: This quantity provides the average seek distance and time for each access. Since some accesses will not require a seek, this quantity should be lower than the “average non-zero seek”. Each I/O operation can be viewed as requiring this average seek, even though the seek distance is an integer. In subsequent sections, this quantity will be used to estimate the total execution time.

Total Time: Expressed in seconds, this quantity represents the total execution time of the merge phase. If the “infinitely fast CPU” option is specified, this time reflects the I/O time of the particular prefetching strategy, since no computational delays are present.
4.5 One-Disk Results

When only one disk is available, all of the runs will be stored on that disk. As a result, no overlap of I/O operations on different disks is possible. If the CPU is modeled as being infinitely fast, no computational delays are present. As a result, the CPU can issue the next I/O request as soon as the demand-fetch block is serviced. Additionally, no overlap of I/O and computation is possible. Hence, for the single-disk case with an infinitely fast CPU, no difference in execution time exists between strategies which use synchronized prefetching and those which use unsynchronized prefetching.

Since all I/O operations occur at each disk with no intervening CPU delays, the total execution time is the product of the total number of blocks and the average I/O time for each block.\(^{12}\) The total number of blocks is the product of the number of runs and the length of each run; for all simulations, each run contains exactly 1000 blocks. The total execution time is given by \(1000k\Pi\).

4.5.1 Demand Run Only

In this model, introduced by Kwan and Baer [13], \(k\) runs are placed contiguously on a single disk, and the memory storage consists of \(k\) blocks, one for each run. When a memory-resident block of a run is depleted, the next block of that run is fetched from the disk. In this model, the next block to be depleted is chosen randomly from the \(k\) runs with equal probability.

The analysis by Kwan and Baer [13] shows that when \(k\) runs are read in random order, \(\frac{k^2}{3} + O(k)\) moves will be required. Since the \(\frac{k^2}{3}\) term dominates for most values of \(k\), each access will skip over an average of \(\frac{k}{3}\) runs [13]. For smaller values of \(k\),

---

\(^{12}\)Defined previously in Section 4.3.1, \(\Pi\) represents the average time to fetch one block
Table 4.2: Simulation results for one disk.

<table>
<thead>
<tr>
<th></th>
<th>Demand Run Only</th>
<th>Run Only</th>
<th>All runs, One Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k = 5 runs</td>
<td>k = 25 runs</td>
<td>k = 5 runs</td>
</tr>
<tr>
<td>Blocks Cached</td>
<td>-</td>
<td>-</td>
<td>4995</td>
</tr>
<tr>
<td>Blocks Prefetched</td>
<td>-</td>
<td>-</td>
<td>3996</td>
</tr>
<tr>
<td>Wait Ratio</td>
<td>-</td>
<td>-</td>
<td>0.0115</td>
</tr>
<tr>
<td>Total Blocks Accessed</td>
<td>5000</td>
<td>25000</td>
<td>5000</td>
</tr>
<tr>
<td>Total Seeks</td>
<td>4001</td>
<td>23900</td>
<td>4006</td>
</tr>
<tr>
<td>Average All Seek (cy)</td>
<td>10.622</td>
<td>55.290</td>
<td>5.331</td>
</tr>
<tr>
<td>Total Time</td>
<td>78.75</td>
<td>770.46</td>
<td>65.53</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>1</td>
<td>1.201</td>
</tr>
</tbody>
</table>

Fetching N blocks from each run

<table>
<thead>
<tr>
<th></th>
<th>Total Time, N = 10</th>
<th>Total Time, N = 20</th>
<th>Total Time, N = 30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17.66</td>
<td>124.58</td>
<td>15.74</td>
</tr>
<tr>
<td></td>
<td>14.00</td>
<td>87.39</td>
<td>13.09</td>
</tr>
<tr>
<td></td>
<td>12.13</td>
<td></td>
<td>11.67</td>
</tr>
</tbody>
</table>

However, the \( O(k) \) term will become more significant. As a result, a slight discrepancy between the expected and measured seeks may appear for these smaller values of \( k \).

The average I/O time required to fetch a block in this model is given by the sum of the transfer time, \( T \), the average rotational latency, \( R \), and the average seek time. Since a run occupies \( m \) consecutive cylinders, and an average of \( \frac{k}{3} \) runs are skipped over on any read, the average time to access a block is given by:

\[
\Pi = m \left( \frac{k}{3} \right) S + R + T \tag{4.1}
\]

Table 4.2 shows the simulation results for \( k = 5 \) and \( k = 25 \). Equation 4.1 predicts an average seek distance of \( m \left( \frac{k}{3} \right) = 11.105 \) cylinders for \( k = 5 \) and 55.55 cylinders for \( k = 25 \). These correspond well with the simulation results of 10.622 and 55.290, respectively. For \( k = 5 \), \( \Pi = 15.936 \) ms, and the expected total time is 79.68 seconds. The simulation time of 78.75 seconds corresponds well with the expected value.
For \( k = 25 \), the expected average seek distance of \( m(\frac{k}{n}) = 55.56 \) cylinders closely matches the measured value of 55.290. Since \( \Pi = 38.161 \) ms, the expected total time is 954.025 seconds. The simulated total time of 770.46 is lower, because the clipping of the seek time\(^{13} \) at 25 ms limits the time taken by long seeks. The savings in time on these long seeks reduces the total seek-time component, thereby reducing the total execution time.

The slight discrepancy between the measured and expected seek distances appears only when \( k = 5 \). This discrepancy is not unexpected, though, since the approximation of \( \frac{k}{3} \) for the average seek distance is less accurate for smaller values of \( k \).

When \( k \) is increased from 5 to 25, the total execution time should increase by at least a factor of 5, since the size of the data has increased fivefold. By noting that the total seek distance is proportional to \( k^2 \), the total execution time is expected to increase more significantly than a factor of 5. The expected total execution times of 954.025 seconds for \( k = 25 \) and 79.68 for \( k = 5 \) differ by a factor of 11.97. The respective simulated times of 770.46 and 78.75, however, differ by only a factor of 9.783. The discrepancy between the expected and simulated times for \( k = 25 \) is due to the non-linear seek model explained in the previous paragraph.

### 4.5.2 "Demand Run Only": Prefetching from a Single Run

In this strategy, \( N \) blocks will be fetched from the demand-fetch run\(^{14} \) on each I/O operation. The goal is to reduce the average seek time and average rotational component of each block by amortizing the initial seek time and rotational latency over \( N \) consecutive blocks.

---

\(^{13}\)As explained in Section 4.3.1, the seek time is clipped at 25 ms in order to model the disk more realistically.

\(^{14}\)The run which contains the demand-fetch block will be referred to as the demand-fetch run.
The average I/O time for this strategy is given by:

$$\Pi = m \left( \frac{k}{3N} \right) S + \frac{1}{N} R + T$$  \hspace{1cm} (4.2)

The results from Table 4.2 show diminishing marginal returns as \(N\) increases. Only the seek time and rotational latency are amortized over \(N\) blocks. Since the transfer time is unaffected, \(\Pi\) approaches \(T\) asymptotically as \(N\) increases.

For \(k = 5\), Equation 4.2 predicts the simulated behavior quite well. For \(N = 10\), \(\Pi = 3.4358\) ms, and the expected total time is 17.18 seconds. This figure corresponds well with the simulated total time of 17.66 seconds. Similarly, for \(N = 20\), \(\Pi = 2.7424\) ms, and the expected total time is 13.712 seconds. Finally, for \(N = 40\), \(\Pi = 2.395\) ms, and the expected total time is 11.95 seconds. All of these values closely match their measured counterparts.

For \(k = 25\), the simulated behavior deviates slightly from that predicted by Equation 4.2 due to the non-linear simulation model for seeks. For \(N = 10\), \(\Pi = 5.659\) ms and the expected total time is 96.34 seconds. Since long seeks are clipped, the measured time of 87.39 is lower than the expected time. Figure 4.1 shows the effect on the total time as \(N\) is varied for data sizes of \(k = 5\) and \(k = 25\).

**Memory Requirements: “Demand Run Only”**

In order to fetch \(N\) blocks from the demand run on every fetch, a cache with a capacity of \(kN\) blocks is required. These cache requirements can be explained by the following observations. When each of the \(k\) runs has fetched \(N\) blocks, the cache will contain \(kN\) blocks. All subsequent depletions, however, can continue without requiring a fetch until some run no longer has any cached blocks. At least \(N\) blocks will have been depleted, though, before this fetch is required. The new fetch of \(N\)
Figure 4.1: The effect of fetching N blocks on the total time for the single-disk case. In both graphs, the cache is large enough to ensure that all prefetches are successful, i.e., the success ration is 1.
blocks can thus be accommodated in the cache. Therefore, with a cache size of \( kN \) blocks, each I/O operation can always fetch \( N \) blocks.

### 4.5.3 "All Runs": Fetching from All Runs

In this strategy, every time an I/O request is issued, the next block from each of the \( k \) runs is fetched. On each pass, the disk is scanned from one end to the other, and the direction is reversed after each pass. As a result, each block can be fetched with a seek of \( m \) cylinders, and only \( (k - 1) \) seeks are required to fetch \( k \) blocks. These \( (k - 1) \) prefetches will be added to the cache. Additionally, the average seek is now the distance required to traverse one run—\( m \) cylinders.

The average time to fetch a block is given by:

\[
\Pi = m \left( \frac{k - 1}{k} \right) S + \mathcal{R} + T
\]

Table 4.2 shows the result of fetching the next block from each run on every I/O operation.

For \( k = 5 \), the expected average seek distance of \( m(\frac{k-1}{k}) = 5.333 \) cylinders matches the measured value of 5.331 cylinders. Also, the average non-zero seek of 6.654 cylinders closely matches \( m = 6.667 \), the length of each run. In other words, each seek traverses the distance between runs.

Amortizing the seeks leads to a much greater savings for the 25-run case, resulting in a more noticeable difference in the total execution time. For \( k = 25 \), the data shows that only 24006 seeks are required to access 25000 blocks, and that each seek covers 6.665 cylinders, the length of each run. The expected average seek distance of \( m(\frac{k-1}{k}) = 6.4003 \) cylinders and the expected total time of 339.53 seconds closely match their measured counterparts. Although the seek time is reduced, the rotational latency and transfer time are unaffected. As a result, the total execution time does
not decrease as significantly as does the seek time. Even though the seek distance has been reduced by a factor of 8.63 over the "demand run only" case, the total execution time has been reduced by only a factor of 2.2264.

**Fetching N Blocks from Each Run**

By fetching $N$ blocks from each run on every I/O operation, the reduction in seek distance can be combined with the amortization of seek time and rotational latency. As a result, the average seek time and rotational latency components will be reduced by an additional factor of $N$.

The average time to fetch one block is given by:

$$\Pi = m \left( \frac{k - 1}{kN} \right) S + \frac{1}{N} R + T$$

Table 4.2 shows the results of this strategy. For $k = 5$ and $N = 10$, $\Pi = 3.14798$ ms, and the expected total time is 15.7399 seconds. This expected value matches the simulated total time of 15.74 seconds. Similarly, for $N = 20$, the expected total time is 12.99, and for $N = 40$, the expected total time is 11.62. All of these expected values are corroborated by the simulation results.

When $k = 25$ and $N = 10$, $\Pi = 3.201$ ms, and the expected total time of 80.03 seconds matches the measured time of 80.00 seconds. Since each seek corresponds to the length of one run, no long seeks are ever encountered, and, hence, no seeks must be clipped. As a result, the simulation results match the expected values.

Figure 4.1 shows the effect on the total time as $N$ is varied for data sizes of $k = 5$ and $k = 25$. As $N$ increases, the average I/O time, $\Pi$, approaches $T$, the transfer time. As a result, both single-disk strategies converge for very large $N$. 

Figure 4.2: Single-disk “All Runs”: Cache performance as a function of $N$.

Memory Requirements: “All Runs”

Figure 4.2 shows the behavior of the cache as a function of $N$. The success ratio shown indicates the probability of having enough free cache blocks to accommodate all blocks from a prefetch. This quantity corresponds to $\varphi$ which was derived in Chapter 3.

For the range of $N$ displayed, the results seem to indicate that the cache size required for a success ratio of 1 is $(C_1 + kN), 1 \leq N \leq 10$, where $C_1$ is the size of the cache for $N = 1$. When $N = 1$, the cache requirements can be predicted by the analysis techniques from Chapter 3. This Markov analysis shows that the cache size, $C_1$, grows as a super-linear function of the success ratio, $\varphi$. Since $C_1$ itself grows
super-linearly, the additional \( kN \) blocks required to support fetches of \( N = 5 \) or \( N = 10 \) blocks are not a significant addition.

As displayed in Figure 4.1, the total time is not reduced significantly when \( N \) is increased from 5 to 10. As a result, fetching \( N > 10 \) blocks would not lead to a significant savings in total time.

4.5.4 Single-Disk: Finite-Speed CPU

When an infinitely-fast CPU was used, no difference existed between unsynchronized and synchronized prefetching for the single-disk case. When computational delays are introduced, however, strategies with synchronized prefetches should begin to perform worse than their unsynchronized counterparts, and both strategies should perform worse than the corresponding strategies with an infinitely fast CPU.

Figure 4.3 shows the performance of both single-disk prefetching strategies with \( k = 25 \) and \( N = 10 \). The total execution time is plotted against the time to merge one block. As the time required to merge approaches zero,\(^{15}\) the difference between the synchronized and unsynchronized cases narrows. The synchronized time decreases linearly because the CPU activity does not overlap with the I/O; reducing this computational delay leads to a linear reduction in the total time. With unsynchronized prefetching, however, the I/O can overlap with the CPU’s computation time. During these delays, however, a new I/O operation can not be issued. As this delay increases, the probability that the disk is idle increases. Since the computational delay is small when compared with the average time to fetch one block, this increase in total time due to an idle disk is slight.

\(^{15}\)The infinitely fast CPU speed can be considered to require zero time to merge one block.
Figure 4.3: The performance of single-disk prefetching strategies when a finite-speed CPU is used. The cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1.

4.6 Multiple-Disk Results

The four $D$-disk prefetching strategies, discussed in Section 4.2 will be simulated with $D = 5$ disks and two data sizes—$k = 5$ and $k = 25$ runs; each run will contain exactly 1000 blocks. Tables 4.3 shows the results of the four prefetching strategies when $k = D = 5$. Table 4.4 shows the corresponding results when $k = 25$ and $D = 5$.

4.6.1 Estimating the Total Execution Time

When $k = D$, each disk contains exactly one run. As a result, random seeks are eliminated entirely, and the blocks from each run can be read sequentially. Seeks will be required only when a cylinder boundary must be crossed. Since each cylinder
contains 150 blocks, one seek is required for each 150 blocks accessed. As a result, only 0.667% of the blocks accessed will require a seek, and, hence, these seeks can be ignored when computing \( \Pi \).

When \( k > D \), each disk will contain \( \frac{k}{D} \) runs. As a result, the seek component must be considered. Although the seek distance is proportional to \( k \), it is reduced by a factor of \( D \) from the single-disk case.

When \( k = D \), the strategies "demand run only" and "demand disk all runs" are identical, since each disk contains exactly one run. Similarly, the "all disks one run" and the "all disks all runs" strategies are identical. As a result, only the "demand run only" and "all disks one run" strategies will be discussed when \( k = D \).

When \( N > 1 \) blocks are fetched in conjunction with the "demand run only" strategy, additional overlap is possible with I/O requests being processed at other disks. Once the first block of a fetch of \( N \) blocks has been serviced, the CPU can proceed immediately, deplete another block, and issue a fetch for another \( N \) blocks. If these \( N \) blocks are issued at another disk, they will be serviced concurrently with the pending \((N - 1)\) fetches.

The total time to fetch a block from a disk can be estimated by computing \( \Pi_1 \), the average service time for the first block, and \( \Pi_{N-1} \), the average service time for the remaining \((N - 1)\) blocks.

Predicting the total execution time, however, is not straightforward due to this overlap. Each I/O operation can be viewed as a request for one block followed by a request for \((N - 1)\) blocks. Once the first block has finished, the next block required by the CPU may be one of the \((N - 1)\) blocks being serviced, or the next block may be from a different run. In this second case, that run may reside on the same disk as the \((N - 1)\) blocks still being serviced, or it may reside on a different disk.
Table 4.3: Merging 5 runs using 5 disks. Each run contains 1000 blocks. A large cache size was used to ensure that all prefetches were successful.

<table>
<thead>
<tr>
<th></th>
<th>All disks</th>
<th>All disks 1 run(^{16})</th>
<th>Demand Disk, All Runs</th>
<th>Demand Run Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks Cached</td>
<td>4995</td>
<td>4995</td>
<td>4995</td>
<td>-</td>
</tr>
<tr>
<td>Blocks Prefetched</td>
<td>3996</td>
<td>3996</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wait Ratio (unsynch)</td>
<td>0.0398</td>
<td>0.0410</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wait Ratio (synch)</td>
<td>0.4687</td>
<td>0.4827</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total Blocks Accessed</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
</tr>
<tr>
<td>Total Seeks(^{17})</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Avg.Non-Zero Seek(cy)</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Average All Seek (cy)</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>Total Time (unsync)</td>
<td>11.66</td>
<td>11.68</td>
<td>52.20</td>
<td>52.20</td>
</tr>
<tr>
<td>Speedup(^{18}) (unync)</td>
<td>4.477</td>
<td>4.468</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total Time (sync)</td>
<td>15.99</td>
<td>15.99</td>
<td>52.20</td>
<td>52.20</td>
</tr>
<tr>
<td>Speedup (sync)</td>
<td>3.264</td>
<td>3.264</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Fetching N blocks from each run**

<table>
<thead>
<tr>
<th></th>
<th>Unsynchronized Prefetches</th>
<th>Synchronized Prefetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time, N=10</td>
<td>3.09</td>
<td>3.47</td>
</tr>
<tr>
<td>Total Time, N=20</td>
<td>2.64</td>
<td>2.77</td>
</tr>
<tr>
<td>Total Time, N=30</td>
<td>2.44</td>
<td>2.53</td>
</tr>
<tr>
<td>Total Time, N=40</td>
<td>2.33</td>
<td>2.41</td>
</tr>
<tr>
<td>Total Time, N=60</td>
<td>2.26</td>
<td>2.30</td>
</tr>
<tr>
<td>Total Time, N=80</td>
<td>2.20</td>
<td>2.24</td>
</tr>
<tr>
<td>Total Time, N=100</td>
<td>2.19</td>
<td>2.20</td>
</tr>
</tbody>
</table>

\(^{16}\) The discrepancy between the times for “all disks all runs” and “all disks one run” is explained in Section 4.6.4.

\(^{17}\) A total of \(m - 1\) seeks will occur on each of the \(D\) disks, resulting in a total of \(|m - 1|D\) seeks. Note that \(m \approx 6.667\), is the length of each run in cylinders.

\(^{18}\) The speed-ups are given with respect to the time for the unsynchronized “demand run only” strategy.
Table 4.4: Merging 25 runs using 5 disks. Each run contains 1000 blocks. A large cache is used to ensure that all prefetches are successful.

<table>
<thead>
<tr>
<th></th>
<th>All disks</th>
<th>All disks</th>
<th>Demand Disk,</th>
<th>Demand Run</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All runs</td>
<td>1 run</td>
<td>All Runs</td>
<td>Only</td>
</tr>
<tr>
<td>Blocks Cached(^\text{19})</td>
<td>24975</td>
<td>24975</td>
<td>24975</td>
<td>-</td>
</tr>
<tr>
<td>Blocks Prefetched</td>
<td>23976</td>
<td>19810</td>
<td>19980</td>
<td>-</td>
</tr>
<tr>
<td>Wait Ratio (unsynch)</td>
<td>0.0069</td>
<td>0.0116</td>
<td>0.0021</td>
<td>-</td>
</tr>
<tr>
<td>Wait Ratio (synch)</td>
<td>0.4130</td>
<td>0.4646</td>
<td>1.0000</td>
<td>-</td>
</tr>
<tr>
<td>Total Blocks Accessed</td>
<td>25000</td>
<td>25000</td>
<td>25000</td>
<td>25000</td>
</tr>
<tr>
<td>Total Seeks</td>
<td>20030</td>
<td>19530</td>
<td>22208</td>
<td>19659</td>
</tr>
<tr>
<td>Average All Seek (cy)</td>
<td>5.331</td>
<td>8.774</td>
<td>7.672</td>
<td>10.447</td>
</tr>
<tr>
<td>Total Time (unsync)</td>
<td>67.23</td>
<td>90.38</td>
<td>250.31</td>
<td>390.63</td>
</tr>
<tr>
<td>Speedup(^\text{20}) (unsync)</td>
<td>5.810</td>
<td>4.322</td>
<td>1.561</td>
<td>1</td>
</tr>
<tr>
<td>Total Time (sync)</td>
<td>77.72</td>
<td>111.41</td>
<td>355.90</td>
<td>390.63</td>
</tr>
<tr>
<td>Speedup (sync)</td>
<td>5.026</td>
<td>3.506</td>
<td>1.097</td>
<td>1</td>
</tr>
</tbody>
</table>

Fetching \(N\) blocks from each run

<table>
<thead>
<tr>
<th>Unsynchronized Prefetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time, (N=10)</td>
</tr>
<tr>
<td>Total Time, (N=20)</td>
</tr>
<tr>
<td>Total Time, (N=30)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronized Prefetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time, (N=10)</td>
</tr>
<tr>
<td>Total Time, (N=20)</td>
</tr>
<tr>
<td>Total Time, (N=30)</td>
</tr>
</tbody>
</table>

\(^{19}\) As expected, only \(\frac{1}{5}\) of the total number of blocks are demand-fetches; the remaining \(\frac{4}{5}\) fraction of the total blocks are prefetches.

\(^{20}\) The speed-ups are given with respect to the time for the unsynchronized "demand run only" strategy.
4.6.2 D-Disk, “Demand Run Only” Strategy

In all of the D-disk strategies, k runs are equally distributed over D disks. In this strategy, only the demand-fetch block is read on each fetch. The multiple disks merely reduce the average seek distance.

It is easy to show that the sequence of requests to any disk is random, and hence the analysis of Kwan and Baer [13] holds for each disk. Since each disk contains \( \frac{k}{D} \) runs, the average seek time is \( m(\frac{k}{3D})S \), but the rotational latency is not affected. Since the number of runs on each disk, \( \frac{k}{D} \), is small, a slight discrepancy may appear between the expected and measured numbers of seeks, as explained in Section 4.5.1.

Performance When k=D: “Demand Run Only”

When \( k = D = 5 \) runs are distributed over \( D = 5 \) disks, each disk contains exactly one run. As a result, each run can be read sequentially without requiring any seeks, as discussed earlier. The seek component is therefore eliminated, resulting in the following average time to fetch one block:

\[
\Pi = R + T
\]

For this strategy, \( \Pi = 10.381 \) ms, and the expected total time is 51.905 seconds. The simulated total time of 52.20 seconds corresponds well with this expected value. When only one disk was available, the measured total time was 78.75 seconds. The difference in these two times is attributed entirely to the elimination of the seek component in the \( k = D = 5 \) case.

Fetching N Blocks When k=D: “Demand Run Only”

When \( N \) blocks are fetched on each I/O operation, the average service time can be estimated, as discussed in Section 4.6.1. The average service time for the first block
and the exact service time for the remaining \((N - 1)\) blocks are given by:

\[
\begin{align*}
\Pi_1 &= \mathcal{R} + \mathcal{T} \\
\Pi_{N-1} &= (N - 1)\mathcal{T}
\end{align*}
\]

The total number of fetches required, \(\frac{5000}{N}\), is now the total data size in blocks divided by \(N\). Since operations can potentially overlap at the different disks, the total execution time should be less than \(\frac{5000}{N}(\Pi_1 + \Pi_{N-1})\). Since the CPU must wait for the first block in each request to complete, the minimum execution time should be \(\frac{5000}{N}\Pi_1\).

For \(N = 10\), \(\Pi_1 = 10.381\) ms and \(\Pi_{N-1} = 18.432\) ms. The measured total time of 7.26 seconds lies between the upper bound of \((\Pi_1 + \Pi_{N-1})\frac{5000}{N} = 14.41\) and lower bound of \(\Pi_1(\frac{5000}{N}) = 5.190\) seconds. Concurrency is introduced because the \((N - 1)\) prefetches can overlap with the demand-fetch from another block of \(N\) fetches, if these fetches are being serviced at different disks. If the prefetches are synchronized, then no overlap is possible, and the expected total time should be exactly \((\Pi_1 + \Pi_{N-1})\frac{5000}{N} = 14.41\) seconds. This expected value matches the measured total time of 14.34 seconds.

**Performance When \(k > D\): “Demand Run Only”**

When \(k > D\), each disk contains \(\frac{k}{D}\) runs. As a result, the average seek will skip over \(\frac{k}{3D}\) runs. Therefore, the average time to fetch one block is given by:

\[
\Pi = m \left( \frac{k}{3D} \right) \mathcal{S} + \mathcal{R} + \mathcal{T}
\]

For \(k = 25\), \(\Pi = 15.936\) ms. The expected average seek distance of \(m(\frac{k}{3D}) = 11.11\) cylinders deviates slightly from the measured value of 10.447, and the total expected time of 398.4 seconds corresponds to the measured time of 390.63. The deviation between the measured and expected seek distances is also found for the \(k = 5\) single-disk case. The reasoning was explained previously in Section 4.5.1.
Focusing on N Blocks When \( k > D \): "Demand Run Only"

The average service time for the first block and the exact service time for next \((N-1)\) blocks are given by:

\[
\Pi_1 = \left( \frac{k}{3D} \right) S + R + T \\
\Pi_{N-1} = (N - 1)T
\]

When \( N = 10 \) and \( k = 25 \), \( \Pi_1 = 15.937 \) and \( \Pi_{N-1} = 18.432 \). If prefetches are synchronized, the expected total time should be exactly \((\Pi_1 + \Pi_{N-1})(\frac{25000}{N})=85.88\). This expected value corresponds closely with the measured value of \(87.54\).

However, for unsynchronized prefetches, estimating the total time is not as straightforward. With \( \frac{k}{D} \) runs on each disk, the next request may be from a run which resides on the same disk. This new request of \( N \) blocks can only be processed after the \((N-1)\) prefetches from the previous request complete. As a result, the degree of overlap is less than the \( k = D \) case. The measured total time of 51.55 seconds lies between the lower bound of \( \Pi_1(\frac{25000}{N})=39.8 \) and the upper bound of \((\Pi_1 + \Pi_{N-1})(\frac{25000}{N})=85.88\).

4.6.3 D-Disk, "Demand Disk, All Runs" Strategy

In this strategy, the next block is fetched from each of the \( \frac{k}{D} \) runs on the demand disk. Since the runs are scanned in order,\(^{21}\) \( \frac{k}{D} - 1 \) seeks are required to fetch \( \frac{k}{D} \) blocks. When \( k = D \), this strategy is equivalent to the "demand run only" strategy. Therefore, only the case when \( k > D \) is discussed in this section.

As a result, the average time to fetch one block, without considering any overlap of I/O operations, is given by:

\[
\Pi = m \left( \frac{k - D}{k} \right) S + R + T
\]

\(^{21}\)By reversing the direction of each pass after all runs have been accessed, each seek will traverse one run.
When the prefetches are synchronized, the expected total time is given by 25000\(\Pi\), since each of the 25 runs contains exactly 1000 blocks. If the prefetches are unsynchronized, the CPU can proceed as soon as the demand-fetch block completes. As a result, a new set of I/O requests can be issued to a different disk, thereby introducing some concurrency.

When \(k = 25\), \(\Pi = 13.049\) ms. The expected total time of \(25000\Pi = 326.225\) seconds corresponds to measured synchronized time of 355.90 seconds. The unsynchronized time of 250.31 is lower since fetches can proceed concurrently at other disks.

**Memory Requirements: “Demand Disk, All Runs”**

Figure 4.4 shows the cache requirements for this strategy. For the range of \(N\) plotted, the additional cache blocks required to maintain the same success ratio seems to be a constant offset as \(N\) increases. From the data, this difference appears to be less than 100 blocks when \(N\) increases from 1 to 10. Therefore, supporting these larger values of \(N\) with this strategy seems to require an additional offset in the amount of memory.

### 4.6.4 D-Disk, “All Disks, One Run” Strategy

**Performance When \(k=D\): “All Disks, One Run”**

When \(k = D = 5\), the strategies “all disks, all runs” and “all disks, 1 run” are identical. However, the nature of their implementations is slightly different, resulting in a slight difference in the two execution times. The strategy “all disks, all runs” alternates the direction of the seek on successive passes by alternating the order in which the runs are processed. This alternation introduces a slight difference in the
Figure 4.4: Cache requirements for the “demand disk all runs” strategy, as \( N \) varies. 25 runs, 5 disks.

Scheduling of the processes which prefetch each block, thereby introducing a slight difference of 1–2 ms in the total execution times of the two scenarios. Since this difference is negligible when compared with the total execution time, the implementation does not significantly affect the validity of the results.

Since random seeks are eliminated when \( k = D \), the average time to fetch a block is given by:

\[ \Pi = R + T \]

Since one block is fetched from each disk, \( D \) operations can be performed concurrently. For the case of “all disks”, a perfect overlap of \( D = 5 \) would lead to an expected total time of 10.381 seconds. However, any variance in rotational latency among the disks would lead to a deviation from this ideal overlap of \( D \), and, hence, a
deviation from the ideal execution time. The unsynchronized prefetch times result in a total time of 11.66, but the synchronized prefetch times show a total time of 15.99. If the prefetches are synchronized, the total time taken by the $D$ parallel fetches is equal to the time taken by the slowest fetch. Therefore, the other $(D - 1)$ disks will remain idle while the slowest disk completes its fetch. The increase in total execution time is, therefore, expected.

On each I/O access, one block is the demand-fetch block, and the remaining $(D - 1)$ blocks are prefetches. Note the number of blocks prefetched, as shown in Table 4.3. As expected, only $\frac{1}{D}$ of the total number of blocks are demand-fetches; the remaining $\frac{D-1}{D}$ fraction of the total blocks are prefetches.

**Performance When $k>D$: “All Disks, One Run”**

Since each disk contains $\frac{k}{D}$ runs, each seek will skip on average $\frac{k}{3D}$ runs.\textsuperscript{22} As a result, the average time, at one disk, to fetch one block is given by:

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

This figure represents the service time at only the demand disk. Since $D = 5$ fetches could occur concurrently, the best possible total time is $\Pi \frac{25000}{D} = 5000\Pi$. Any variance in the rotational latency will lead to a deviation from this ideal concurrency of $D$; consequently, the measured execution time would be greater than the ideal 5000$\Pi$ value.

When $k = 25$, $\Pi = 21.493$ ms. When prefetches are synchronized, the measured total time of 111.41 is not far from the ideal total time of 107.46. When prefetches are unsynchronized, additional concurrency is possible if the demand-fetch completes with a service time which is below the expected average. In this case, the CPU

\textsuperscript{22}The $\frac{k}{3D}$ derivation is explained in Section 4.6.2.
can issue another request at each disk, introducing overlap that is not possible if all prefetches are synchronized. For unsynchronized prefetching, the total time of 90.38 shows the expected reduction in time over the synchronized case.

**Memory Requirements: “All Disks, One Run”**

The cache requirements for this strategy are shown in Figure 4.5. Both the \( k = D = 5 \) and \( k = 25, D = 5 \) cases are shown.

When \( k = 5 \), the performance for \( N = 1 \) is predicted by the Markov analysis from Chapter 3, since each disk contains exactly one run. As \( N \) is increased, however, the previous analysis does not hold. The simulation results indicate that the total cache size seems to grow by some offset proportional to \( kN \) from the requirements for the \( N = 1 \) case.

When \( k = 25 \), the analysis from Chapter 3 does not apply to this strategy, since blocks from only \( D = 5 \) runs are fetched on each operation. The results, however, show behavior similar to the \( k = D = 5 \) case, but the cache requirements have grown.

### 4.6.5 D-Disk, “All Disks, All Runs” Strategy

This strategy differs from “all disks one run” only when \( k > D \), since each disk will have more than one run. The case when \( k = 5 \) is discussed previously in Section 4.6.4.

Since each disk contains \( \frac{k}{D} \) runs, the seeks on each disk can be viewed in isolation of the other disks. Since the next block is fetched from each run by scanning in order of runs, \( (\frac{k}{D} - 1) \) seeks will be required to fetch \( \frac{k}{D} \) blocks from each disk. Therefore, the average time, at one disk, to fetch one block is given by:

\[
\Pi = m \left( \frac{k - D}{k} \right) S + R + T
\]
Figure 4.5: Cache requirements for “all disks one run”, D=5 disks
This figure represents the service time at only one disk. To fetch one block from each of the 25 runs, each disk will require an average of $5 \Pi$ time, but the five disks can service requests concurrently. As a result, the best possible total time is $\Pi \frac{25000}{D} = 5000 \Pi$. Any variance in the rotational latency will lead to a deviation from this ideal concurrency of $D$; consequently, the measured execution time would be greater than the ideal 5000$\Pi$ value.

When predicting the expected total time, the ideal behavior given by $\Pi$ will be assumed. Any variance in the rotational latency will lead to longer total execution times. When $k = 25$, $\Pi = 13.048$ ms. As a result, the measured unsynchronized time of 67.23 is higher than the expected total time of $(5 \Pi)(1000) = 65.240$.

**Fetching N Blocks: “All Disks, All Runs”**

The first block in the fetch of $N$ blocks will incur a seek and rotational latency in addition to the transfer time. The remaining $(N-1)$ blocks will only be charged the transfer time. As a result,

$$\Pi_1 = \left(\frac{k-D}{k}\right)S + R + T$$

$$\Pi_{N-1} = (N-1)T$$

When $N = 10$, $\Pi_1=13.048$ and $\Pi_{N-1} = 18.432$ ms. In the ideal case, the service time at each disk would simply be $\frac{k}{D}(\Pi_1 + \Pi_{N-1})$. The total time would then be $\frac{k}{D}(\Pi_1 + \Pi_{N-1})1000$ for this ideal case, since each run contains exactly 1000 blocks.

If the prefetches are synchronized, then any variance in the rotational latency would cause $(D-1)$ disks to wait for the slowest disk to complete its requests. As a result, the total time for synchronized prefetches should be higher than the corresponding time for unsynchronized prefetches. The expected total time of 15.74 seconds corresponds well with the measured unsynchronized time of 16.09 seconds.
and the synchronized time of 16.93 seconds. Since each disk must fetch blocks from multiple runs, the additional concurrency introduced by fetching \(N\) blocks is negligible or non-existent.

**Memory Requirements: “All Disks, All Runs”**

Although the seek component differs significantly between the \(k = 5\) and \(k = 25\) cases, the memory requirements for both cases need not be analyzed separately. The Markov model discussed previously in Chapter 3 describes the cache size \(C\) required to support a given success ratio \(\varphi\) when \(N = 1\). When \(N > 1\) blocks are fetched from each run, however, the Markov model does not provide a solution.

The analysis from Chapter 3, however, assumes that all fetches complete together; i.e., that the prefetches are synchronized. The simulation results, however, do not show any deviation in \(\varphi\) between synchronized and unsynchronized prefetches for the same value of \(C\). The only difference which can be found appears in the total execution times.

Figure 4.6 shows the cache requirements as \(N\) increases from 1 to 10. The cache size for the \(N = 1\) case grows as predicted by the analysis from Chapter 3. When \(N\) increases from 1 to 10, only \(\sim 250\) additional cache blocks are required to support the same success ratio. Since these additional 250 blocks represent only a slight increase from the cache size required for \(N = 1\), supporting this larger values of \(N\) is reasonable.

**4.7 Summary**

As shown for the single-disk case, fetching one block from each run reduced the average seek distance, resulting in a noticeable savings in the total execution time. Fetching
Figure 4.6: All Disks All Runs: The effect of the cache size on the success ratio. 25 runs, 5 disks.

$N$ blocks also showed a significant reduction in total execution time, since the seek time and rotational latency was amortized over $N$ blocks. With an infinitely fast CPU, unsynchronized and synchronized prefetches exhibited no difference in total execution time. When a finite-speed CPU was introduced, however, the difference between these two prefetching strategies became noticeable.

Figure 4.7 shows the performance of the $D$-disk prefetching strategies when a finite-speed CPU is used with the $D$-disk case. The results are shown for $N = 10$, $k = 25$, and $D = 5$. As the time to merge one block decreases, the synchronized time for a strategy should begin to approach the unsynchronized time. Since the computational delay does not overlap with any I/O in the synchronized cases, the total time should decrease linearly as the time to merge one block decreases. With
unsynchronized prefetches, however, CPU activity and I/O can overlap. As the time to merge one block approaches zero, the total time will gradually approach the time taken by an infinitely fast CPU.

Figure 4.8 shows a comparison of the $D$-disk prefetching strategies for $k = 5$ and $k = 25$. For all strategies, $N$ is varied to study the effect on the total execution time. All strategies have a sufficiently large cache to ensure a success ratio of 1.

As $N$ increases, the average time to fetch one block approaches $T$, the transfer time. Therefore, the marginal returns in the total execution time diminish as $N$ increases. When $k = D = 5$, only two prefetching strategies exist. The "all disks one run" is the better of the two, because it exhibits more overlap than the "demand run only" strategy.

When $k = 25$ and $D = 5$, the "all disks one run" and "all disks all runs" strategies outperform the other two strategies, since they maintain a greater degree of concurrency. Since the "all disks all runs" and "all disks one run" exhibit similar concurrency, their performance becomes similar as $N$ increases; the greater amortization of seek time and rotational latency reduces the impact of the "all disks all runs" strategy's lower average seek distance. Even though the total time for all strategies is reduced, the total time taken by the "demand run only" strategy is more than twice that taken by the "all disks all runs" strategy.

The graph in Figure 4.9 shows the effect of cache size on total execution time\textsuperscript{23} for both prefetching strategies when $k = 5$, $D = 5$. As $N$ increases from 1 to 10, the total time of each strategy drops, but the "all disks one run" strategy requires a slightly larger cache size than the "demand run only" strategy. As $N$ increases further, the "all disks one run" strategy still performs better than the "demand run

\textsuperscript{23} Unsynchronized prefetching is applied to all strategies.
Figure 4.7: The effect of a finite-speed CPU on the D-disk prefetching strategies. In all cases, the cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1.
Figure 4.8: The performance of the $D$-disk strategies when fetching $N$ blocks. The cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1.
only" strategy. At these large values of \( N \), the "demand run only" strategy shows a sufficient amount of overlap, but this overlap is still less than that shown by "all disks one run" strategy.

Figures 4.10 and 4.11 show the effect of cache size on the \( D \)-disk prefetching strategies when \( k = 25 \). When \( N = 1 \), the "all disks all runs" strategy clearly outperforms the others, but as \( N \) increases, the execution times drop for all strategies. When \( N = 10 \), the "all disks all runs" and "all disks one run" strategies are still the best performers; each requires less than half of the total execution time of the other two strategies. The results for "all disks one run" are quite interesting, since it performs nearly as well as "all disks all runs" but requires a smaller cache size.

The prefetching strategies discussed can provide a substantial reduction in the total execution time if adequate cache memory is available. As shown by the analysis and simulation results, a tradeoff can be made between the cache size and the total execution time. For a data size of 5 runs, the "all disks one run" strategy can easily support fetching \( N = 10 \) blocks with a 200-block cache. This cache size represents only \( \frac{200}{5000} = 4\% \) of the data size. For a data size of 25 runs, the "all disks all runs" strategy can support fetching \( N = 10 \) blocks with a 1800-block cache—only 7.2\% of the data size of 25000 blocks. As shown by the performance data, the total execution time drops substantially as more cache memory is available. Once a certain cache size is reached for each strategy, the total execution time can not be decreased further by increasing the size of the cache.
Figure 4.9: Comparison of prefetching strategies for $D = 5$ disks, $k = 5$ runs. The cache is large enough to ensure that all prefetches are successful, i.e., the success ratio is 1.
Figure 4.10: Total time as a function of cache size. $N = 1$ and $N = 5$. $k = 25$ runs.
Figure 4.11: Total time as a function of cache size. $N = 10$. $k = 25$ runs.
Chapter 5

Conclusions

The previous chapters presented several different prefetching strategies which introduced concurrency and reduced the average I/O time to fetch one block. In the first prefetching strategy studied, the next block from each of $D$ I/O streams was fetched on each I/O operation. For this strategy, a Markov model was developed and analyzed. By categorizing the states and determining the steady-state probabilities, the average parallelism was computed as a function of the cache size and the number of I/O streams, $D$. The analysis showed that for asymptotically large data sizes, the cache size required to support a desired I/O parallelism was a function of $D$.

This Markov model showed the cache requirements for the following strategies: "all disks all runs", "all disks one run" with $k = D$, and single-disk "all runs". From this original Markov model, several related strategies emerged. These strategies attempted to address the constraints of limited prefetching from the $D$ disks. For each strategy, the total I/O time was analyzed, and the cache performance was investigated through simulation. The results indicated that a trade-off exists between total execution time and the cache size. For each strategy, a certain cache size was required to ensure that all prefetches were successful. The results showed that a cache size which is 4%–7% of the total data size was sufficient to support parallel fetches from each of the $D$ disks.

The simulation results show some promising strategies which evolved from the Markov model. Using a similar approach, an analytic model could be developed for
the remaining strategies. To analyze any of these strategies, however, an entirely new Markov model must be developed. The analysis from the current model relies upon the symmetry of the states and corresponding state transitions. The derivation of the steady-state probabilities hinges upon the definition of the state space.

These simulation results give rise to new research problems:

- How do the cache requirements grow for the “demand disk all runs” and “all disks one run” strategies?

- How does fetching $N$ blocks from each run affect the cache requirements? The Markov model showed the cache behavior when one block from each run was fetched on an I/O operation. Originally, the cache size required to maintain the same success ratio was conjectured to grow by a factor of $kN$ or $(kN)^2$ when $N$ blocks were fetched from each of the $k$ runs. For the range of $N$ simulated, however, the results indicate that an additional $kN$ blocks rather than a factor of $kN$ blocks may be required to maintain the same success ratio. Developing an analytic model to determine whether this behavior continues for all $N$ would be worthwhile.

- How can the effects of a finite-speed CPU be incorporated into such an analytic model?
Bibliography


