Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

MULTI-DISK SUBSYSTEM ORGANIZATIONS FOR VERY LARGE DATABASES

by
Zhiyi Huang

This thesis investigates efficient mappings of very large databases with non-uniform access to its data to a multi-disk subsystem.

Two algorithms are developed to distribute the database across multiple disks, possibly with replication, in order to minimize latency and maximize throughput. These algorithms are compared with respect to the amount of replication overhead incurred to achieve desired throughput.

A simulator is developed to simulate these two mapping algorithms and investigate the efficiency of these two mappings.

MULTI-DISK SUBSYSTEM ORGANIZATIONS FOR VERY LARGE DATABASES

by
Zhiyi Huang

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

May 1994

APPROVAL PAGE

MULTI-DISK SUBSYSTEM ORGANIZATIONS FOR VERY LARGE DATABASES

Zhiyi Huang

Dr. M. Palis, Thesis Advisor Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. J. Carpinelli, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. E. Hou, Committee Member Date
Assistant Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Zhiyi Huang

Degree: Master of Science in Electrical Engineering

Date: May 1994

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, USA, 1994
- Bachelor of Science in Electrical Engineering,
  Beijing University of Aeronautics and Astronautics, Beijing, P.R. China, 1991

Major: Electrical Engineering

This thesis is dedicated to
my family

ACKNOWLEDGMENT

The author would like to express her sincere gratitude to her advisor, Professor Michael A. Palis, for his guidance, support, kindness, encouragement and friendship throughout the process of producing this thesis.

Thanks for Professor E. Hou and Professor J. Carpinelli for serving as members of the thesis committee.

TABLE OF CONTENTS

Chapter | Page

1 INTRODUCTION . . . 1
  1.1 Motivation . . . 1
  1.2 Objectives . . . 2
  1.3 Previous Work . . . 2
2 MULTI-DISK SYSTEM AND DATABASE MODELS . . . 6
  2.1 Multi-Disk System Model . . . 6
    2.1.1 The Model . . . 6
    2.1.2 Assumptions and Parameters . . . 8
  2.2 Database Model . . . 9
  2.3 Performance Metrics . . . 9
3 MAPPING ALGORITHMS . . . 11
  3.1 Setting up Database . . . 11
  3.2 Algorithm One . . . 14
  3.3 Algorithm Two . . . 17
4 THE SIMULATOR . . . 21
  4.1 Motivation of Simulation . . . 21
    4.1.1 Advantages of Simulation . . . 21
    4.1.2 Discrete Event Simulation . . . 21
  4.2 Creating the Database . . . 22
    4.2.1 Gaussian Distribution . . . 22
    4.2.2 Exponential Distribution . . . 24
  4.3 Mapping the Database to the Multi-Disk Subsystem . . . 25
  4.4 Simulating the Multi-Disk Subsystem . . . 25
5 EXPERIMENTAL RESULTS . . . 27

Chapter | Page

5.1 Observations 28

6 CONCLUSIONS AND FURTHER STUDY 32

6.1 Conclusions 32

6.2 Further Study 32

APPENDIX A FIGURES FOR CHAPTER 5 34

APPENDIX B SOURCE CODES 48

REFERENCES 67

LIST OF FIGURES

Figure		Page
1.1	Traditional System	3
1.2	Synchronized System	4
1.3	Declustered System	5
2.1	The Multi-Disk Subsystem Model	6
3.1	One Gaussian Curve	11
3.2	Two Gaussian Mode with Intersection	12
3.3	Density function of Exponential distribution	13
3.4	An Example of Algorithm One	16
3.5	An Example for Algorithm Two	19
4.1	Density Function of Gaussian Distribution	23
4.2	Density Function of Uniform Distribution	24
5.1	Generating Results	29
A.1	throughput::100 classes, Gaussian	35
A.2	throughput::50 classes, Gaussian	36
A.3	throughput::200 classes, Gaussian	37
A.4	throughput::64 disks, Gaussian	38
A.5	throughput::128 disks, Gaussian	39
A.6	throughput::64 disks, 100 classes	40
A.7	throughput::128 disks, 100 classes	41
A.8	throughput::Two Different Distributions	42
A.9	latency::100 classes, Algorithm Two	43
A.10	latency::50 classes, Algorithm Two	44
A.11	latency::200 classes, Algorithm Two	45
A.12	latency::64 disks, Algorithm Two	46

Figure	Page
A.13 latency::128 disks, Algorithm Two	47

CHAPTER 1

INTRODUCTION

1.1 Motivation

In the last few years, the performance of processors has been growing steadily, and with multiprocessor organizations, the processing power of a computer system has been increasing at a rapid pace. However, the I/O performance has not kept pace with the gains in processing power. Currently, there exists a huge disparity between the I/O subsystem's performance and the processing power.

With such differences in processing and I/O capacities, eventually the problem solving speed will be determined by how fast the I/O can be done. If the I/O is done serially, even if we solve the problem considerably faster, the speed of the solution will depend on the serial I/O bottleneck. So, it is essential that some kind of improvements be made in I/O performance to avoid these bottlenecks.

The importance of balancing the I/O bandwidth and the computational power has been pointed out by Kung, [1] where it is shown that for some of the applications, the I/O problem cannot be alleviated by adding more memory at the processors.

For such applications, it is essential that the I/O power grows as fast as the processing power of the system. Unless the I/O can also be speeded up by the same factor as that of the processing, we cannot truly obtain linear speedups. Hence, to improve the overall performance of a system, it is essential that the I/O subsystem performance also has to be improved.

The time required for searching a database is, in the limit, proportional to the size of the database. When databases grow and become very large, a single I/O subsystem consisting of few disks are no longer sufficient.

As the number of users increases, the chance that contention occurs (e.g. when two or more users trying to access the same data at the same time) also increases.

A multi-disk subsystem has the potential of serving a request for data on a number of disks concurrently. The number of disks that will take part in the execution of a request depends on the location of the data to be accessed and the amount of data to be transferred.

There have been a number of proposals to organize a number of disks together to form a powerful disk system. Some of the previous studies in improving I/O performance include [2]-[5]. I will briefly describe these approaches in section 1.3.

1.2 Objectives

The objective of this thesis is to investigate efficient mappings of very large databases with non-uniform access to its data to a multi-disk subsystem.

Two algorithms are developed to distribute the database across multiple disks, possibly with replication, in order to minimize latency and maximize throughput. These two algorithms are compared with respect to the amount of replication overhead incurred to achieve desired throughput.

To access the performance of the algorithms, a simulator is developed to simulate these two mapping algorithms and investigate the efficiency of these two mappings.

1.3 Previous Work

A system where all the disks are organized as independent units and where each file is located only on one disk is called a traditional system. See Fig.1.1.

In this figure, f stands for a file, as do as g and h. Each file is stored in its entirety on one disk.

This traditional system is simple to implement and has a reasonably good performance on single block requests. If the files are not uniformly distributed, a large number of requests may arrive at a single disk. Hence, this possibility suggests that a traditional system under unbalanced conditions could yield poor performance. In a traditional system, the response time of a request is improved by decreasing the waiting time in the queue when the number of disks in the system is increased. The service time remains the same. [3]

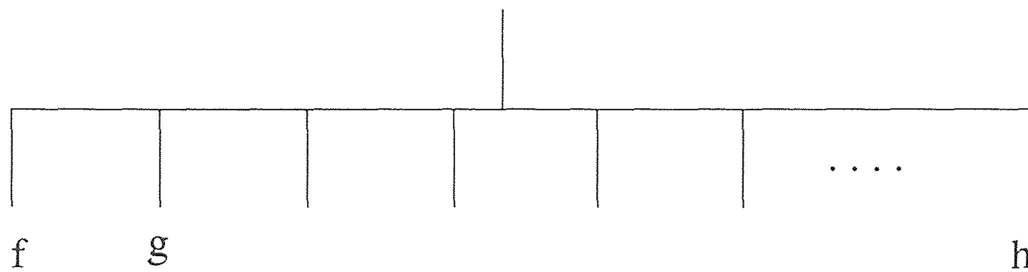

Figure 1.1 Traditional System

Kim [2] shows that disk synchronization leads to efficient transfer of large blocks of data when compared to a traditional system.

Disk synchronization is useful when file read time is the dominant part of the I/O service time. In this organization, a file is interleaved bytewise among the disks. The disks are synchronized such that each disk head is positioned at the same place. All the disks together function as a single large disk with m times the transfer rate and m times the capacity of a single disk, where the seek time (the time spent in accessing the right track of the disk) and latency time (the time taken for the required block to come under the read/write head) remain the same. See Fig.1.2.

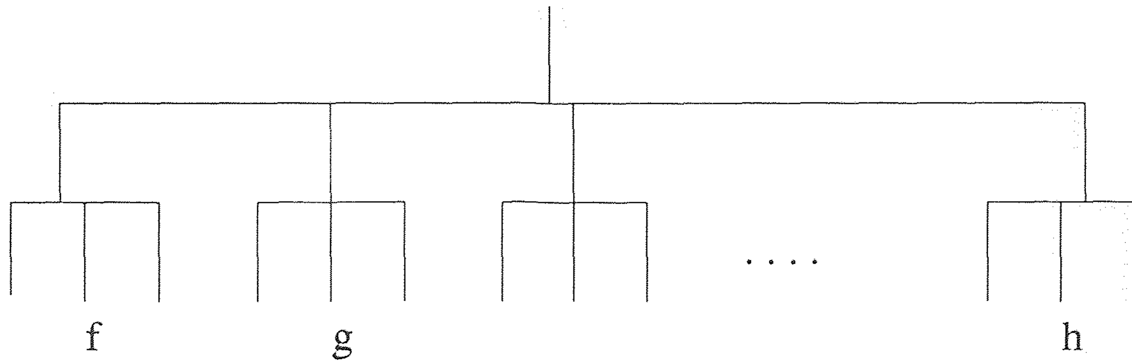

Figure 1.2 Synchronized System

In this figure, three disks are synchronized as a single large disk unit, each file is located on one disk unit.

When the transfer time dominates the I/O service time, this organization yields high performance. Since all the disks are coupled together, each disk sees the same request rate and hence a balanced load on all the disks. The utilization of each disk is higher since each request has to be processed by all disks. Hence, this organization is useful when large files need to be transferred.

Studies by Livny *et al.* [5] and Salem em et al. [4] report that a system with data declustering is better than a traditional system.

Disk striping or data declustering calls for blockwise interleaving of the file such that a number of blocks of a file can be read or written in parallel on different disks. One way of interleaving of the file is such that if block i is on disk j, then $i+1$ is on $j+1$ and so on. See Fig.1.3.

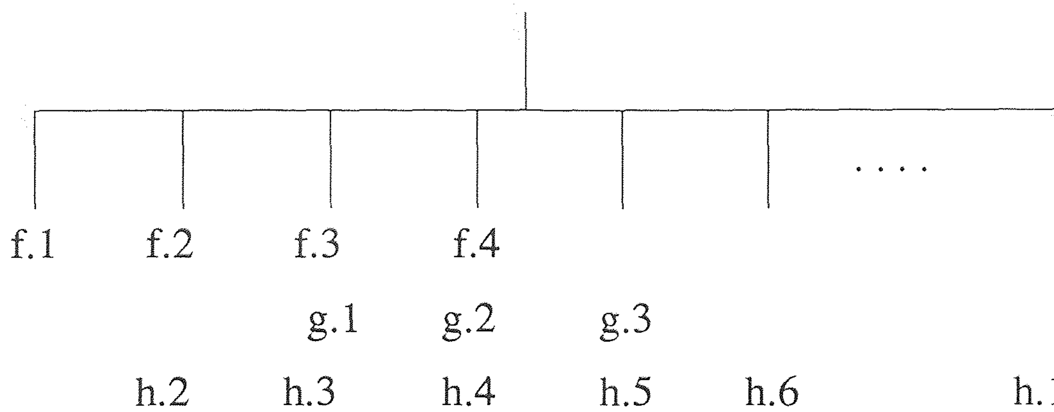

Figure 1.3 Declustered System

In this figure, file f has 4 blocks, they are interleaved onto 4 disks. Blocks of file g are interleaved onto 3 disks. Blocks of file h are interleaved onto 6 disks.

If the I/O request is for a single block, then the request is queued in the disk containing that block. If a multiple block request arrives at the disk system, then that request is broken up into a number of single-block requests concurrently.

By interleaving the data, we can keep the disk system balanced with respect to request rates. In this organization, the seek time and latency penalties are paid for each block of transfer and hence the service time has effectively increased.

In this thesis, a traditional system with some modifications is used. A file is allowed to have several copies on disks.

CHAPTER 2

MULTI-DISK SYSTEM AND DATABASE MODELS

2.1 Multi-Disk System Model

2.1.1 The Model

In this thesis, the multi-disk subsystem is modeled as m single queue, single-disk server queuing systems and a CPU which serves as a dispatcher for user requests, as shown in Fig.2.1. Each disk server has its own queue, and the system has a unique incoming request queue to buffer all the requests generated by users from different terminals. The incoming request queue has infinite size: it can store all the requests and no requests are lost after they are generated. Several requests may be served concurrently on a number of disks if they access different disks.

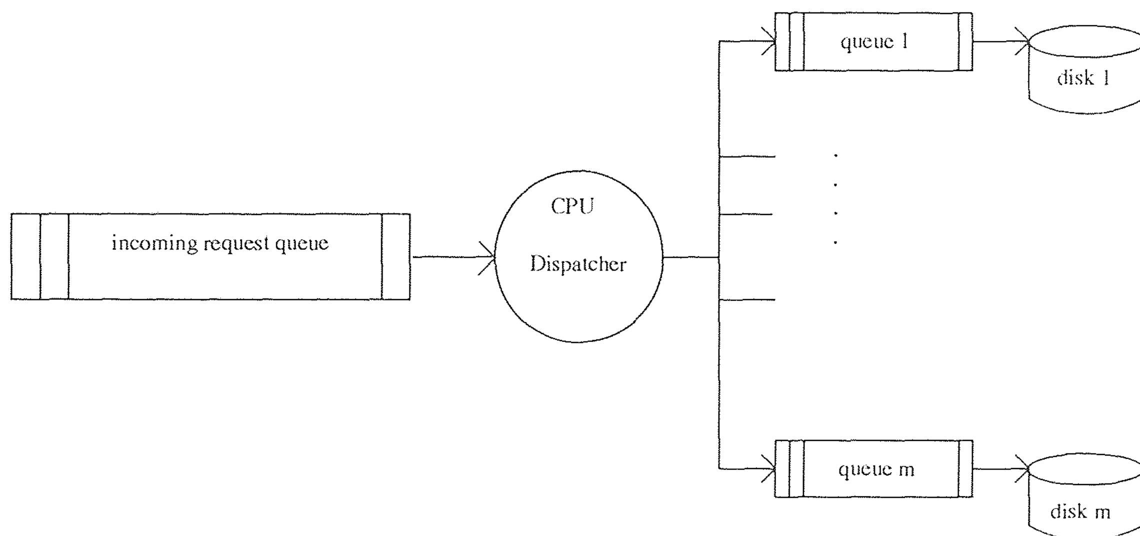

Figure 2.1 The Multi-Disk Subsystem Model

The reason why a queue is used is the existence of contention. Contention arises when two or more requests are made concurrently for a resource that cannot by shared.

In the multi-disk subsystem, this happens when:

1. Two or more requests want to access the same disk at the same time; or
2. One or more requests want to access a disk which is currently being accessed.

The CPU is used as a dispatcher for incoming requests. It knows what is stored on each disk, and keeps track of the queue size of each disk. In this thesis, the CPU is not modeled, and is assumed to take zero time to dispatch a request. The performance of the disk subsystem is the only thing modeled.

The CPU performs in the following steps:

while (the incoming request queue is not empty) do

Fetch a request R from the head of the incoming request queue;

Find disks which contain the data that R wants to access;

Among disks containing R, find a disk D which has the smallest queue size;

If (queue of D is full)

Do not dispatch R, leave it unchanged in the incoming request queue;

Else, remove R from CPU's queue and insert it into D's queue;

end.

Each disk server performs in the following steps:

while (the queue of disk is not empty) do

Fetch a request from the head of its own queue;

Serve that request;

Return the size of its queue to CPU;

end.

2.1.2 Assumptions and Parameters

The assumptions about the multi-disk subsystem model are summarized as follows:

- The disk service time for one request is a fixed constant.
- One time unit is defined as the disk service time for one request.
- The incoming request queue size is infinite.
- The CPU's dispatching time is zero, that is, CPU can dispatch all the queries to disks if the corresponding disks are not full, this costs zero time. In case a disk is full, queries wishing to access to this disk have to stay at the incoming request queue.
- The incoming requests follow a Poisson distribution.

The parameters for the multi-disk subsystem model are:

- *no_disks*: the number of disks used in the disk subsystem;
- *disk_size*[i]: length of disk i's queue;
- *disk*[i]: linked list of requests dispatched to disk i;
- *maxqueue_size*: the maximum queue size of each disk;
- *request*: linked list of requests in the incoming request queue;
- *request_size*: the number of outstanding requests in the incoming request queue;

2.2 Database Model

A database is a collection of files or data. It is stored on some media, such as disks. These files or data are public resource: every body can access them.

Each file or data has a frequency of access. Some files or data may be interesting to many people, so they will have a high frequency of access. Some files or data may be interesting to few people, so they will have a low frequency of access. The frequency distribution of file access may follow some rules.

In this thesis, it is assumed that the frequency distribution of access is Gaussian or Exponential. These files or data are grouped into N classes. The total frequency of access of a class is the sum of the frequency of all the components in a class. Every class has the same size, that is, the same number of components.

The parameters for this database model are:

- $class_freq[c]$: access frequency of class c;
- $no_classes$: total number of classes used in the simulation;
- $class_list[i]$: linked list of classes assigned to disk i;
- $class_freq[c]_{disk}$: access frequency of class c on disk;

$$class_freq[c]_{disk} = \frac{class_freq[c]}{\#\ of\ \ disks\ \ storing\ \ class[c]} \tag{2.1}$$

- $disk_freq[i]$: the sum of $class_freq[c]_{disk}$ of classes assigned to disk i;
- $total_class[i]$: number of classes assigned to disk i.

2.3 Performance Metrics

To investigate the performance of multi-disk and database models, some definitions are needed:

- Throughput:

  The number of completed requests per unit time interval.

- Latency:

  In this multi-disk subsystem, this is defined as the time a request stayed in the system. That is, the time this request leaves the system upon completion of service minus the time this request enters the system and waits in the queue.

- Average latency:

  The sum of latencies of all requests served divided by the total number of requests served.

$$average_latency = \frac{sum\ \ of\ \ latencies\ \ of\ \ each\ \ request}{\#\ \ of\ \ requests} \tag{2.2}$$

- overhead:

  In order to increase throughput, it might be needed to replicate some frequently accessed files. This will increase the size of storage. Overhead is defined as:

$$overhead = \frac{\#\ \ of\ \ files\ \ actually\ \ stored\ \ on\ \ disks - size\ \ of\ \ database}{size\ \ of\ \ database} \tag{2.3}$$

where the size of database is defined as:

number of different files stored on the disk without replication.

CHAPTER 3

MAPPING ALGORITHMS

The goal of mapping the database onto disks is to maximize the throughput given a certain maximum amount of allowable overhead.

3.1 Setting up Database

We assume the following frequency distributions for the database:

First, it is assumed that the frequency distribution of information access is uni-modal Gaussian with μ=400 and σ=100. See Fig.3.1.

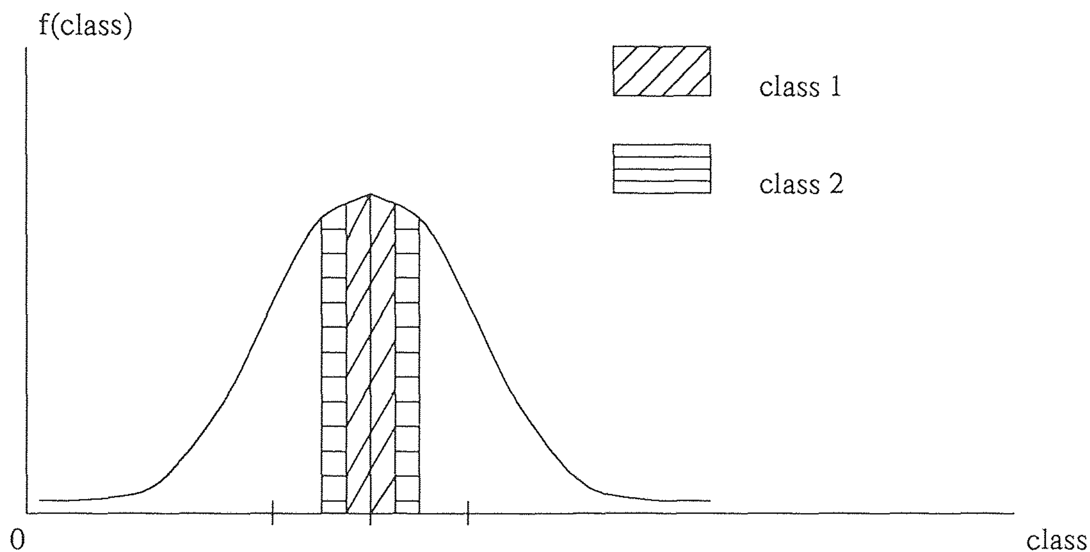

Figure 3.1 One Gaussian Curve

According to Zipf's Law: 80% of access is to 20% of the data, data access is not uniformly distributed. Many database files follow a Gaussian distribution. For example:

- Organization of UNIX file system;
- Electronic office filling;
- computerized libraries;
- Electronic storage and retrieval of articles from newspapers.

Second, it is assumed that the frequency distribution of information access is bi-modal Gaussian, one is with μ=400 and σ=100, another is with μ=1200 and σ=50. See Fig.3.2.

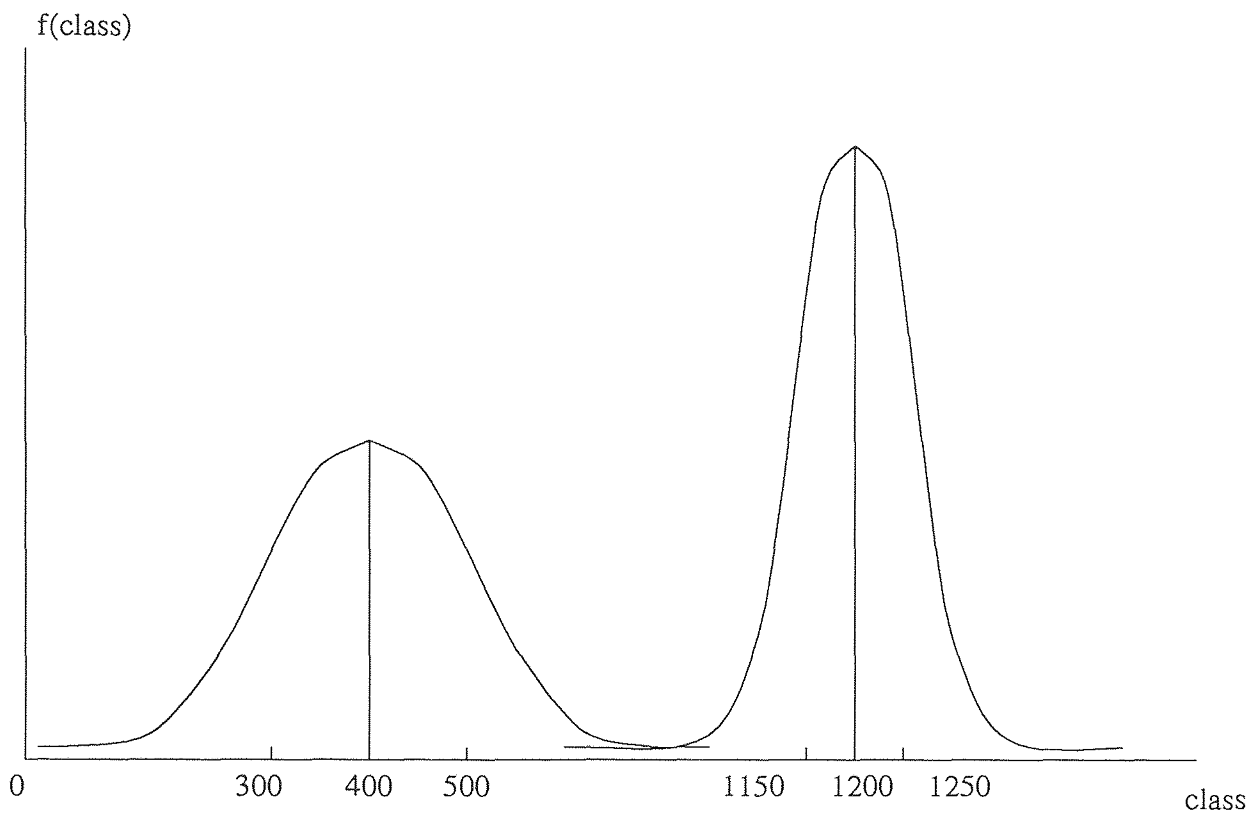

Figure 3.2 Two Gaussian Mode with Intersection

A databases may have two or more sub-databases which follow Gaussian distribution. For example, a *New York Times* database may have *sport* and *estate* sub-databases, each of which follows a Gaussian distribution.

Third, it is assumed that the frequency distribution of information access is Exponential with σ=15. See Fig.3.3.

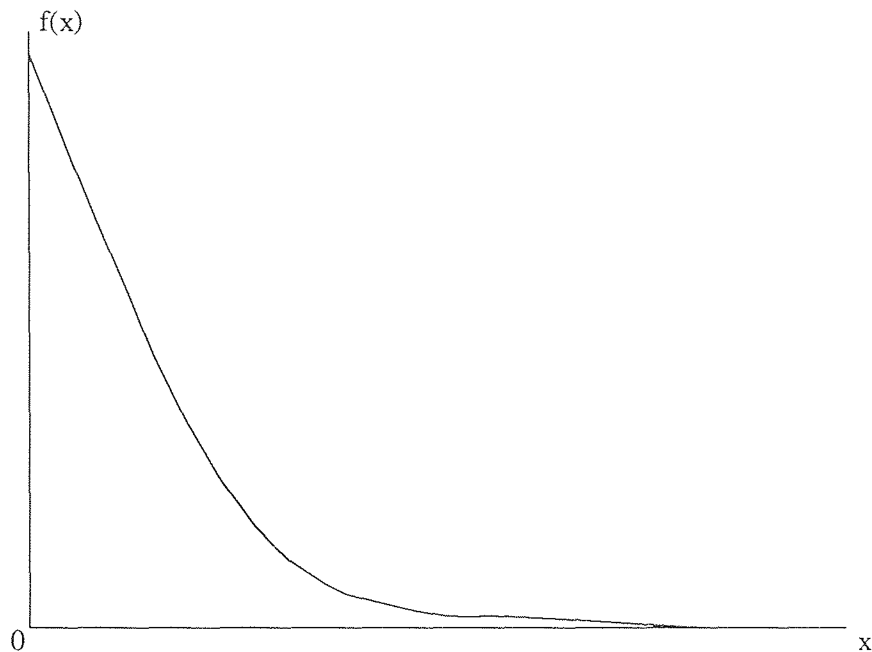

Figure 3.3 Density function of Exponential distribution

For example, a temporal database which stores data time-wise for future usages.

For convenience, files will be used to represent information that is stored in a database from now on.

Files are grouped into N classes. The way of grouping files is to evenly divide the shape under the density function into N sub-areas, as shown in Fig.3.1.

Now I have N classes. *no_classes* will be used to represent the total number of classes being grouped, and *no_disks* will be used to represent the total number of disks being used in the disk subsystem.

We now describe two algorithms for mapping the database classes to the disks. Both algorithms allow replication of frequently accessed classes to maximize throughput and minimize latency, subject to the constraint that the overhead incurred is no more than a maximum allowable overhead.

3.2 Algorithm One

/*** Mapping Step ***/

for each disk, initialize its disk_freq to zero

let $s = max\{no_classes, no_disks\}$

while $(s > 0)$ *do*

Find the disk which has the minimum disk_freq, called d_{min};

Find the class which has the maximum class_freq, called c_{max};

Assign c_{max} *to the class_list of* d_{min};

Recalculate the disk_freq and total_class of d_{min};

$s--$;

end.

The first three rows of Fig.3.4 illustrate the Mapping Step for a database $c1$ to $c6$ and four disks.

The Mapping Step maps the database to disks without replication (i.e., the overhead is zero). However, the disk access frequencies may be severely unbalanced. By replicating frequently accessed class among several disks, the difference among disk access frequencies can be minimized, thereby increasing throughput.

Before we describe the Replication Step, we first introduce the concept of a disk equivalence class:

Two disks belong to the same equivalence class if and only if they contain the same set of database classes.

For example, in row 3 of Fig.3.4 (after the Mapping Step), each disk belongs to its own equivalence class. However, in row 4, disks 2 and 3 belong to the same equivalence class. This results when $c2$ is replicated in disk 3 and $c3$ and $c6$ is replicated in disk 2.

In the Replication Step discussed below, replication is done among disk equivalwnce classes as opposed to two individual disks.

/*** Replication Step ***/

overhead = 0;

while (overhead < overhead_allowed), do

Find the disk equivalence class D_x with the maximum disk_freq;

Find the disk equivalence class D_y with the minimum disk_freq;

Merge D_x and D_y into a single equivalence class as follows:

(a) copy into D_x all database classes that are in D_y but not in D_x;

(a) copy into D_y all database classes that are in D_x but not in D_y;

Recalculate disk_freq of each disk;

Recalculate overhead;

end.

We illustrate the Replication Step just described using the example shown in Fig.3.4. Step 3 shows the contents of the disks after the Mapping Step.

Steps 4-6 of Fig.3.4 correspond to the first three iterations of the Replication Step.

After step 3, we find that disk 2 has the minimum *disk_freq*, and disk 3 has the maximum *disk_freq*. They are each in their own equivalence classes.

Hence, the classes residing in these disks are replicated on both disks, i.e., class $c2$ is copied into disk 3 and classes $c3$ and $c6$ are copied into disk 2.

The result of this replication is to merge the disk equivalence classes {2} and {3} into a single equivalence class {2, 3}. The new *disk_freq* of this equivalence class is 24.

Classes : c1 c2 c3 c4 c5 c6
class_freq : 26 20 18 14 12 10
total_freq[i] : frequency of disk[i] being accessed

disk 1	disk 2	disk 3	disk 4	steps	overhead
c1 26	c2 20	c3 18	c4 (classes) 14 (total_freq) min	1	——
c1 26	c2 20	c3 18 min	c4, c5 26	2	——
c1 26	c2 20 min	c3, c6 28 max	c4, c5 26	3	0
c1 26 max	c2, c3, c6 24 min	c2, c3, c6 24	c4, c5 26	4	0.5
c1, c2, c3, c6 74/3 min	c1 c2, c3 ,c6 74/3	c1, c2, c3, c6 74/3	c4, c5 26 max	5	1.33
c1, c2, c3, c4, c5, c6	c1, c2, c3, c4, c5, c6	c1, c2 , c3, c4, c5, c6	c1, c2, c3, c4, c5, c6	6	3

Figure 3.4 An Example of Algorithm One

After step 4, disk 1 has the maximum *disk_freq*. The equivalence class consisting of disks 2 and 3 has the minimum *disk_freq*. Thus, the two equivalence classes are merged into a single equivalence class {1, 2, 3}. This is achieved by copying $c1$ into disks 2 and 3, and copying $c2$, $c3$ and $c6$ into disk 1.

Step 5 is similar to step 4.

Algorithm One terminates when the current overhead exceeds the maximum allowable overhead, or when each class has a copy in every disk. For example, if the maximum allowable overhead is 1.8, Algorithm One returns the database map shown in step 5 of Fig.3.4.

The replication step of Algorithm One could potentially replicate many classes during a single iteration. Consequently, the increase in overhead per iteration could be very large. Indeed, there is no way to control overhead increase per iteration. (refer to step 5 and 6 in Fig.3.4.)

What a customer will be interested in is the overhead he paid and the throughput he gained. Overhead is critical. It needs to be under control.

Algorithm two presented below can overcome this problem.

3.3 Algorithm Two

/*** Mapping Step ***/

This is the same as in algorithm one.

/*** Replication Step ***/

overhead = 0;

while (overhead < *overhead_allowed) do*

find disk D_x *with the maximum disk_freq;*

find disk D_y *with the minimum disk_freq;*

in D_x, *find the database class* c_x *with the maximum class_freq that*
is not in D_y;

in D_y, *find the database class* c_y *with the maximum class_freq that*
is not in D_x;

copy c_x *to* D_y;

copy c_y *to* D_x;

recalculate disk_freq;

recalculate overhead;

end.

Fig.3.5 illustrates Algorithm Two using the same instance as that given in fig.3.4.

The first three steps are the same as Algorithm One.

Using the terminology of Algorithm Two, we see that, after step 3, D_x is disk 3 and D_y is disk 2. Moreover, c_x is class $c3$ (i.e., the class in D_x but not in D_y which has the maximum frequency of access). Similarly, c_y is class $c2$ (i.e., the class in D_y but not in D_x which has the maximum frequency of access). Thus, $c3$ is copied into disk 2 and $c2$ is copied into disk 3.

After step 4, we see that D_x is again disk 3 and D_y is again disk 2. Moreover, c_x is class $c6$ (note that c_x cannot be $c2$ or $c6$ since both have copies in D_y, or disk 2). Now, c_y does not exist because all classes in D_y (= disk 2) have copies in D_x (= disk 3). Therefore, $c6$ is copied into disk 2 and no new class is copied into disk 3.

As in Algorithm One, Algorithm Two terminates when the current overhead exceeds the maximum allowable overhead. For example, if the maximum allowable overhead is 1.8, Algorithm Two returns the database map shown in step 10 of Fig.3.5.

Classes : c1 c2 c3 c4 c5 c6

class_freq : 26 20 18 14 12 10

total_freq[i] : frequency of disk[i] being accessed

disk 1	disk 2	disk 3	disk 4	steps	overhead
c1 26	c2 20	c3 18	c4 (classes) 14 (total_freq) min	1	——
c1 26	c2 20	c3 18 min	c4, c5 26	2	——
c1 26	c2 20 min	c3, c6 28 max	c4, c5 26	3	0
c1 26	c2, c3 19 min	c2, c3, c6 29 max	c4, c5 26	4	0.33
c1 26 max	c2, c3, c6 24 min	c2, c3, c6 24	c4, c5 26	5	0.5
c1, c2 19.7 min	c1, c2, c3, c6 33.7 max	c2, c3, c6 20.7	c4, c5 26	6	0.833
c1, c2, c3 25.7	c1, c2, c3, c6 30.7 max	c2, c3, c6 17.7 min	c4, c5 26	7	1
c1, c2, c3 21.3 min	c1, c2, c3, c6 26.3 max	c1, c2, c3, c6 26.3	c4, c5 26	8	1.167
c1, c2, c3, c6 24.7 min	c1, c2, c3, c6 24.7	c1, c2, c3, c6 24.7	c4, c5 26 max	9	1.33
c1, c2, c3, c4, c6 29.5 max	c1, c2, c3, c6 22.5 min	c1, c2, c3, c6 22.5	c1, c4, c5 25.5	10	1.667
c1, c2, c3, c4, c6 27.2 max	c1, c2, c3, c4,c6 27.2	c1, c2, c3, c6 22.5 min	c1, c4, c5 23.2	11	1.833

...

Figure 3.5 An Example for Algorithm Two

In general, Algorithm Two performs more iterations than Algorithm One for the same maximum allowable overhead. The reason is that Algorithm Two replicates at most two classes at a time. Consequently, the increase in overhead per iteration is bounded by a fixed constant. This is not true for Algorithm One.

As we shall see in Chapter 5, the ability of Algorithm Two to control the overhead increase per iteration translates, in general, to a more balanced distribution of disk access frequencies. This in turn, results in higher throughput and lower latency than Algorithm One, for the same amount of overhead.

CHAPTER 4

THE SIMULATOR

In this chapter, it is shown how to simulate the database model and multi-disk subsystem model.

4.1 Motivation of Simulation

4.1.1 Advantages of Simulation

Most complex, real-world systems with stochastic elements are difficult to describe by a mathematical model which can be evaluated analytically. Thus, simulation is often the only method for investigation.

Simulation allows us to estimate the performance of an existing system under some projected set of operating conditions.

Alternative proposed system designs can be compared via simulation to see which best meets a specified requirement.

In a simulation we can maintain much better control over experimental conditions than would generally be possible when experimenting with the system itself.

4.1.2 Discrete Event Simulation

Discrete event simulation is concerned with the modeling of a system as it evolves over time using a representation in which state variables change only at a countable number of points in time. These points in time are the ones at which an event occurs, where an event is defined to be an instantaneous occurrence which may change the state of a system.

Although a discrete event simulation could conceptually be done by hand calculations, the amount of data that must be stored and manipulated for most real-world systems dictates that discrete event simulations be done on a digital computer.

4.2 Creating the Database

In the simulation, several probability distributions are used to represent the access frequencies of the database classes.

Bellow we describe these distributions and the methods used to generate the access frequency for each database class.

4.2.1 Gaussian Distribution

The density function for this distribution is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \qquad -\infty < x < \infty \tag{4.1}$$

Where μ and σ are the mean and standard deviation respectively. The corresponding distribution function is given by

$$F(x) = P(X \leq x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(v-\mu)^2} dv \tag{4.2}$$

In such case we say that the random variable X is *normally distributed* with mean μ and variance σ. See Fig.4.1.

Suppose each file is represented by a unique identification number. I will use z to represent the identification number. Because z is Gaussian distributed, the frequency of access of each file will also have Gaussian distribution.

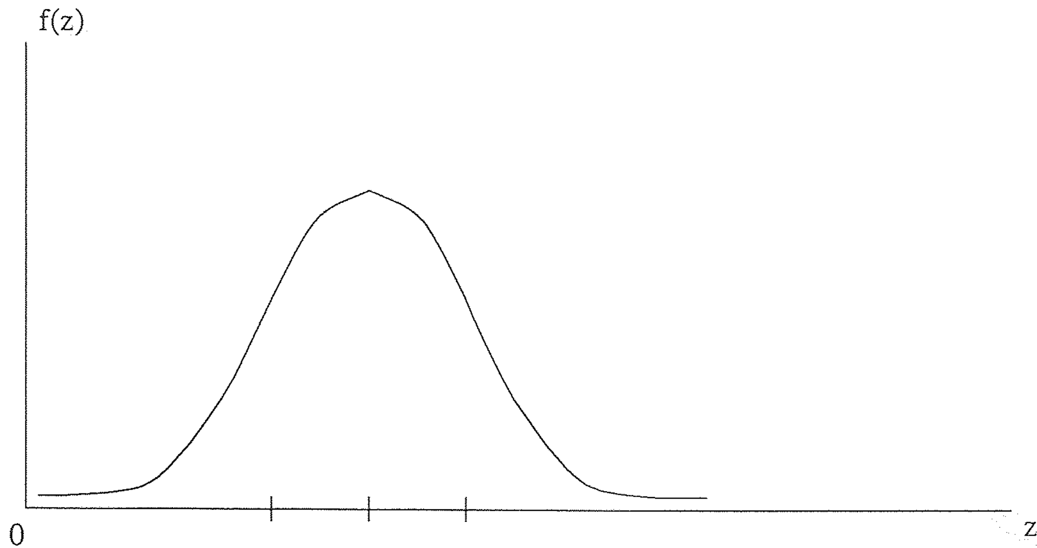

Figure 4.1 Density Function of Gaussian Distribution

In the simulation, a standard Gaussian distributed random variable is generated first. The standard Gaussian distribution has mean μ equal to zero, and variance σ equal to 1. Any other kind of Gaussian distribution can be obtained once the mean μ and variance σ are given according

$$X' = \mu + \sigma X \tag{4.3}$$

where X stands for standard Gaussian distributed random variable, and X' stands for an arbitrary Gaussian distributed random variable with mean μ and variance σ. [6]

In the subsequence, $N(0, 1)$ will be used to represent the standard Gaussian distribution, and U(0,1) to represent the standard Uniform distribution in the interval $[0, 1]$. Fig.4.2 is the density function of the standard Uniform distribution.

One of the early methods for generating $N(0, 1)$ random variable, according to Box and Muller,[13] is evidently still in wide use despite the current availability of

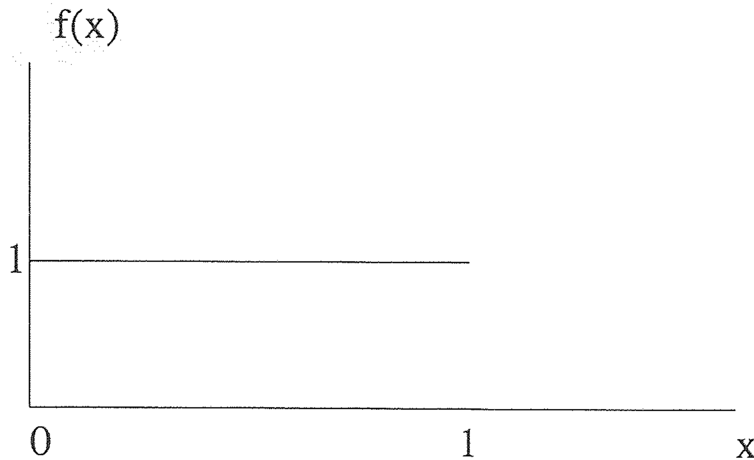

Figure 4.2 Density Function of Uniform Distribution

much faster algorithms. It does have the advantage, however, of maintaining a one-to-one correspondence between the $U(0,1)$ random variables and the $N(0,1)$ random variable produced.

The method simply says:

1) generate U_1 and U_2 as independent uniform distributed random variables,

2) set $X = \sqrt{-2\ln U_1}\cos 2\pi U_2$,

3) X is an independent Gaussian distributed random variables.

Hence, an Gaussian distributed random variable X' can be obtained using formula 4.3.

For the bi-modal case,

1. generate a standard Gaussian distributed random variable X;
2. generate a (0,1) evenly distributed random variable var;
3. if $var > .5$, generate X' using formula 4.3;
4. if $var < .5$, generate X' using formula 4.3.

4.2.2 Exponential Distribution

The density function of an exponential distributed random variable is:

$$f(x) = (1/\sigma)e^{-x/\sigma} \quad x \geq 0 \tag{4.4}$$

The algorithm for generating this kind of random variable is as follows:[6]

1. generate U(0, 1);
2. set $X = -\sigma \ln U(0,1)$ and return.

The files under these three distributions are divided into N classes evenly, where N is a variable. Each class has the same size, this is the input to the mapping algorithms.

4.3 Mapping the Database to the Multi-Disk Subsystem

M disks are used to stored the database. It is assumed that every disk has unlimited capacity to store data, and for each file, each disk has the same service time. (Service time is the total of seek time, latency time and read time.)

Algorithm One and Two are used to map the database to this multi-disk subsystem.

4.4 Simulating the Multi-Disk Subsystem

The CPU is used as a dispatcher in my model. It dispatches requests stored in incoming request queue. It knows what is stored on each disk, and keeps track of the queue size of each disk. In this thesis, the CPU is not modeled, and is assumed to take zero time to dispatch a request. That is, as long as there is a request, CPU will service it using zero time.

Each disk server will service one request using one disk service time if there is at least a request waiting in the disk's queue.

During one disk service time, a number of λ user requests will be generated and dispatched by CPU.

A Poisson distribution is used to simulate the user requests coming in. These requests are buffered in the incoming request queue, waiting to be served by CPU.

Let X be a discrete random variable which can take on the values $0, 1, 2, \ldots$ such that the probability of X is given by

$$f(x) = P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!} \qquad x = 0, 1, 2, \ldots \tag{4.5}$$

where λ is a given positive constant. This distribution is called the *Poisson distribution* and a random variable having this distribution is said to be *Poisson distributed.*

The algorithm for generating Poisson λ random variables is as follows: [6]

1. let $a = e^{-\lambda}, b = 1$, and $i = 0$;
2. generate U_{i+1}, and replace b by bU_{i+1};
3. if $b < a$, set $x = i$, and go to step 2. Otherwise, go to step 4;
4. replace i by $i + 1$, go back to step 2.

CHAPTER 5

EXPERIMENTAL RESULTS

To figure out the influences of the *no_classes*, the *no_disks*, and the frequency distribution of disk access to system *overhead* and *throughput*, the *no_classes* is varied from 50 to 200, and the *no_disk* from 32 to 256. Three kinds of density distribution functions are used: the uni-modal Gaussian with μ equals to 400 and σ equals to 100; the bi-modal Gaussian in which two Gaussian distributions are interconnected to each other, the other one is with μ equals to 1200 and σ equals to 50; and the Exponential with σ equals to 15.

The procedure for generating results is shown in Fig.5.1.

The CHOOSE box chooses a combination of the number of disks, number of classes, and the class frequency distribution. These are the input parameters for the database which will be mapped to the multi-disk subsystem.

ALGORITHM X (X=1 or 2) is used to map the database, possibly with some iterations of replication steps.

SIMULATOR takes user requests from the incoming request queue, and uses the database developed using ALGORITHM X to simulate the multi-disk subsystem, reports results to OBSERVE.

OBSERVE gathers outputs from SIMULATOR.

5.1 Observations

1. For a fixed overhead, Algorithm Two gives higher throughput than Algorithm One. See Fig.A.1 -Fig.A.3 in appendix A.

   Fig.A.1 is the results with 100 classes, and Gaussian distribution. The number of disks is varied from 32 to 128.

   From Fig.A.1, for example, for the case where there are 100 classes and 64 disks, for a given overhead of 0.5, the throughput for Algorithm One is 51 and for Algorithm Two is 60.

   The maximum percentage of different throughput between Algorithm One and Algorithm Two is 15%.

   Similar observations can be held from other values of number of disks and number of classes.

2. For each pair of curves, it can be noticed that the throughput of Algorithm One eventually becomes higher that that for Algorithm Two at a "cross-over point".

   For example, in Fig.A.1, for the case where there are 100 classes and 128 disks, Algorithm One becomes better when overhead is greater or equal to 1.95.

   However, the difference between throughput value beyond the "cross-over point" is very small.

   In the above example, the maximum percentage of different throughput between Algorithm One and Algorithm Two after "cross-over point" is 4.4%.

   Moreover, the maximum possible throughput that can be achieved by Algorithm One is not significantly higher than the throughput at the cross-over point. In the above example, it is 4.5%.

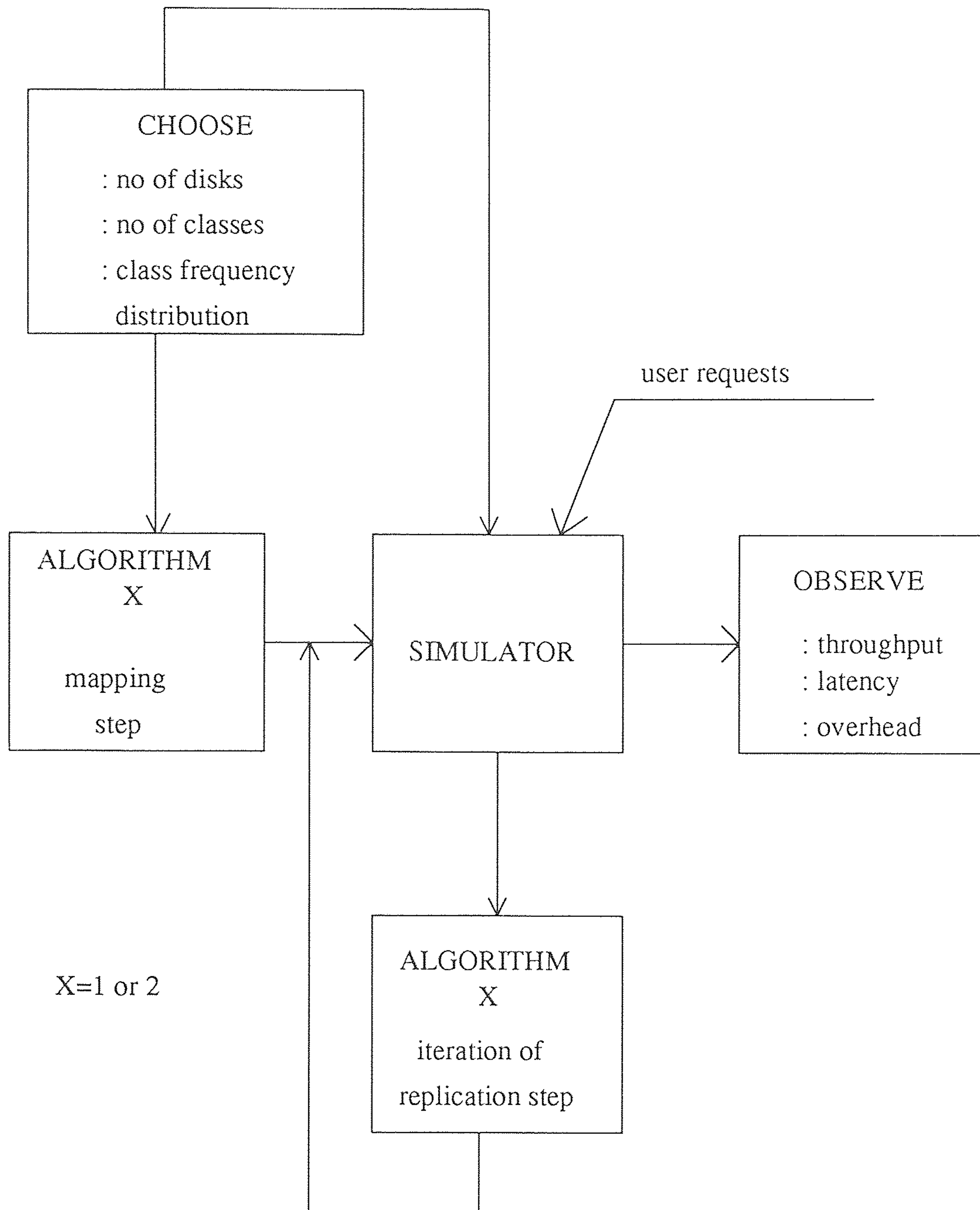

Figure 5.1 Generating Results

3. For a fixed *no_classes* and overhead, throughput increases as the number of disks increasing. See Fig.A.1 - Fig.A.3.

   For example, in Fig.A.1, for a fixed *no_classes* equals to 100, the throughput with 128 disks is much higher than using 64 disks for a certain amount of allowable overhead.

4. the throughput curves using uni-modal and bi-modal Gaussian distributions are very similar. See Fig.A.6 and Fig.A.7 in appendix A.

   Fig.A.6 is the result with 64 disks and 100 classes.

   From Fig.A.6, for example, the two curves are nearly the same.

   Therefore, if someone has a multi-disk subsystem established using Gaussian distribution, when there is another database which has to be stored onto the same multi-disk subsystem, as long as the new database holds Gaussian distribution of file access, we need not to rearrange the older database. We just "add" the new one to the old one.

5. For two totally different file access distributions, the throughput curves are different. See Fig.A.8 in appendix A.

   Fig.A.8 is the results using Exponential distribution and Gaussian distribution separately with 64 disks and 100 classes.

   From this figure, we can see that Algorithm Two is still better than One, although the throughput curves are different from those using Gaussian distribution.

6. For a fixed *no_classes* and overhead, the less disks we have, the lower latency we need to pay. See Fig.A.9 - Fig.A.11 in appendix A.

Fig.A.9 is the results of 100 classes. Number of disks are varied from 32 to 128. Gaussian distribution is used.

From Fig.A.9, for example, the latency using 128 disks is higher than using 64 disks.

CHAPTER 6

CONCLUSIONS AND FURTHER STUDY

6.1 Conclusions

This thesis investigate the efficient mappings of very large databases with non-uniform access to its data to a multi-disk subsystem.

Two algorithms are developed to distribute the database across multiple disks, possibly with replication, in order to balance the frequency of access to disks.

From the results of the previous chapter, it is clear that replication improve both throughput and latency. Because of replication, the storage overhead is increased.

These two algorithms can maximize throughput and latency for a given maximum allowable overhead. However, Algorithm One cannot control the increase in overhead, Algorithm Two can control it.

In general, Algorithm Two gives better throughput and latency for the same amount of allowable overhead.

6.2 Further Study

In this thesis, I focussed on a single processor service system. CPU time is assumed to be negligible. But in practice, we should consider CPU time.

To reduce the CPU time, multiple processors could be used. Each processor may or may not have its own memory. Now given multi-processor and multi-disk subsystem, how should these two subsystems be connected?

There are three different architectures, namely, Shared Everything, Shared Nothing and Shared Disks. In shared Everything, any processor can access any disk

and all memory is shared. In Shared Nothing, neither disk nor memory is shared. In Shared Disks, any processor can access any disk, but each has its own private main memory. These architectures deserve further study.

In the, it is assumed that no index files are used and files are stored on disks directly to simplify the task to find the right starting. Since each file may have more than one level of index file, and each file may have different length, how to construct and store those index files and main files to achieve the minimum seek time, maximum throughout are still need to be studied.

Finally, this thesis is based on a traditional multi-disk subsystem, as we defined in Section 1.3. Disk synchronization and disk striping multi-disk subsystems are more powerful than a traditional one in improving the I/O service time, how to introduce these two multi-disk subsystems is still an area of research.

APPENDIX A

FIGURES FOR CHAPTER 5

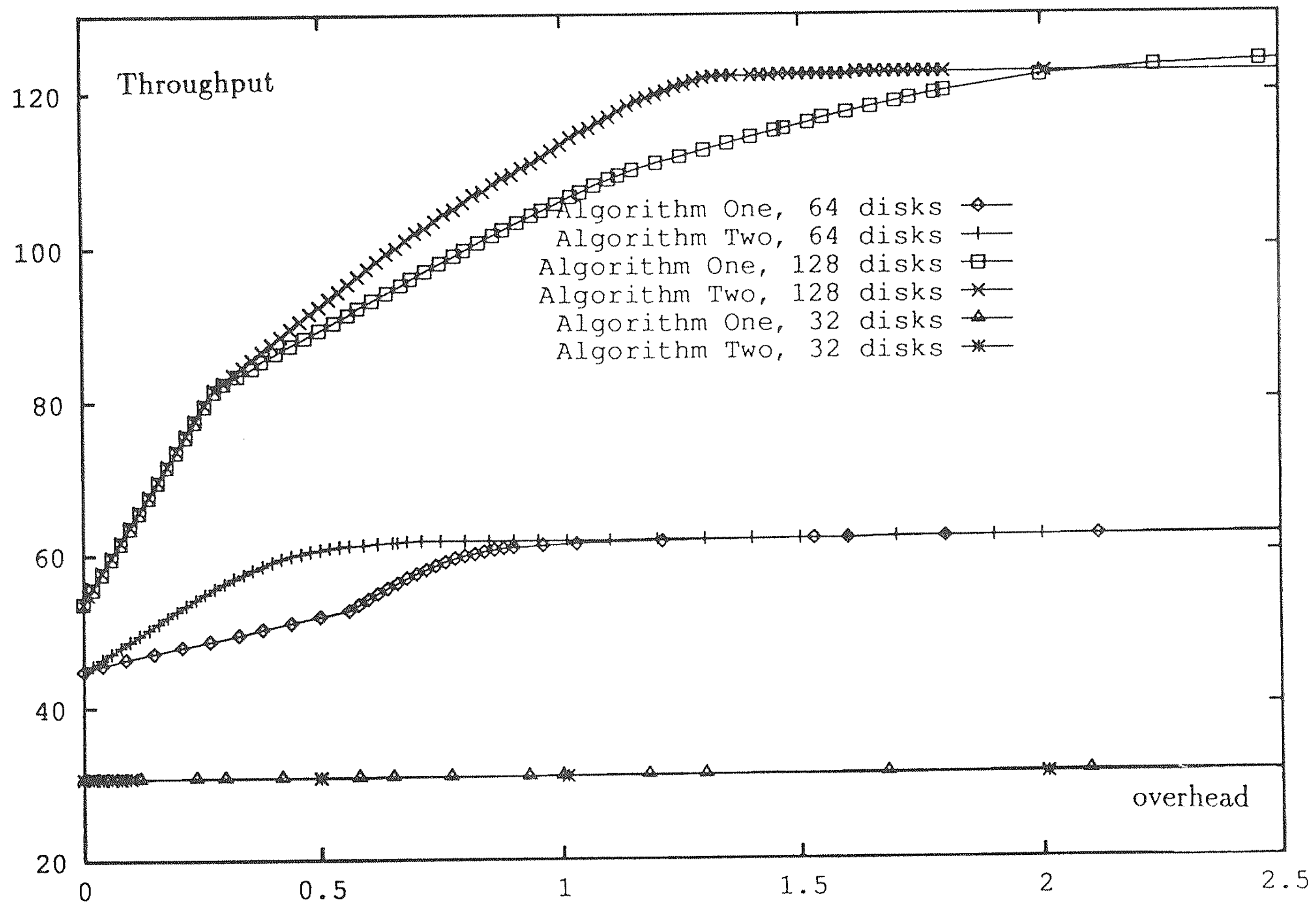

Figure A.1 throughput:: 100 classes, Gaussian

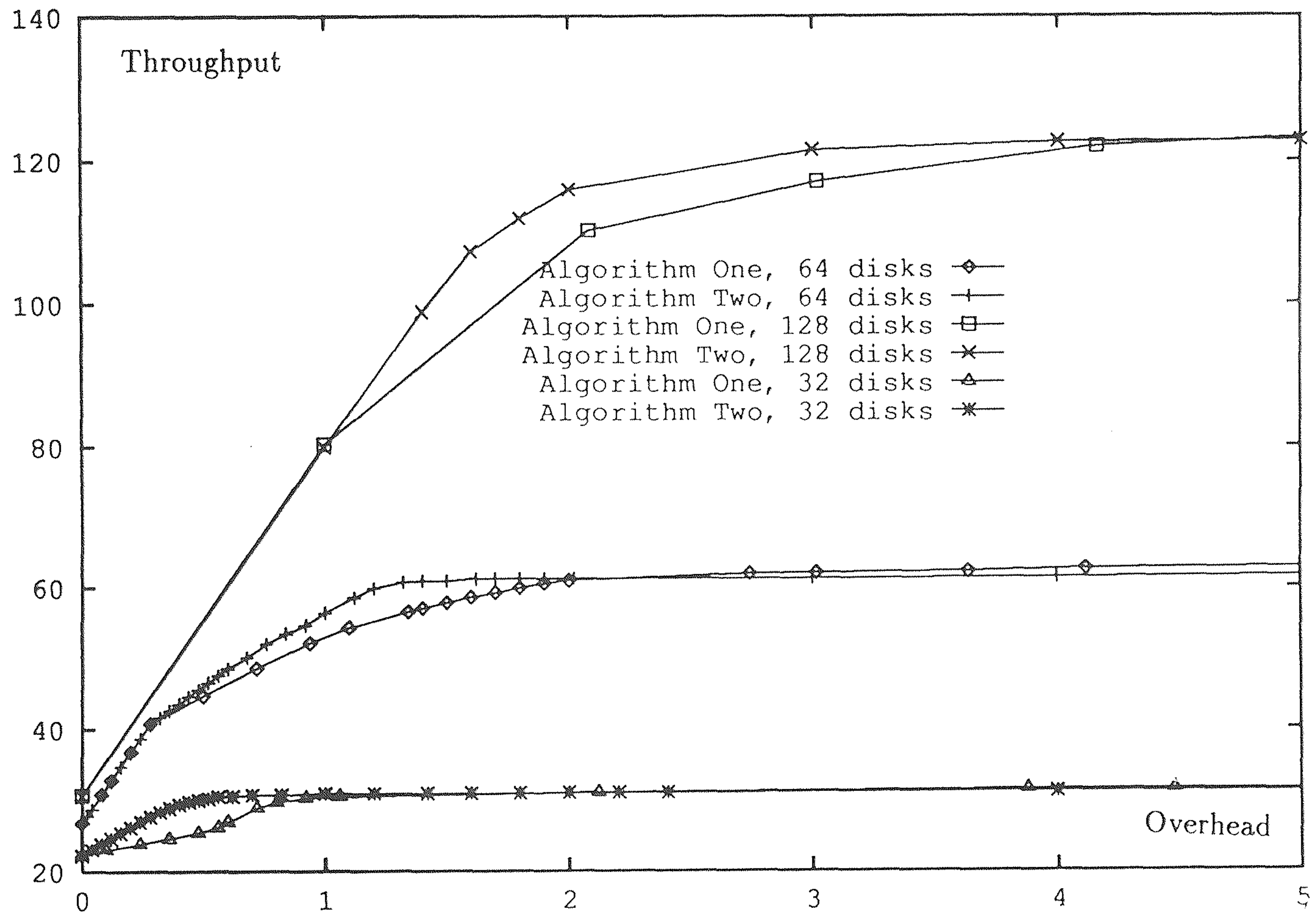

Figure A.2 throughput:: 50 classes, Gaussian

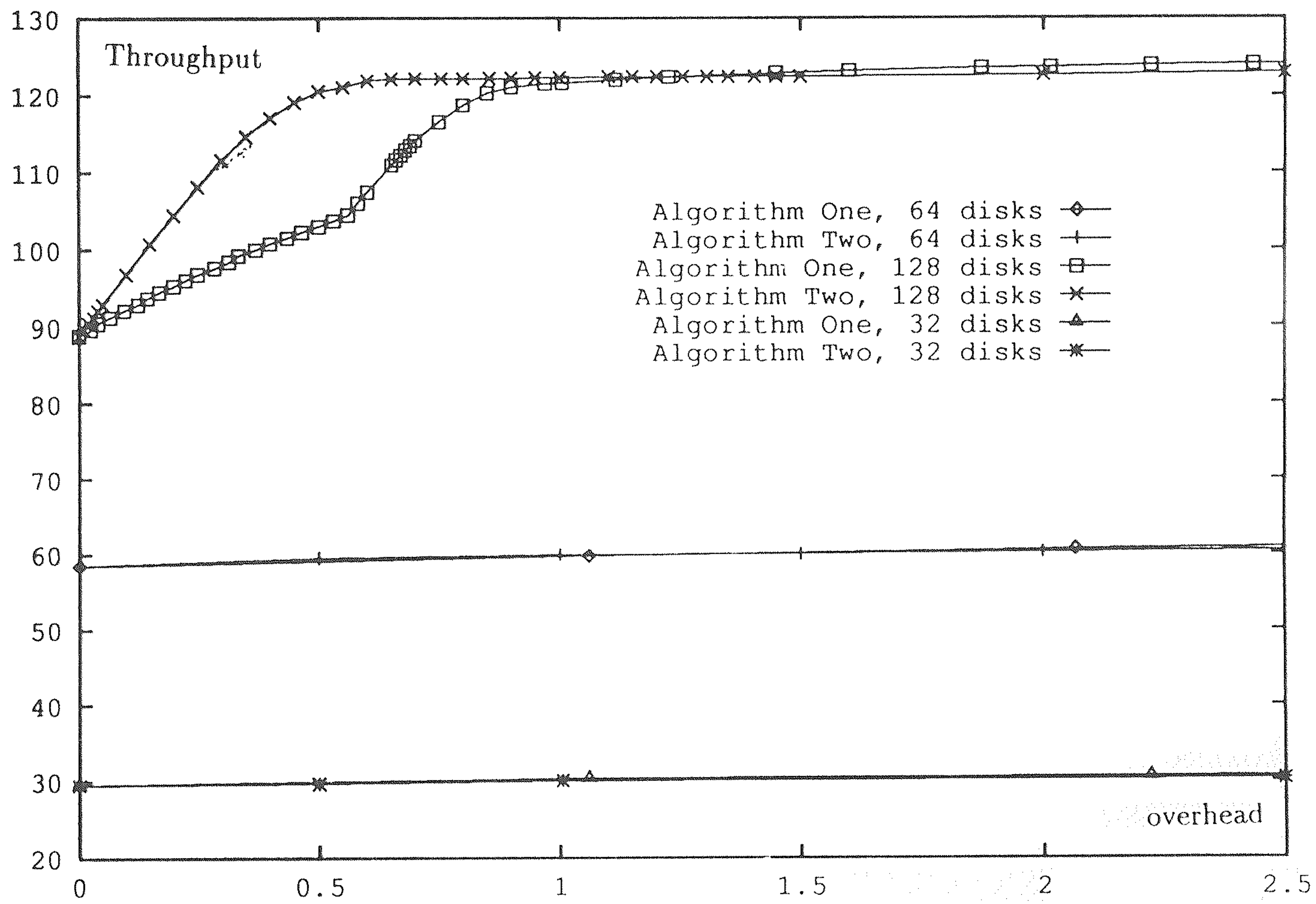

Figure A.3 throughput:: 200 classes, Gaussian

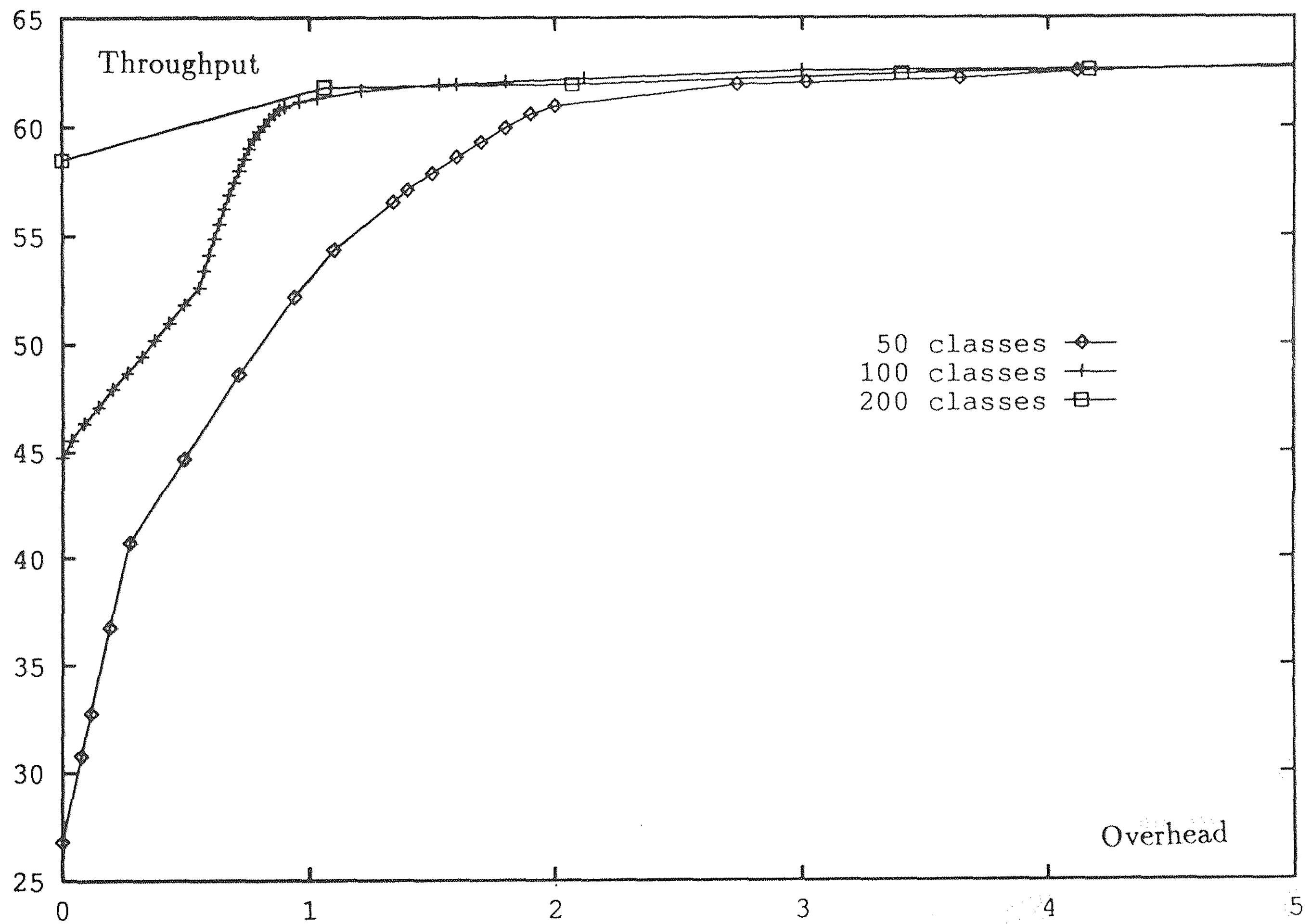

Figure A.4 throughput:: 64 disks, Gaussian

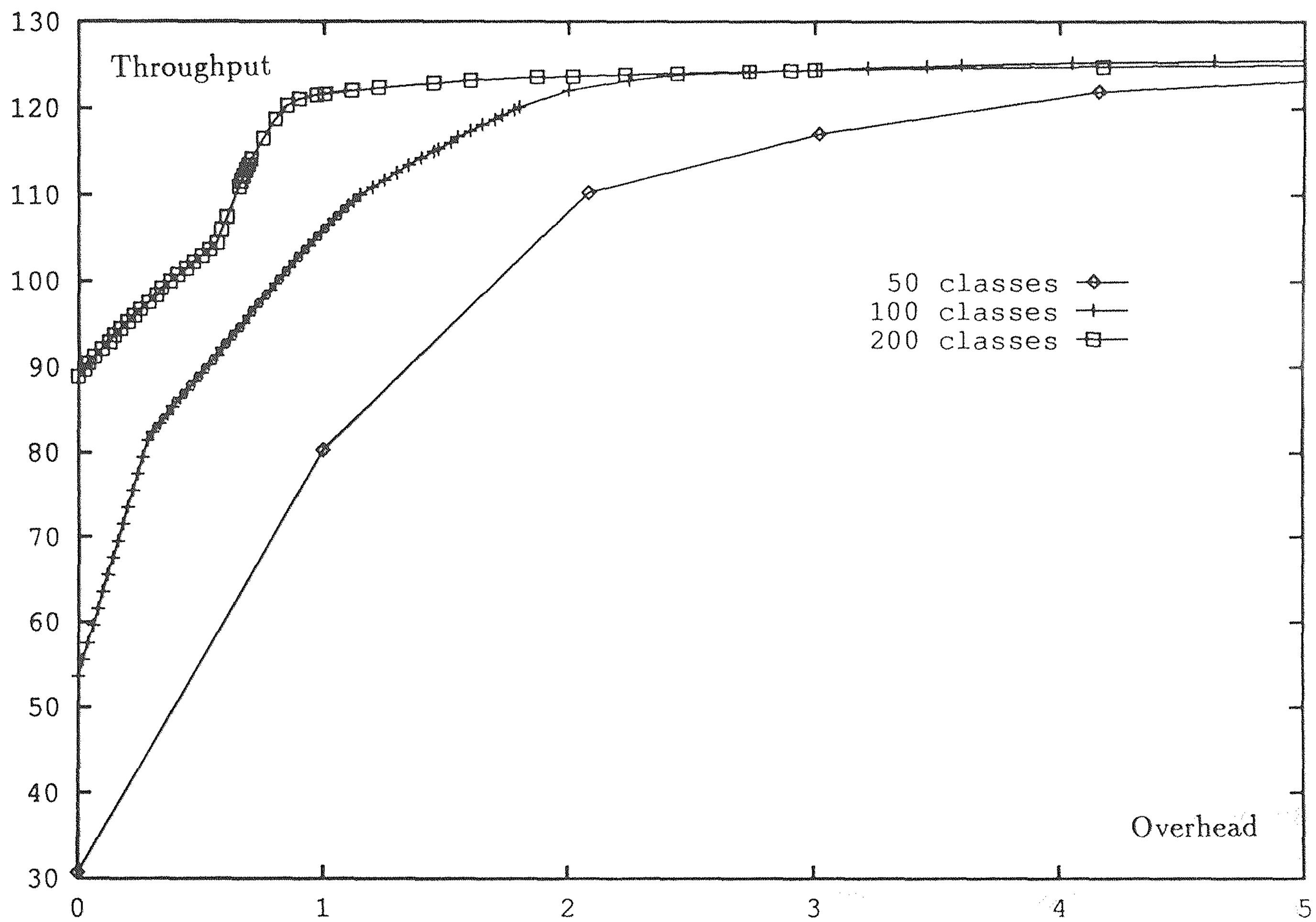

Figure A.5 throughput:: 128 disks, Gaussian

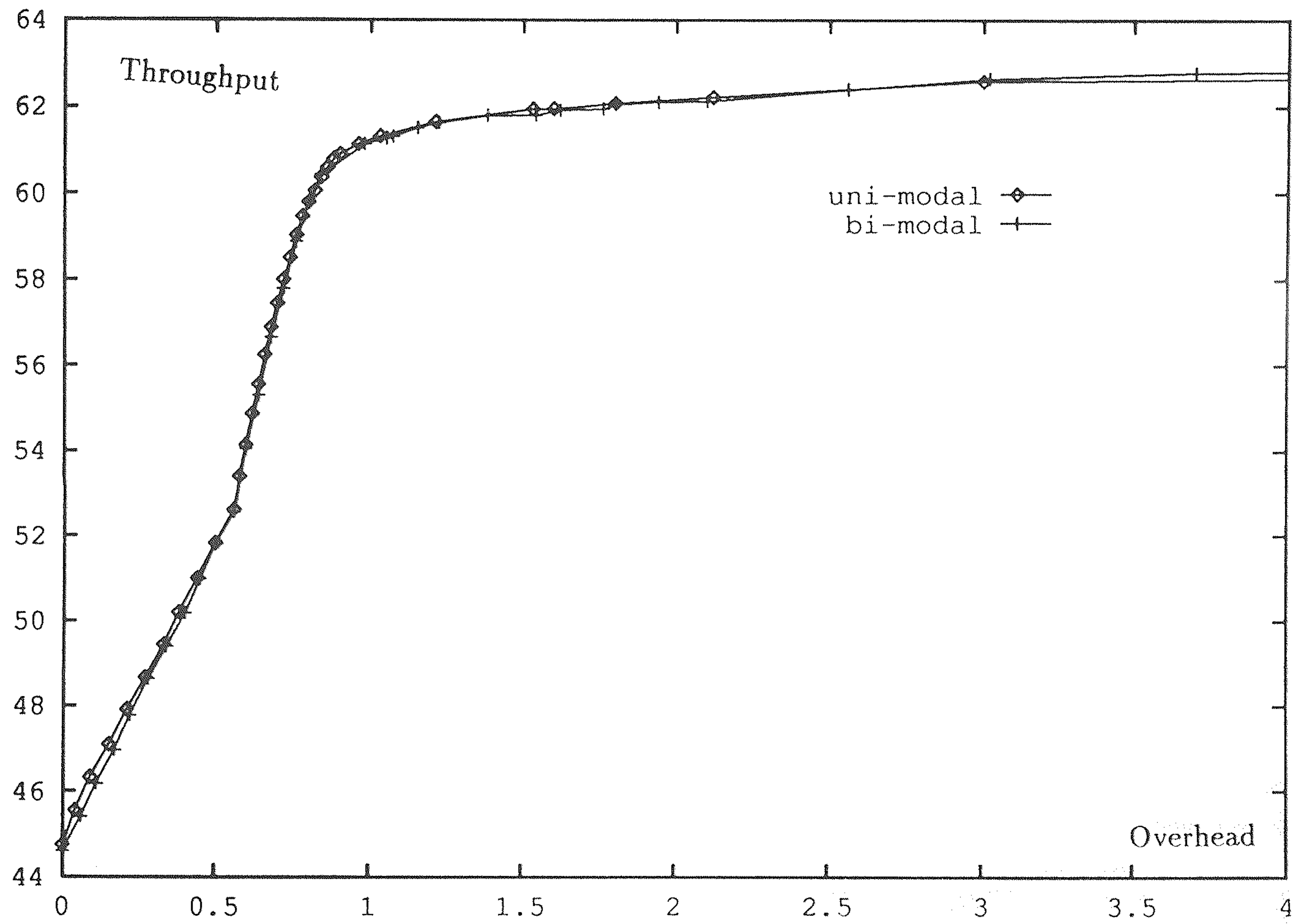

Figure A.6 throughput:: 64 disks, 100 classes

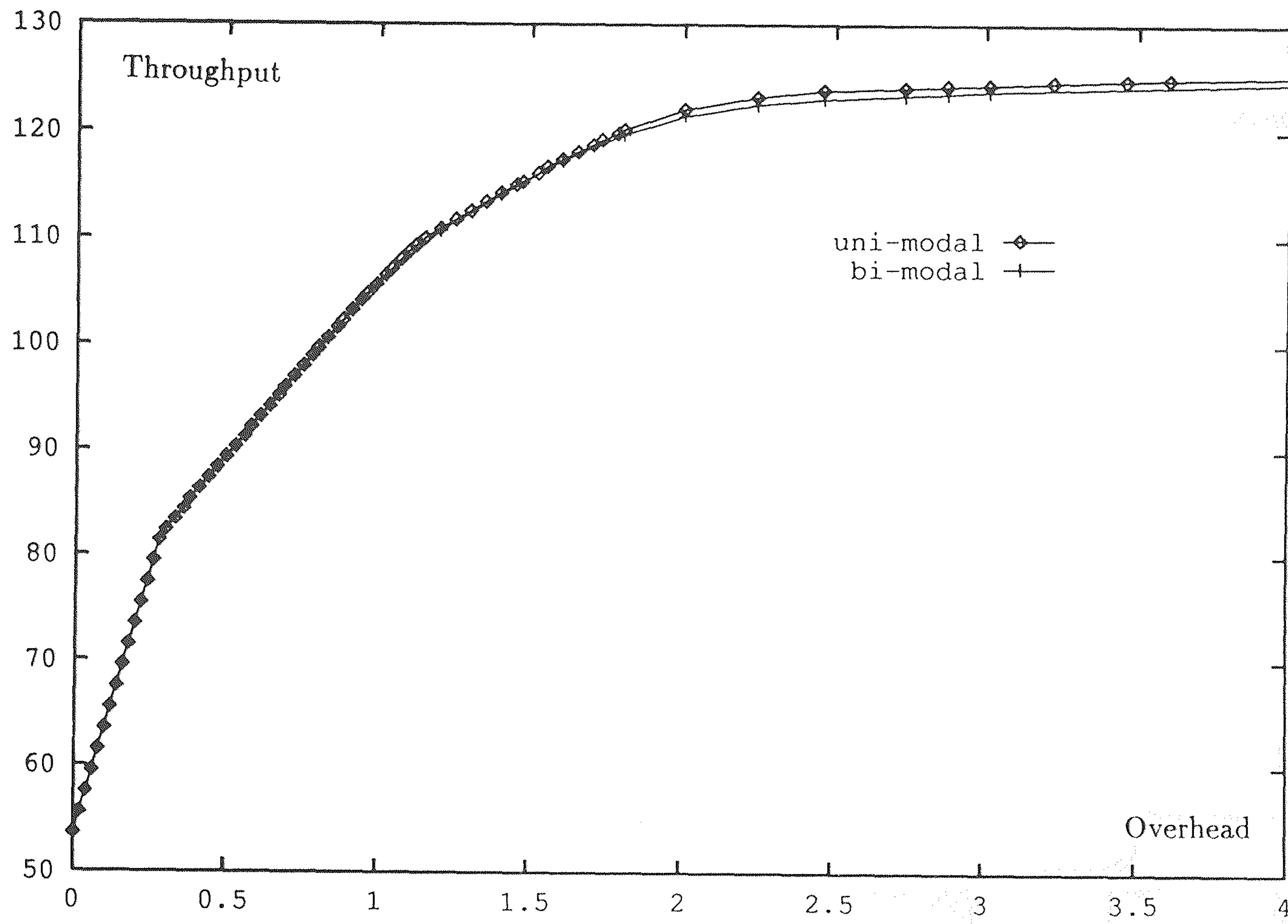

Figure A.7 throughput:: 128 disks, 100 classes

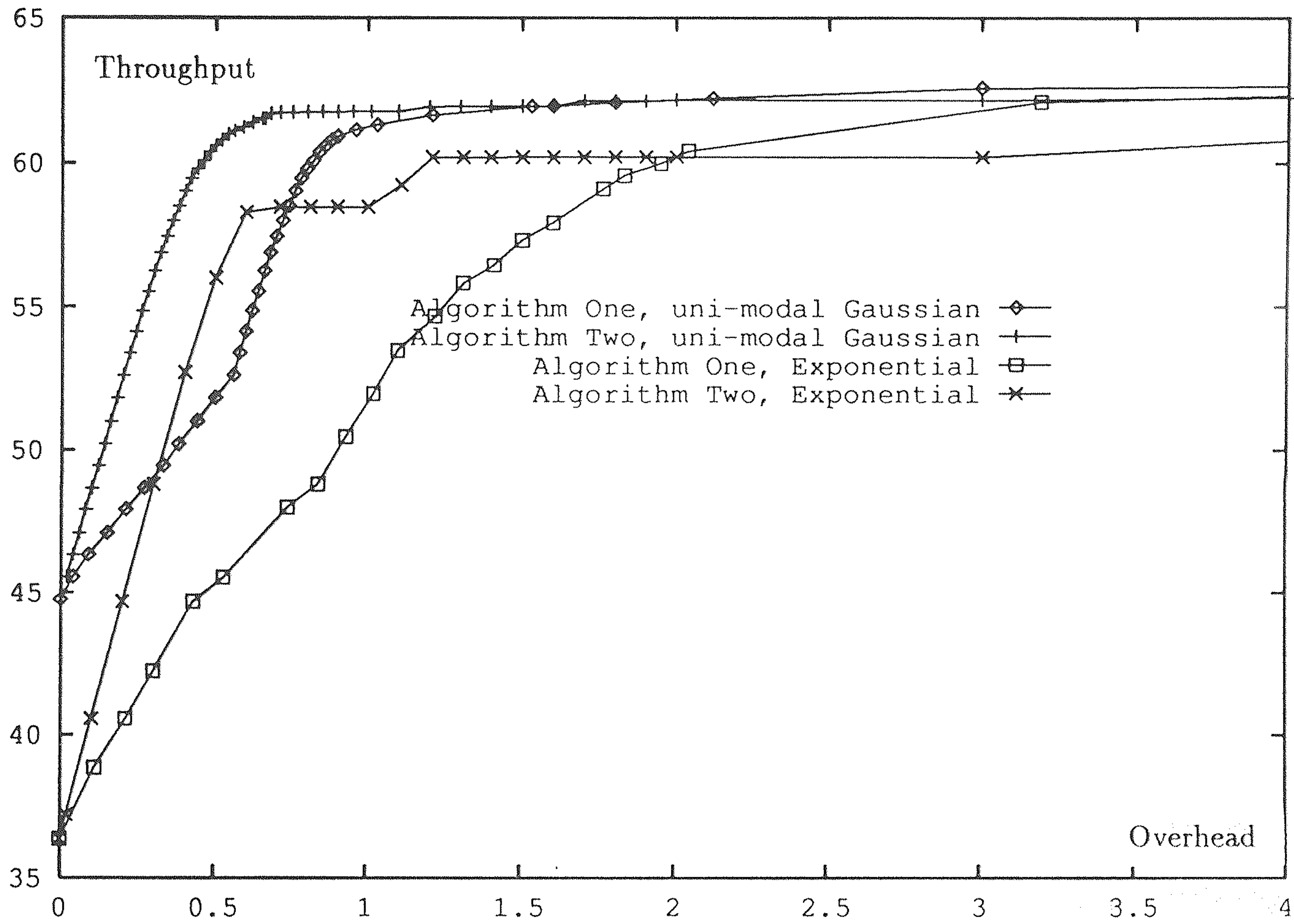

Figure A.8 Two Different Distributions

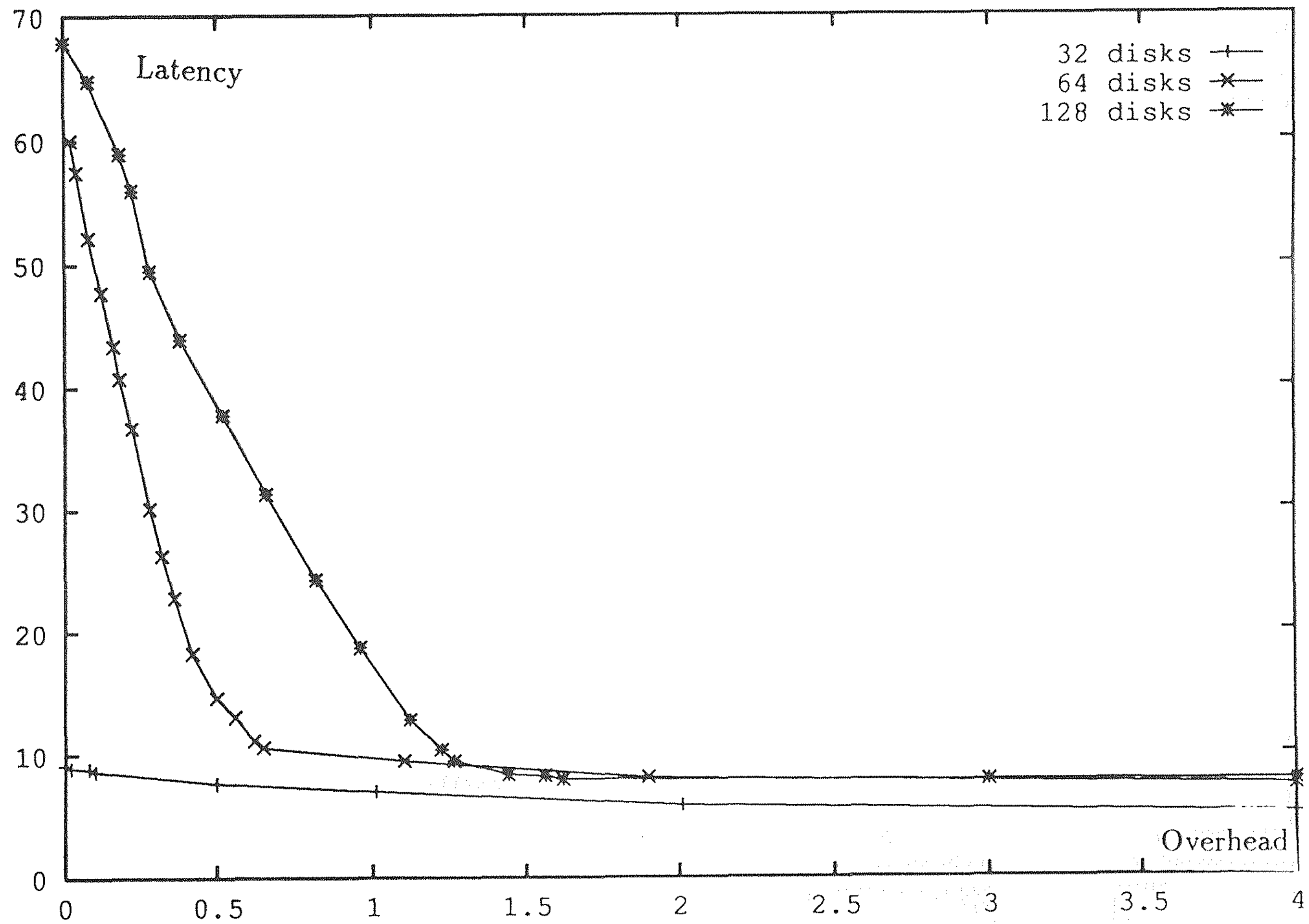

Figure A.9 latency:: 100 classes, Algorithm Two

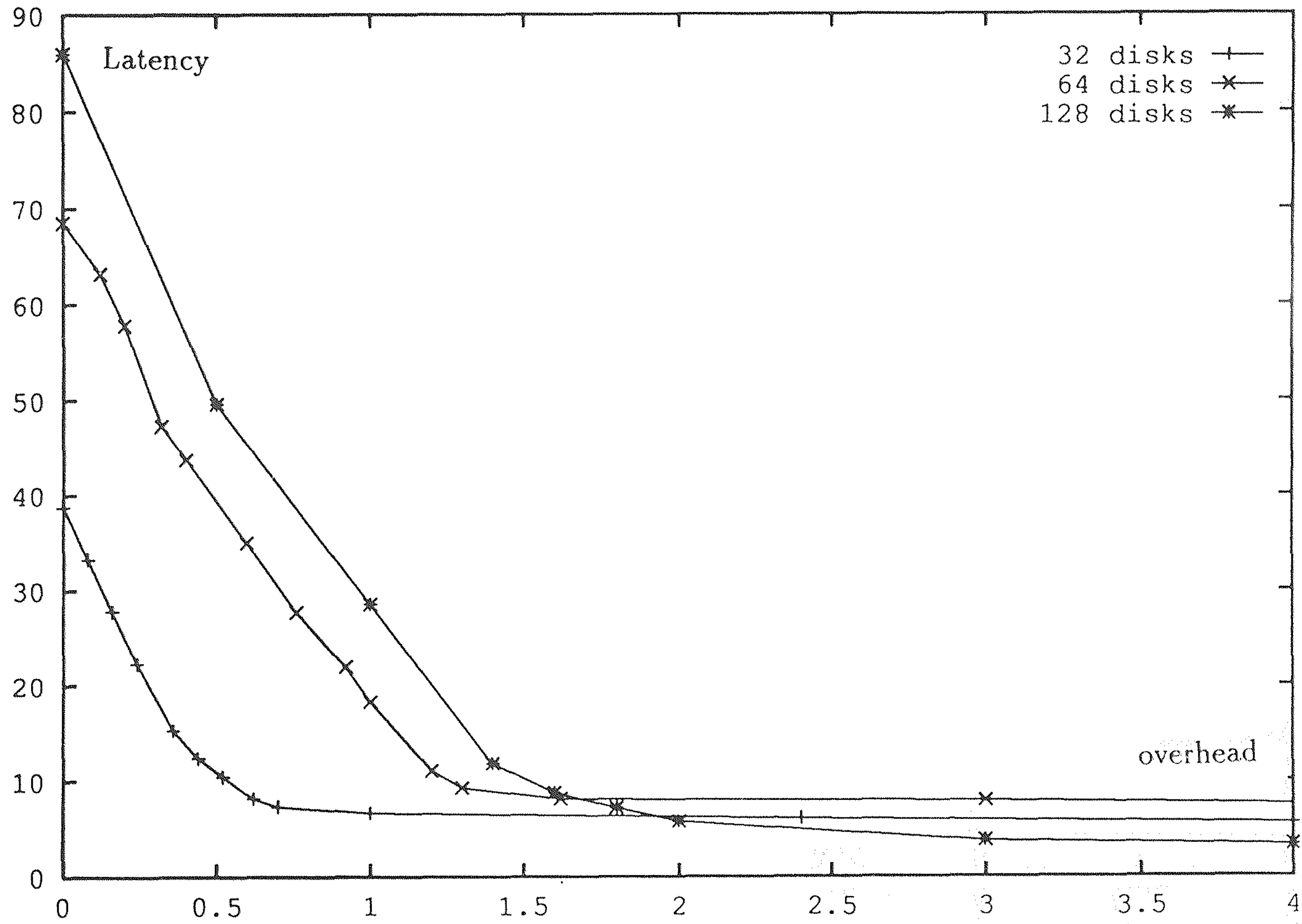

Figure A.10 latency:: 50 classes, Algorithm Two

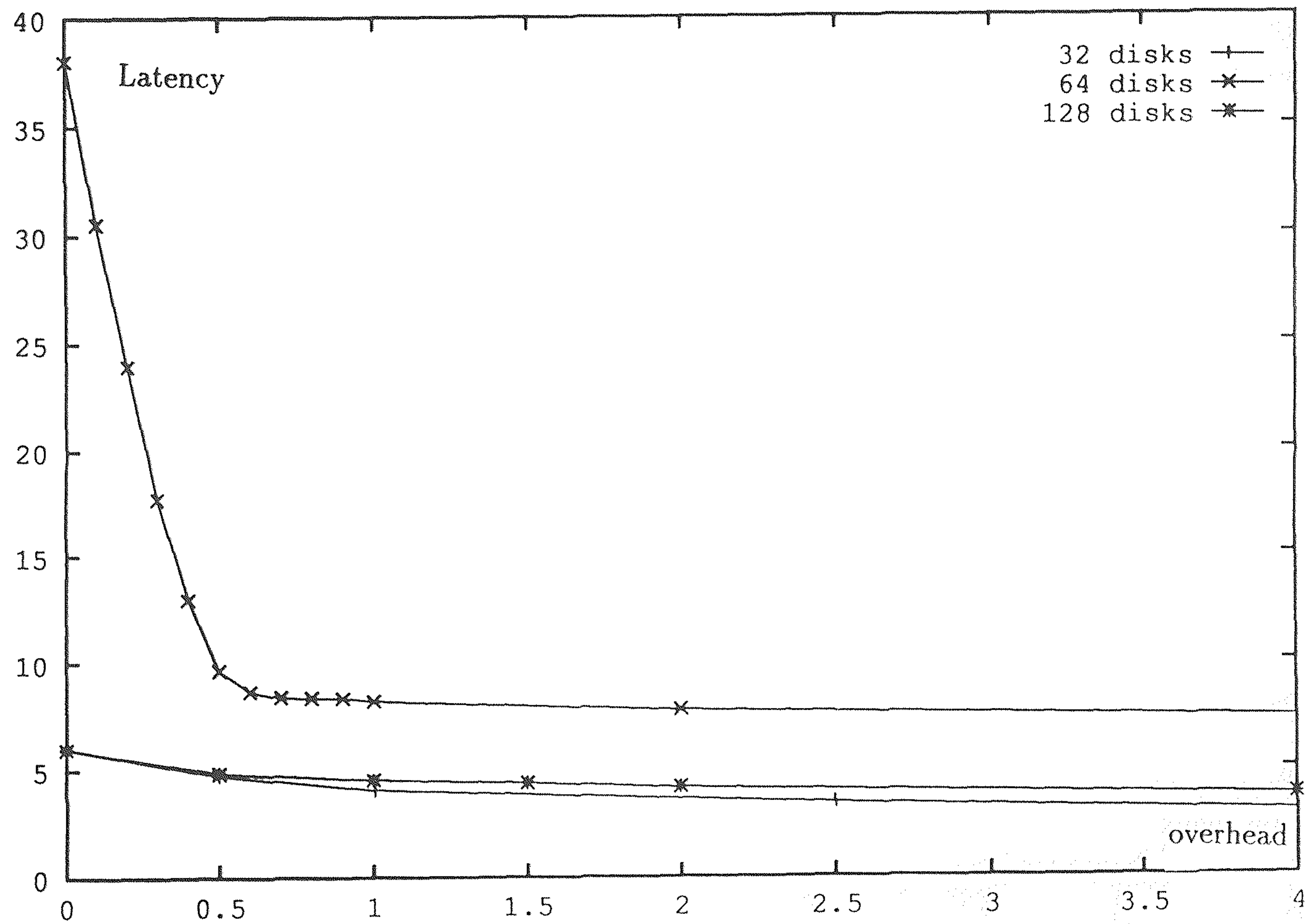

Figure A.11 latency:: 200 classes, Algorithm Two

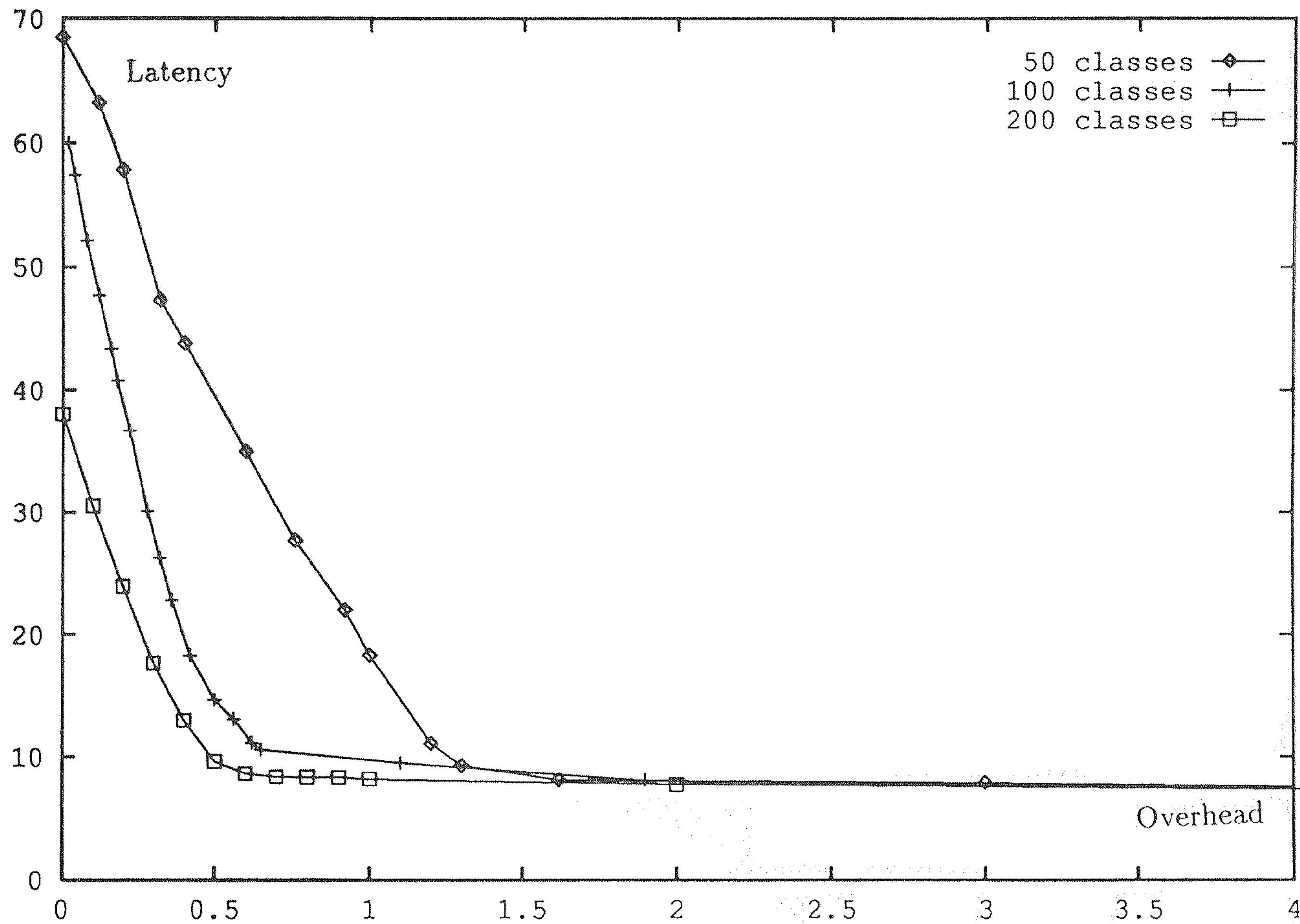

Figure A.12 latency:: 64 disks, Algorithm Two

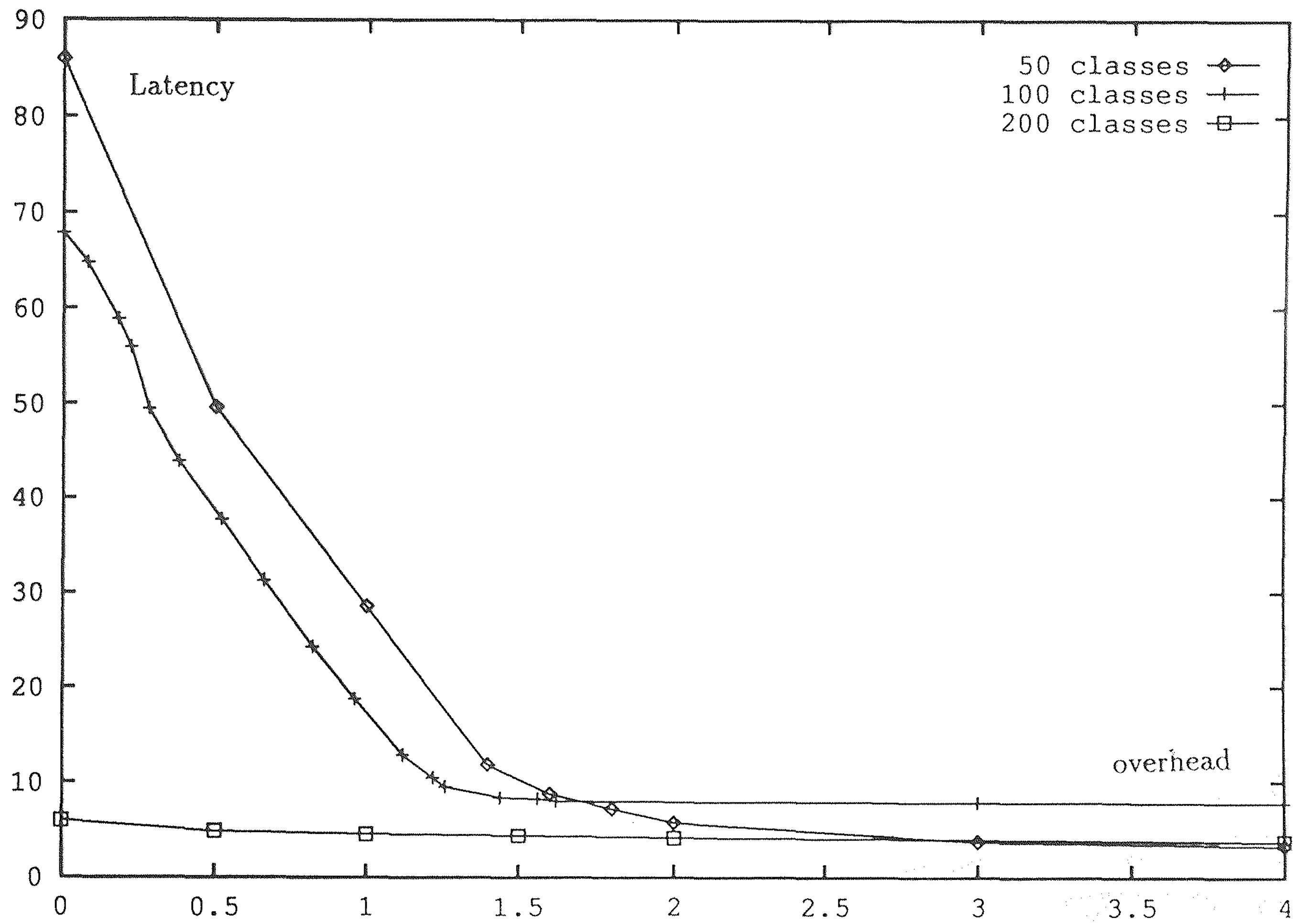

Figure A.13 latency:: 128 disks, Algorithm Two

APPENDIX B

SOURCE CODES

```
/*************************************************************
            head.h

     globle definitions of source codes
*************************************************************/

#define DISK 1024
#define MAXSIZE 50000
#include <stdio.h>
#include <math.h>

typedef struct queue {
  int q_name;
  int c_name;
  int time_in;
  struct queue *next;
} QUEUE, *QUEUEP;




typedef struct table {
  int c_name;
  double c_freq;
  struct table *next;
} TABLE, *TABLEP;


int disk_size[DISK], request_size, name;
int  req_freq, no_loop, no_disk, ntime;
int  diskfull[DISK], out, delay;
double GaussRe1, GaussRe2;
int total_class[DISK];

int SET;
int total_classes;
```

```
double class_freq[3000];
int  maxclass, maxdisk, mindisk;
double  maxfreq, maxtotal, mintotal;

double total_freq[DISK];
double accuracy;
double overhead;

float lamta;
QUEUEP before;
QUEUEP request;
QUEUEP bfr;
QUEUEP cpu, ptr[DISK];
QUEUEP delocate(QUEUEP ff);
QUEUEP relocation(QUEUEP ff);
QUEUEP nextlocation(QUEUEP ff);
QUEUEP disk[DISK];
TABLEP class_list[DISK];
TABLEP fp[DISK];
TABLEP bpr[DISK];


/**************************************************
                  main.c

   this is the main program
   using infinite system and disk buffers
**************************************************/

#include <stdlib.h>
#include "head.h"


main(argc, argv)
int argc;
char *argv[];
{
  int m;

  help(argc);
  init1(argc, argv);
  setup_database();
  disk_overhead();
```

```
    init2();
    run();
    result();
    finish();
  free_database();

}  /*  end of main  */


/*****************************************************
             run.c

        Run the simulation
*****************************************************/

#include <stdlib.h>
#include "head.h"


void run()
{
  int i, k, iseed, l, j;
  double a, b, drand48();
  QUEUEP ww[DISK];

  iseed=64;
  srand48(iseed);
  a=pow(M_E, -lamta);

  for (i=0; i<no_loop; i++) {
    b=1.0;
    while((b*=drand48()) >= a) req_freq++;

    for(k=0;k<no_disk;k++) {
          if(disk_size[k]>0) SET=1;
        }

        if(SET==1)  diskserver();
    if(req_freq > 0) {
       for(k=1; k<=req_freq; k++) {
                  gauss_incoming(name);
                  name++;
                }
        }
```

```
        if(request_size>0) {
          if(bfr != NULL) {
            cpu=bfr;
            bfr=NULL;
            before=NULL;
          }
          for(l=request_size; l>0; l--) {
  dispatch();
  }
        }
  if ((bfr==NULL)&&(request_size==0)&&(cpu->c_name!=0)){
     printf("error w\n");
     exit(1);
          }
  if ((bfr!=NULL)&&(request_size==0)) {
     printf("error s\n");
     exit(1);
          }
  if ((bfr==NULL)&&(request_size!=0)) {
     printf("error r\n");
     exit(1);
          }
  if (request->next!=NULL) {
     printf("error t\n");
     exit(1);
          }
  if (request!=cpu) {
     printf("error y\n");
             printf("request_size is %d\n", request_size);
     exit(1);
          }
  if (cpu->next!=NULL) {
     printf("error p\n");
             printf("i= %d lamta= %f \n", i, lamta);
     exit(1);
          }

    req_freq=0;
        ntime++;
  }
}  /* end of run */
```

```
/*******************************************************
               general-utilities.c

*******************************************************/

#include "head.h"

void help(int argc)
{
  if (argc != 2) {
  printf("Usage: input accuracy\n");
  exit(1);
  }
}


void init1(int argc, char *argv[])
{
  int d;

   sscanf(argv[1], "%lf", &accuracy);
   no_disk=32;
   lamta=32.0;
   no_loop=300;
   for(d=0;d<no_disk;d++) {
      if((class_list[d]=(TABLEP)malloc(sizeof(TABLE)))==NULL){
        printf("malloc error in init1\n");
        exit(1);
      }
      class_list[d]->c_name=-1;
      class_list[d]->c_freq=0.0;
      total_freq[d]=0.0;
      total_class[d]=0;
      class_list[d]->next=NULL;
      fp[d]=class_list[d];
      bpr[d]=fp[d];
  }
   GaussRe1=GaussRe2=0;
   overhead=0.0;
}
```

```
void init2()
{
   int d;

   if((request=(QUEUEP) malloc(sizeof(QUEUE))) == NULL) {
     printf("malloc error in init2(1)\n");
     exit(1);
   }
   request_size=0;
   request->q_name=-1;
   request->c_name=0;
   request->time_in=0;
   request->next=NULL;
   cpu=request;  /* cpu initially points to the beginning of
                    request's link list */

   for (d=0; d<no_disk; d++)  {
     if((disk[d]=(QUEUEP) malloc(sizeof(QUEUE))) == NULL) {
       printf("malloc error in init2(2) when d is %d\n", d);
       exit(1);
     }
     disk_size[d]=0;
     disk[d]->q_name=-1;
     disk[d]->c_name=-1;
     disk[d]->time_in=0;
     disk[d]->next=NULL;
     diskfull[d]=0;
     ptr[d]=disk[d];  /* ptr[d] initially points to the beginning
                         of disk[d]'s link list   */
   }


   bfr=NULL;  /* bfr records the first address blocked by disk due
                 to disk full  */

   before=NULL;
   name=0;
   out=0;
   delay=0;
   SET=0;
   ntime=0;
}
```

```
void result()
{
   double throughput, latency;
   throughput=(double)out/ntime;
   latency=(double)delay/out;
   printf("%d %.7f   %.7f %.7f %.7f\n", \
           no_disk,overhead,throughput,throughput/31.8267,latency);
}



void finish()
{  int i, d;
   QUEUEP tmp, fs;

   for(d=0;d<no_disk;d++) {
     if(disk_size[d]>0) {
       for(i=disk_size[d];i>0;i--) {
         tmp=ptr[d]->next;
         free(ptr[d]);
         ptr[d]=tmp;
       }
       free(ptr[d]);
     }
  }

  if(request_size>0) {
    if(bfr != NULL) fs=bfr;
    if(bfr == NULL) fs=cpu;
    for(i=request_size;i>0;i--) {
      tmp=fs->next;
      free(fs);
      fs=tmp;
    }
    free(fs);
  }
}


QUEUEP delocate(QUEUEP ff)
{
  QUEUEP tmp;
  tmp=ff->next;
  if(before != NULL) before->next=tmp;
```

```
  ff->next=NULL;
  free(ff);
  return(tmp);
}



QUEUEP nextlocation(QUEUEP ff)
{
   if (bfr == NULL) bfr=ff;
   before=ff;
   ff=ff->next;
   before->next=ff;
   return(ff);
}




QUEUEP relocation(QUEUEP ff)
{
  QUEUEP tmp;
  tmp=ff->next;
  free(ff);
  return(tmp);
}



/*****************************************************
                 system-utilities.c

*****************************************************/

#include "head.h"


void diskserver()
{
     int d, t=0;
     for(d=0; d < no_disk ; d++) {
      if ( disk_size[d]>0) {
        if(ptr[d]->c_name ==0) {
         printf("diskserver error\n");
```

```
          exit(1);
        }
        out++;
        t = ntime - (ptr[d]->time_in);
        delay+=t;
        disk_size[d]--;
        ptr[d]=relocation(ptr[d]);
      }
    }
    SET=0;
}


int dispatch()
{
   int i, j;
   int set=0;
   int less_queue;
   int less_size;

   less_queue=-1;
   less_size=-1;
   if(cpu->c_name == 0) {
     printf("dispatch error\n");
     exit(1);
   }
   for(i=0;i<no_disk;i++) {
      fp[i]=bpr[i];
      for(j=total_class[i];(j>0) && (set==0);j--) {
          if(cpu->c_name == fp[i]->c_name) set=1;
          if(fp[i]->next == NULL) j=-1;
          else fp[i]=fp[i]->next;
      }
      if(set==1) {
         if (less_size == -1) {
           less_size=disk_size[i];
           less_queue=i;
         }
         else if(disk_size[i] < less_size) {
            less_size=disk_size[i];
            less_queue=i;
         }
      }
      set=0;
```

```
  }

        disk_size[less_queue]++;
        disk[less_queue]->q_name=cpu->q_name;
        disk[less_queue]->c_name=cpu->c_name;
        disk[less_queue]->time_in=cpu->time_in;
        if((disk[less_queue]->next=(QUEUEP)malloc(sizeof(QUEUE)))
           == NULL) {
         printf("malloc error in dispatch\n");
         exit(1);
        }
        disk[less_queue]=disk[less_queue]->next;
        disk[less_queue]->c_name=0;
        disk[less_queue]->next=NULL;
        request_size--;
        cpu=delocate(cpu);
}



/*******************************************************
              setup_database.c

*******************************************************/

#include "head.h"
#include <stdio.h>
#include <math.h>

setup_database()
{
        int i, j;
        TABLEP ff[DISK];

        calculation();
        database();
        calculation();
        adjustment();
/*
         for(i=0;i<no_disk;i++) {
          ff[i]=bpr[i];
          j=total_class[i];
          printf("\n\ndisk %d's class_list is:  ", i);
          while(j>0) {
```

```
            printf("%d, %f ", ff[i]->c_name, ff[i]->c_freq);
            ff[i]=ff[i]->next;
            j--;
          }
          printf("\ndisk %d's total_class is %d, total_freq is %f\
                 \n", i, total_class[i], total_freq[i]);
        }
*/
}



/*******************************************************
   database-utilities.c

*******************************************************/

#include "head.h"

calculation()
{
   int i;

   double mu, sigma;
   double x1, x2, z1, z2, f1, f2, f;

   total_classes=100;
   for(i=0;i<total_classes+1;i++) class_freq[i]=0.0;
   mu=400.0;
   sigma=100.0;
   x2=mu;
   x1=mu-4.0;
   for(i=0;i<total_classes;i++) {
     z1=(x1-mu)/sigma;
     z2=(x2-mu)/sigma;
     f1=.5+.5*erf(z1/(sqrt(2)));
     f2=.5+.5*erf(z2/(sqrt(2)));
     if(x2==4.0) f1=0.0;
     f=(f2-f1)*2;
     class_freq[i+1]=f;
     x2-=4.0;
     x1-=4.0;
   }
```

```
}


Maxclass()
{
   int i;

   maxfreq=0.0;
   maxclass=total_classes-1;
   for (i=1;i<=total_classes;i++) {
     if(class_freq[i] > maxfreq) {
        maxfreq=class_freq[i];
        maxclass=i;
     }
   }
   if(maxfreq>0) class_freq[maxclass]=-1.0;
}


MaxMindisk()
{
   int i;

   maxtotal=mintotal=total_freq[0];
   maxdisk=mindisk=0;
   for (i=1;i<no_disk;i++) {
     if(total_freq[i] > maxtotal) {
        maxtotal=total_freq[i];
        maxdisk=i;
     }
     if(total_freq[i] < mintotal) {
        mintotal=total_freq[i];
        mindisk=i;
     }
   }
}


          /*** Mappig Step  ***/
```

```
database()
{
    int i, j, end;

    end = (total_classes > no_disk) ? no_disk : total_classes;
    for(i=0;i<end;i++) {
      Maxclass();
      class_list[i]->c_name=maxclass;
      class_list[i]->c_freq=maxfreq;
      total_freq[i]+=(double) maxfreq;
      total_class[i]++;
      if((class_list[i]->next=(TABLEP)malloc(sizeof(TABLE)))==NULL){
        printf("malloc error in database(1)\n");
        exit(1);
      }
      class_list[i]=class_list[i]->next;
      class_list[i]->next=NULL;
    }
    if(total_classes>no_disk) {
     for(j=no_disk;j<total_classes;j++) {
       Maxclass();
       MaxMindisk();
       if(maxfreq>0) {
          class_list[mindisk]->c_name=maxclass;
          class_list[mindisk]->c_freq=maxfreq;
          total_freq[mindisk]+=(double) maxfreq;
          total_class[mindisk]++;
          if((class_list[mindisk]->next=(TABLEP)malloc(sizeof(TABLE)))
             == NULL) {
            printf("malloc error in database(2)\n");
            exit(1);
          }
          class_list[mindisk]=class_list[mindisk]->next;
          class_list[mindisk]->next=NULL;
       }
     }
   }
}  /*end of mapping step*/



  /*** replication step for algorithm one ***/
```

```
adjustment()
{
   int i, j;
   int sett=0;
   int tot;
   int class;
   int copy=0;
   double eventotal;
   TABLEP fp[DISK], tmp;

   MaxMindisk();
   disk_overhead();
   eventotal=0.0;
   while((overhead < accuracy) && (maxdisk != mindisk)) {
     tot=total_class[mindisk]+total_class[maxdisk];
     fp[maxdisk]=bpr[maxdisk];
     for(i=0;i<total_class[maxdisk];i++) {
       class=fp[maxdisk]->c_name;
       eventotal=eventotal+class_freq[class];
       class_list[mindisk]->c_name=fp[maxdisk]->c_name;
       class_list[mindisk]->c_freq=fp[maxdisk]->c_freq;
       if((class_list[mindisk]->next=(TABLEP) malloc(sizeof(TABLE)))
         == NULL){
           printf("malloc error in adjustment(1)\n");
           exit(1);
       }
       class_list[mindisk]=class_list[mindisk]->next;
       class_list[mindisk]->next=NULL;
       fp[maxdisk]=fp[maxdisk]->next;
     }

     fp[mindisk]=bpr[mindisk];
     for(i=0;i<total_class[mindisk];i++) {
       class=fp[mindisk]->c_name;
       eventotal=eventotal+class_freq[class];
       fp[mindisk]=fp[mindisk]->next;
     }

     tmp=bpr[maxdisk];
     bpr[maxdisk]=bpr[mindisk];
     class_list[maxdisk]=class_list[mindisk];
     for(i=0;i<no_disk;i++) {
       if(bpr[i]==tmp) copy++;
```

```
      if(bpr[i]==bpr[mindisk])  copy++;
    }

    eventotal=eventotal/copy;
    total_freq[maxdisk]=eventotal;
    total_freq[mindisk]=eventotal;
    total_class[mindisk]=total_class[maxdisk]=tot;

    for(i=0;i<no_disk;i++) {
      if(bpr[i]==tmp) {
        bpr[i]=bpr[mindisk];
        class_list[i]=class_list[mindisk];
        total_freq[i]=total_freq[mindisk];
        total_class[i]=total_class[mindisk];
      }
      if(bpr[i]==bpr[mindisk]) {
        total_freq[i]=total_freq[mindisk];
        total_class[i]=total_class[mindisk];
      }
    }
    tmp=NULL;

    MaxMindisk();
    disk_overhead();
    eventotal=0.0;
    copy=0;
  }
}  /* end of replication step */



   /*** replication for algorithm two ***/
adjustment()
{
   int i, j;
   int sett=0;
   int tot;
   int class;
   int copy=0;
   double eventotal;
   TABLEP fp[DISK], tmp;

   MaxMindisk();
   disk_overhead();
```

```
eventotal=0.0;
while((overhead < accuracy) && (maxdisk != mindisk)) {
  tot=total_class[mindisk]+total_class[maxdisk];
  fp[maxdisk]=bpr[maxdisk];
  for(i=0;i<total_class[maxdisk];i++) {
    class=fp[maxdisk]->c_name;
    eventotal=eventotal+class_freq[class];
    class_list[mindisk]->c_name=fp[maxdisk]->c_name;
    class_list[mindisk]->c_freq=fp[maxdisk]->c_freq;
    if((class_list[mindisk]->next=(TABLEP)malloc(sizeof(TABLE)))
       == NULL){
        printf("malloc error in adjustment(1)\n");
        exit(1);
    }
    class_list[mindisk]=class_list[mindisk]->next;
    class_list[mindisk]->next=NULL;
    fp[maxdisk]=fp[maxdisk]->next;
  }

  fp[mindisk]=bpr[mindisk];
  for(i=0;i<total_class[mindisk];i++) {
    class=fp[mindisk]->c_name;
    eventotal=eventotal+class_freq[class];
    fp[mindisk]=fp[mindisk]->next;
  }

  tmp=bpr[maxdisk];
  bpr[maxdisk]=bpr[mindisk];
  class_list[maxdisk]=class_list[mindisk];
  for(i=0;i<no_disk;i++) {
    if(bpr[i]==tmp) copy++;
    if(bpr[i]==bpr[mindisk])  copy++;
  }

  eventotal=eventotal/copy;
  total_freq[maxdisk]=eventotal;
  total_freq[mindisk]=eventotal;
  total_class[mindisk]=total_class[maxdisk]=tot;

  for(i=0;i<no_disk;i++) {
    if(bpr[i]==tmp) {
      bpr[i]=bpr[mindisk];
      class_list[i]=class_list[mindisk];
      total_freq[i]=total_freq[mindisk];
```

```
        total_class[i]=total_class[mindisk];
      }
      if(bpr[i]==bpr[mindisk]) {
        total_freq[i]=total_freq[mindisk];
        total_class[i]=total_class[mindisk];
      }
    }
    tmp=NULL;

    MaxMindisk();
    disk_overhead();
    eventotal=0.0;
    copy=0;
  }
} /* end of replication step */


free_database()
{
   int i, d;
   TABLEP  tmp, ff[DISK];

   for(d=0;d<no_disk;d++) {
     ff[d]=bpr[d];
     for(i=total_class[d];i>0;i--) {
       tmp=ff[d]->next;
       free(ff[d]);
       ff[d]=tmp;
     }
   }
}



disk_overhead()
{
   int i;
   int over=0;

   for (i=0;i<no_disk;i++)  over+=total_class[i];
   overhead=(double)(over-total_classes)/total_classes;
}
```

```
void gauss_incoming(int name)
{
    if((name % 2) == 0) {
      gauss48();
      request->c_name=match(GaussRe1);
    }
    else request->c_name=match(GaussRe2);
    request_size++;
    request->q_name=name;
    request->time_in=ntime;
    if((request->next=(QUEUEP)malloc(sizeof(QUEUE)))==NULL){
     printf("malloc error in gauss_incoming\n");
     exit(1);
    }
    request=request->next;
    request->c_name=0;
    request->next=NULL;
}



int match(xx)
float xx;
{
  int i;
  double step, y1, y2;

  step=4.0;
  y1=y2=400.0;
  for(i=0;i<100;i++) {
    if((4.0<xx) && (xx<=796.0))
      if(((y1-step<xx)&&(xx<=y1))||((y2<xx)&&(xx<=y2+step))){
       return(i+1);
    }
      y1-=step;
      y2+=step;
  }
  if((xx<=4.0) || (xx>796.0)) return(100);
}



gauss48()
```

```
{
   double sigma=100.0;
   int u=400.0;
   double log(), sqrt(), drand48();
   double r, a, b, theta, st, ct;

   theta = 2.0 * M_PI * drand48();
   r = sigma * sqrt( -2.0 * log(drand48()) );
   st = sin(theta);
   ct = cos(theta);
   a=r*st;
   b=r*ct;
   GaussRe1=a+u;
   GaussRe2=b+u;

}
```

REFERENCES

1. H. T. Kung. "Memory Requirements for Balanced Computer Architectures", Proc. 13th Annu. Int. Symp. Comput. Architecture, 1986.

2. M. Y. Kim. "Synchronized disk Interleaving", IEEE Trans. Comput., Vol. C-35, N0. 11, November, 1986.

3. A. L. Narasimha Reddy and Peithviraj Banerjee. "An Evaluation of Multiple-Disk I/O Systems", IEEE Transactions on Computers, Vol. 38, No. 12. December 1989.

4. K. Salem and H. Garcis-Molina. "Disk Striping", Int. Conf. on data Engineering, 1986.

5. M. Livny and S. Khoshafian and H. Boral. "Multi-disk Management Algorithms", Proc. ACM SIGMETRICS Conf., pages 69-77, May 1987.

6. A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*, NewYork: McGraw-Hill Book Company, 1982.