A Performance Model of a PC Based Voice Mail Disk System

by

Michael W. P. Hermus

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1995

Author ....
Department of EECS
May 19, 1995

Certified by ....
Dr. Charles E. Rohrs, Visiting Scientist
Thesis Supervisor

Certified by ....
Maura E. Higgins
Company Supervisor

Accepted by ....
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Performance Model of a PC Based Voice Mail Disk System

by

Michael W.P.Hermus

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 1995 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science and Bachelor of Science in Electrical Engineering and Computer Science.

ABSTRACT

This Thesis presents a model of the performance of a PC based voice mail disk system, towards the goal of answering performance questions about systems that have not yet been built. The voice mail system analyzed is built on a slightly modified 80386 motherboard of the type used to build IBM compatible computers. This thesis is primarily aimed at characterizing the performance of the disk subsystem, which is composed of a SCSI disk controller card, the SCSI bus, and 1 - 6 hard disk drives of variable type.

In order to model the effects that number of disks, disk speed, bus speed, and other system parameters, have on system performance as a whole, two types of model are presented. One is a mathematical model based on queuing theory and Markovian systems, and the second is a software simulator written in the C language. Using this simulator in conjunction with analytical queuing models, this thesis makes conclusions about performance under various conditions and system configurations.

The main observation of this thesis is that current system cannot handle the traffic generated by a high capacity voice mail system because certain disk requests are issued only to a single disk on the system, regardless of the number of drives present. A solution to this problem is presented.

Thesis Supervisor: Charles E. Rohrs

Title: Visiting Scientist, Laboratory for Information and Decision Systems

Table of Contents

1.0 Introduction

1.1 General Overview.

This Thesis presents a performance model of a PC based voice mail disk system, and makes certain conclusions based on the use of that model. The primary motivation for this thesis is the lack of any tool or process to gauge the performance impact of the disk subsystem component's performance on the system as a whole. Since a voice mail system can be viewed as a specialized file server, it is probable that disk performance has a significant impact on total system performance.

The performance model contained herein relates the performance of certain input parameters such as disk seek times with a particular measure of overall system performance. Thus, it allows conclusions to be made about the feasibility of future systems with expanded capacity or differently configured hardware.

In order to model the effects that number of disks, disk speed, bus speed, and other system parameters have on system performance as a whole, two types of model are presented. One is a mathematical model based on queuing theory and Markovian systems, and the second is a software simulator written in the C language. The program simulates two things: the behavior of the disk subsystem in response to any given set of disk requests, and the particular set of disk requests generated by the voice mail system. Using this simulator in conjunction with analytical queuing models, this thesis makes conclusions about performance under various conditions and system configurations.

Thus, this thesis accomplishes two things: 1) It creates a performance model that relates the performance of meaningful system components to some useful measure of overall system performance, and 2) It uses this performance model to draw meaningful conclusions about the system and feasibility of future systems.

1.2 Hardware Overview

The function of the voice mail system, at the highest level, is to store and retrieve digitized voice messages. This includes the following functionality: incoming callers leaving voice messages, users retrieving voice messages, users processing voice message (saving, forwarding, etc.), and users creating messages and sending them to other users.

To accomplish these tasks, the system employs a 80386 motherboard enhanced with a combination of custom and off-the-shelf components. The system is connected to a telephone switch, or Private Branch Exchange(PBX), via telephone lines. The system can be connected to a number of switches made by different manufacturers.

Figure 1 shows a simplified block diagram of the system hardware, which consists of the following:

1.2.1 80386 Motherboard

This 80386 motherboard is basically the same board found in most 386 PC's today. It consists of the Intel 80386 microprocessor running at 33 megahertz, and a16 bit ISA bus. The motherboard holds up to16 megabytes of memory, but on most systems less is used.

1.2.2 Serial I/O Card

This card provides many functions other than Serial I/O to the voice mail system, such as ROM monitor and Watch Dog Timer. However, the main importance of the SI/O card is that it provides a serial control link from the PBX to the voice mail system. Information regarding each call is passed over this link, such as what extension is being called, who is calling, user key presses, etc.

1.2.3 Token Ring Adapter

The Token Ring Adapter is used with systems containing more than 1 voice mail node. It connects each node to a token ring Local Area Network, over which messages and data are passed. This is an off-the-shelf card.

1.2.4 Digital Signal Processor

The DSP card serves several purposes. First and foremost, it is responsible for all the compression and decompression of digital voice data. Necessarily, it also serves as an interface between the voice mail system bus (the ISA bus) and the Voice Bus. The Voice Bus is a Time Division Multiplex link between the telephone line interface cards, or line cards, and the rest of the voice mail system. The bus carries 32 full duplex channels of un-compressed digital voice between the line cards and the DSP. The DSP also sends and receives control information to and from the line cards via the Command Bus, an RS-232 serial link. In addition to all this, DSP has DTMF recognition and generation capabilities.

To accomplish it's tasks, the DSP contains two ADSP2100 digital signal processors, as well as an 80C186 microprocessor.

1.2.5 Disk Controller Card

The disk controller card is an off-the-shelf SCSI hard disk controller.

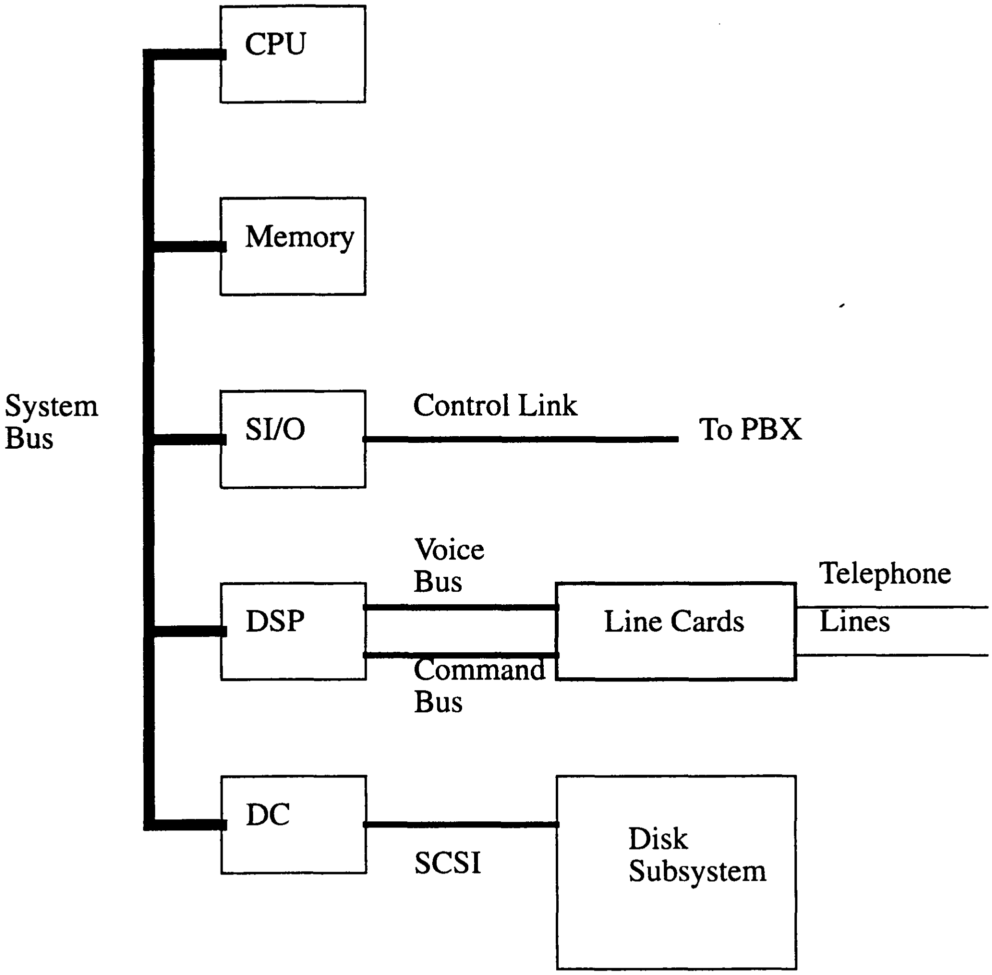

Figure 1. Hardware Architecture

1.2.6 Disk System

The current voice mail system can only have from one to three hard disk drives, each of various capacities and speeds, due to physical cabinet limitations. However, in the future there is no reason the system can not utilize the maximum number of six drives allowed by the SCSI bus.

1.2.7 Line Cards

The line interface cards connect voice mail to telephone lines, which are in turn connected to a PBX. They are the interface between phone lines and the Voice Bus. There are several different line cards, each of which is used for interfacing to different types of telephone lines or PBX's. For instance, one type of line card is used when voice mail is connected to a ROLM PBX, and another is used to connect to an AT&T system. Generally, there will only be one type of line card in any given system.

1.3 Software Overview

Although the topic of this document is a hardware performance model, it is still necessary to understand the system software to some degree.

For the purposes of this discussion, the section of the software that is most important is called the Voice Sub-system. This includes the Voice Channel Protocol Handler and the DSP Interface, both running on the 80386 on top of the CP/386 operating system. The Voice Sub-system also includes the DSP software, running both on the 80C186 and ADSP2100's.

Figure 2 shows a simplified diagram of partial system software and the hardware on which it runs.

1.3.1 VCPH

The Voice Channel Protocol Handler has several functions of interest:

- It controls the transfer of voice data between the hard disks and the system memory.

- It tells the PMDSP Interface to send and receive voice data from the PMDSP card.

- It handles all communication between the PBX and other parts of the voice mail software.

- It manages the voice file system (this part of the VCPH is called the Voice File Handler, or VFH).

1.3.2 DSP Interface

This is the DSP card device driver. It hides hardware complexity and is the general purpose interface between the voice mail software and the DSP card.

1.3.3 PMDSP Software-80C186

The 80C186 software is responsible for most of the functionality of the DSP, with the exception of Compression/Decompression and DTMF generation and recognition.

- It transfers data from the system bus to PMDSP shared memory so that the ADSP2100's can work on it, and vice versa.

- It instructs the ADSP2100's to transfer voice data to and from the TDM Voice Bus

- It sends and receives control information to and from the Command Bus.

1.3.4 PMDSP Software-ADSP2100's

The PMDSP Software consists mainly of compression /decompression algorithms, and DTMF algorithms.

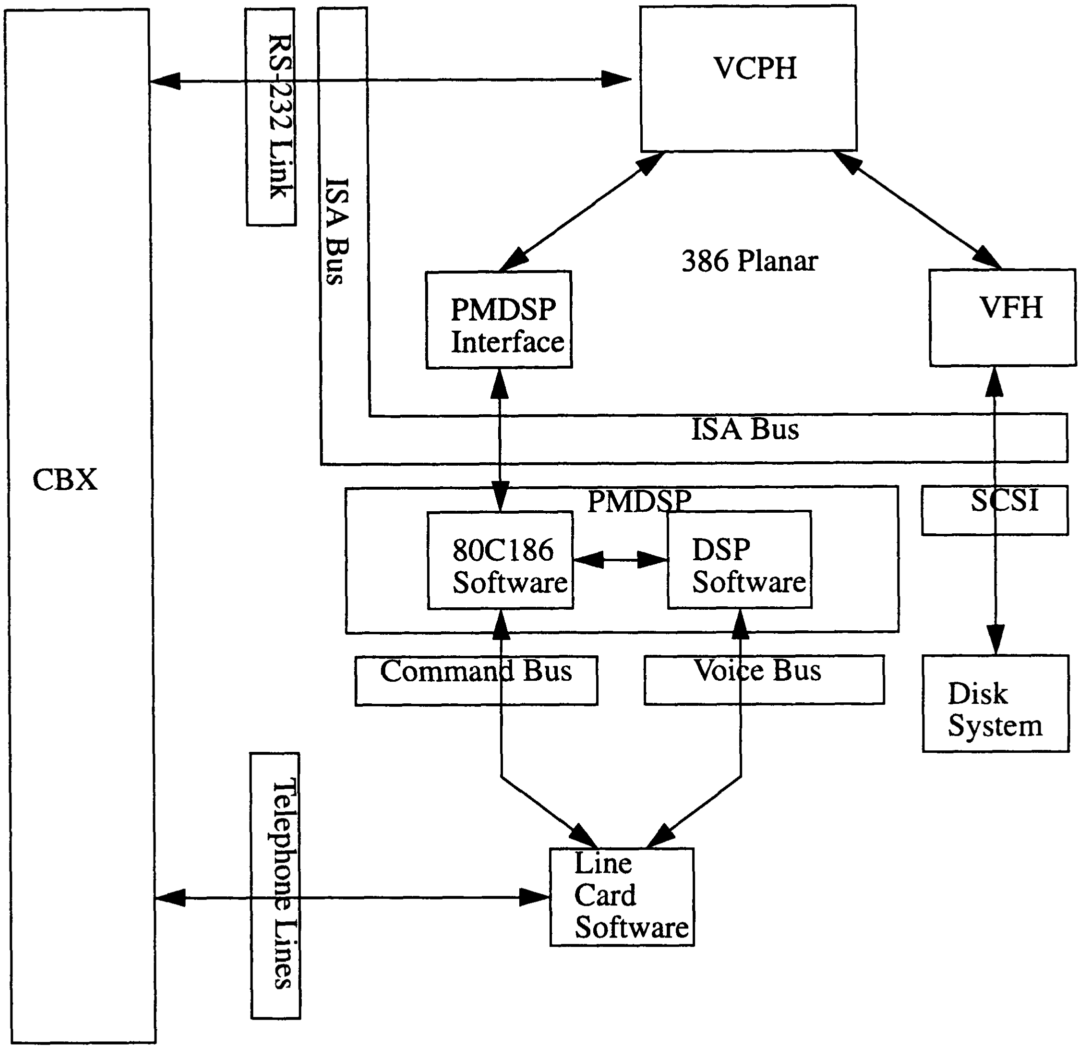

Figure 2. Hardware/Software Architecture

1.4 Thesis Outline

Section 2 presents the various mathematical models used in this thesis, all of which use standard queuing theory and assume Markovian arrival and departure statistics. Models are presented for the call process, single disk system, multiple disk systems, and the SCSI bus.

Section 3 presents the software architecture of the simulator, which is designed too overcome the shortcomings of the mathematical models. It discusses the basic operation of the simulator, showing how it obtains results.

Section 4 discusses the verification of the simulator. The verification process is carried out through the comparison of simulated results to results from the actual system. The simulator was modified somewhat to allow direct comparisons, and as a result, some segments of the program are not verified in that manner. Those were tested separately by providing inputs and checking that the corresponding outputs were appropriate.

In Section 5, the results of the both the simulational and mathematical models are presented. Section 5.1 one explains the basic strategy used to obtain results. Section 5.2 presents the results of the original simulations and states the main observation of those results: the present voice mail disk system is not suitable for systems with large number of channels because there is a performance bottleneck. Section 5.3 presents a likely solution to the bottleneck, and discusses the second set of simulated results obtained with a modified simulator. Section 5.5 presents a comparison between the simulation and the mathematical model.

2.0 Mathematical Model

2.1 Introduction

As most performance models eventually do, this voice mail performance model relies on queuing theory as its basic framework. Queues are simply structures in which jobs wait to be serviced. For example, a line of people at the Department of Motor Vehicles is a queue; the people waiting in line are jobs, and the clerk behind the desk who is dealing with the people is a server.

Queues are used so often that a formalized mathematical theory has evolved around them. There is a special classification system for describing different types of queues, and several formulas exist for deriving information about them.

The characteristics of the queue we are most interested in are as follows:

1)distribution of time between arriving jobs

2) distribution of service time

3)average arrival rate

4)average service rate

5) service discipline (the order in which jobs are serviced)

As in all models, when it is impossible or exceedingly difficult to calculate or model one or more of these characteristics, approximations are used. The resulting utility of the model is determined by the accuracy of the approximations used.

2.2 Call Queue

The processes of disk requests arriving to the disk system is actually subordinate to a higher level process. Disk requests are generated by active user calls, and so the manner in which these calls arrive and are serviced must be modeled.This is done with the "Call Queue", which effectively models the entire voice mail system as a queue with jobs defined as incoming calls.

2.2.1 Queue Characteristics

When a call comes into the system, it is assigned a channel over which to communicate. Once the call is finished, the channel is released. The number of channels occupied does not affect the rate at which calls arrive to the system. If 15 out of 16 channels are occupied, the average arrival rate for a call is approximately the same as when no channels are being used.

Since each channel is effectively a server, the system must be modeled as a multi-server queue. The average service time of each channel is simply the average length of a call. Although the performance of the system has some impact on the length of a call, this impact is a small percentage of that length. Basically, call length is a function of typical user behavior. For a given pattern of user behavior, one can estimate the average call length, and for simplicity an exponential distribution of these lengths is assumed. For a fixed number of users, it is also possible to estimate the average rate of calls to the system. The time between these calls is modeled as having an exponential distribution. Thus, the call queue is a multiple server queue with exponential arrivals and exponential service times. A 16 channel system would result in a M/M/16 call queue.

2.2.2 Mathematical Specification

A useful way of visually representing the operation of this queue is the Markov chain (otherwise known as the state transition diagram) that it represents. In this particular Markov chain, the state of the queue simply corresponds to the number of calls in the system (or number of channels active).

If C is the total number of channels in the system, figure 3 depicts the Markov chain that can be used to model this queue:

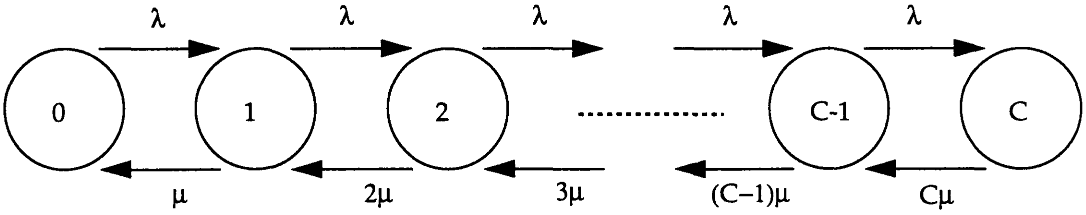

Figure 3. Markov Chain for Call Queue (C channels)

It is assumed that when the system is full (in state C), new calls are discarded with no change to the arrival statistics.

We define the probability of the system having k channels active (being in state k) as π_k. The general formula for a birth-death system states that:

$$\pi_k = \frac{\lambda_0 \cdot \lambda_1 \cdots \lambda_{k-1}}{\mu_k \cdot \mu_{k-1} \cdots \mu_1} \pi_0$$

When we substitute the values for our system, we get:

$$\pi_k = \frac{\lambda^k}{\kappa\mu \cdot (\kappa-1)\mu \cdots 2\mu \cdot \mu} \pi_0$$

This simplifies to:

$$\pi_k = \frac{1}{k!}\left(\frac{\lambda}{\mu}\right)^k \pi_0$$

Because these are probability measures of being in a particular state, if we add them all up they must equal 1. This allows us to calculate π_0.

$$\sum_{k=0}^{M} \pi_k = 1$$

Therefore:

$$\pi_0 = \frac{1}{1 + \sum_{j=1}^{M} \frac{1}{j!}\left(\frac{\lambda}{\mu}\right)^j}$$

Once we have π_0, we can calculate any π_k we want:

$$\pi_k = \frac{\frac{(\lambda/\mu)^k}{k!}}{\sum_{j=0}^{M} \frac{(\lambda/\mu)^j}{j!}}$$

Once we know the values of all the π_k's, we can calculate the average number of calls in the queue, N. This is because, in general, the mean queue length is given by the following equation:

$$N = \sum_{k=1}^{\infty} k\pi_k$$

Therefore, the average number of channels active in the "call queue" is given by:

$$N_c = \sum_{k=1}^{M} \frac{\frac{k(\lambda/\mu)^k}{k!}}{\sum_{j=0}^{M} \frac{(\lambda/\mu)^j}{j!}}$$

This number is important in the models that follow. A channel in use generates disk requests at a certain average rate. Thus, it is necessary to know the steady state average number of channels in service to calculate the average arrival rate of disk requests to the disk queue. If N_c =13 channels, then the average rate of arrivals to the disk system is 13 times the rate that would occur with one channel active.

Thus, even though there may be 16 channels on the system, and thus 16 channels worth of disk requests are "available" to arrive, we cannot model each channel as producing disk requests unless it is actively servicing a call. Thus, in all the following formulas, we define the variable M equal to the average number of channels in service N_c, not the total number of channels in the system C.

2.3 Disk Queue

This section presents three separate models for disk queues. One is for a single disk system, and the other two are used for multiple disk systems. The multiple disk systems are differentiated by disk arrival rates: in the first requests arrive at each disk at the same rate, and in the second the rates are not equal.

All three models use the parameter M in key equations, which is defined to be equal to the value N_c as calculated in Section 2.2 above. In addition all three models use the parameter μ_t, which is defined as the total service time for disk requests. This value includes the time spent at the SCSI bus, and is therefore presented in section 2. 4 below, dealing with the "bus" queue.

2.3.1 Single Disk System

The queue for the disk subsystem is implemented in software as a part of the disk device driver. When a software task makes a request to or from disk, it sends the request to the disk driver and the driver enqueues the request and blocks the requesting task until the request is completed. When the queue is non-empty, the driver takes SCSI commands off the queue and sends them to the disk controller. When the disk controller indicates it is ready for another command, the driver dequeues one and sends it.

2.3.1.1 Disk Queue Characteristics

Unlike the "call queue", the disk queue has only one server: the disk drive. In the interests of simplicity the distribution of both arrival and service times is assumed to be exponential. The service discipline of the disk queue is FCFS (First Come First Serve). In reality, the driver in the present system orders requests by location on disk; however, this can be modeled as FCFS with an adjustment to average seek time to compensate. Since the system only enqueues or dequeues one job at a time, the disk queue is a birth-death system.

One important characteristic of the disk queue is the fact that there is a limited population of jobs available to arrive at the queue. The reason is that any given open voice channel will only have one task making disk requests at a time. This complicates the queue, because the number of jobs in the queue influences the average arrival rate. The more jobs on the queue, the fewer jobs there are available to arrive.

Note: The average service rate for a single disk system must take into account both the time it takes to service the request at the disk itself, and the transfer time across the SCSI and ISA buses. Even so, The average service rate can be obtained by a relatively straightforward calculation involving the speed of drive (seek times and rotational speed), transfer rate of the SCSI bus, transfer rate of the ISA bus, and average job size.

2.3.1.2 Mathematical Specification

Since there are a limited number of jobs available, we model each job as arriving at the disk queue with an individual arrival rate λ. In other words, if there are 6 jobs in the queue, and the maximum number of jobs is 16, then there are 10 jobs left each arriving at a rate of λ. Since we have assumed that the arrivals are exponential processes, the aggregate process is an exponential process with an arrival rate of 10λ. In this case, the parameter λ is defined as the average rate of disk requests for one voice channel.

Figure 4 below shows the Markov chain for this queue, assuming 5 channels. The state is the number of elements in the queue:

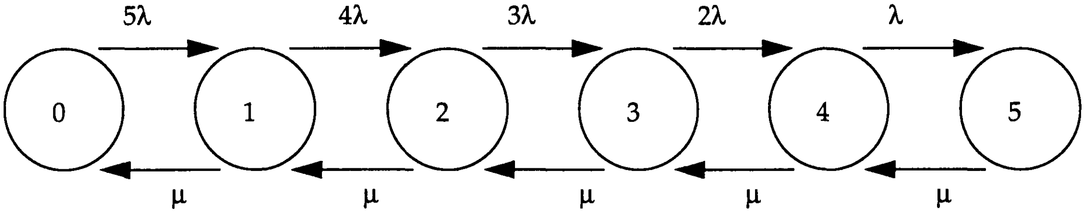

Figure 4. Markov Chain for Disk Queue (5 Channels)

The parameter representing service rate, μ, does not change this time because the service time is independent of number of jobs in the queue.

Given this as the representation of our queue with average number of active channels M, we can calculate the probability of the queue being in any given state recursively, in the same manner as the previous section.

Using the general formula for a birth-death system. we can find π_k:

$$\pi_k = \frac{M!}{(M-k)!} \left(\frac{\lambda}{\mu}\right)^k \pi_0$$

$$\pi_k = \frac{\dfrac{(\lambda/\mu)^k}{(M-k)!}}{\displaystyle\sum_{j=0}^{M} \frac{(\lambda/\mu)^j}{(M-j)!}}$$

In the same way as before, we can multiply the probability of each state by the number of jobs each state represents, and sum them all. This gives us the average number of jobs in the system, N:

$$N = \sum_{k=1}^{M} \frac{\dfrac{k\,(\lambda/\mu)^k}{(M-k)!}}{\displaystyle\sum_{j=0}^{M} \frac{(\lambda/\mu)^j}{(M-j)!}}$$

Unlike the call queue, however, N is not the final objective. From N, we can calculate the average time spent waiting in the queue, T. This is given by Little's Theorem:

$$T = \frac{N}{\lambda_A}$$

The parameter λ_A is the average arrival rate, which in this case is not completely straightforward, because the arrival rate depends on the present state of the system. In a particular state k, there are (M-k) requests available to arrive at the queue. Since each request arrives with average rate λ, the arrival rate is $(M\text{-}k)\lambda$. Thus, the steady state average arrival rate is given by:

$$\lambda_A = \sum_{k=0}^{M} \pi_k\,(M-k)\,\lambda$$

Using the definition of N, we can simplify this to:

$$\lambda_A = (M - N)\,\lambda$$

So, the average time spent in the disk system (including service) is:

$$T_d = \sum_{k=1}^{M} \frac{\dfrac{k\,(\lambda/\mu)^k}{(M-k)!}}{\displaystyle\sum_{j=0}^{M} \frac{(\lambda/\mu)^j}{(M-j)!}} \quad \frac{1}{(M-N)\,\lambda}$$

Therefore, given the average disk request rate for one voice channel, the average disk service rate, and the average number of active voice channels, we can now calculate the average number of jobs in the disk queue and the average delay for the system.

2.3.2 Multiple Disk System - Symmetric Accesses

The multiple disk system is more complicated than the single disk system in several ways. Each disk on the system will have a separate queue. However, an additional element is added in the multiple disk system: the bus server. This part of model represents the usage of both the SCSI bus and the ISA bus to transfer data to and from the disks. No matter how many drives there are, only one drive at a time can be actually transferring data.

The disk subsystem model of a multiple disk system is depicted graphically in figure 5 below.

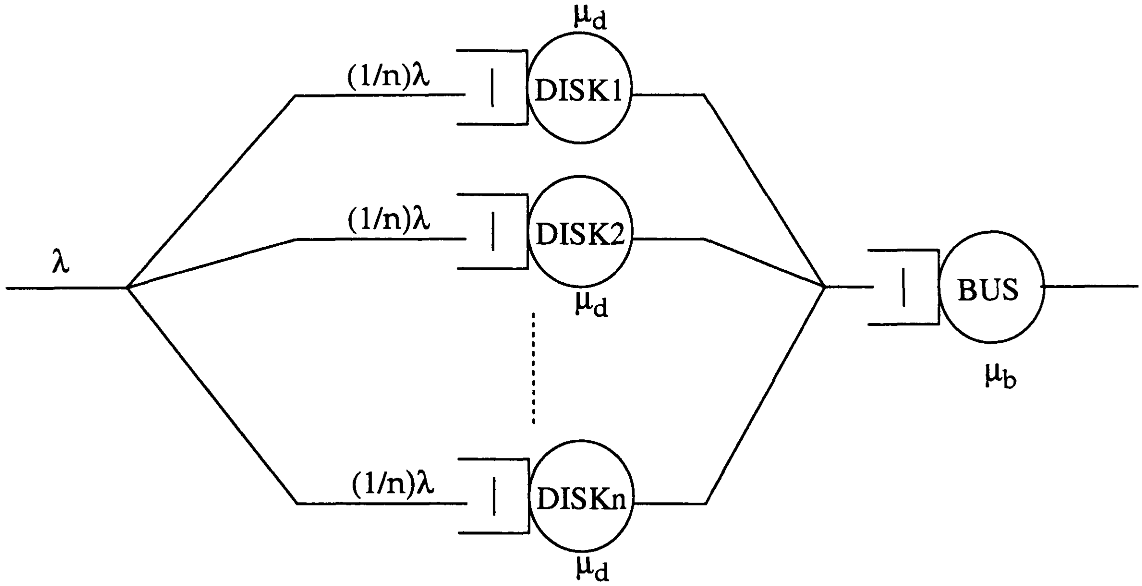

Figure 5. Multiple Disk Subsystem Queuing Model

A multiple disk system has more than one drive mounted on the SCSI bus. There is still only a single controller, a single SCSI bus, and a single driver. When the voice mail system is booted, the software ascertains the number of drives that are attached to the system. Thus, the disk device driver knows how many disks there are and acts accordingly. It creates a separate software queue for each drive, and routes disk requests from other tasks to the appropriate queue. It then sends requests from each queue to the SCSI controller (in arbitrary order). Once it has sent a request from one queue, the driver will not send another request from that queue until the disk system has processed the first request. This ensures that a backlog of requests for one particular drive does not prevent the other drives from being utilized.

When the SCSI controller receives a disk request, it attempts to access the appropriate drive (if the drive in question is not currently busy). If it cannot read/write the data immediately, the disk disconnects from the SCSI bus while it prepares to be accessed. While it is doing this, the controller acknowledges receipt of the first request and asks for another. It can repeat the process with the new request. If a request comes in for a drive that is busy, the controller will wait for that drive to complete before asking for a new request.

Once a disconnected drive is ready, it will wait for the SCSI bus to become free, then reconnect and complete the data transfer.

2.3.2.1 Disk Queue Characteristics

We now have a non-homogeneous population of jobs in the system. These jobs are differentiated by the disks they are trying to access. In other words, a disk request for drive 1 is different from a request for drive 2. In certain cases, different job classes could have different average arrival rates. In the Symmetric Access case, however, it is assumed that it is equally likely for a request to be for any one particular disk. Thus, if the total rate of disk requests is λ and the number of disks in the system is n, the average arrival rate for each job class is $(1/n)\lambda$. The disk service times remain essentially the same as in the single disk case; requests are processed at an average rate of μ_d requests per second.

Once again, both the arrival and service processes are assumed to have exponential distribution. As in the single disk case, there is a limited number of jobs in the system, because only one task per channel can be making a request at any given time.

2.3.2.2 Mathematical Specification

The average arrival rate of disk requests caused by one voice channel is defined as λ. If the number of disks on the system is n, then the average rate of arrivals for each individual disk queue is $1/n$ multiplied by the average arrival rate, λ. The average service rate for each disk is defined as μ_t. This is not simply the inverse of the service time of the disk; as will be shown later, the bus "queue" comes into play.

Modeling the system this way allows us to treat each individual disk queue almost identically to the single disk case. The Markov chain shown in Figure 6 below is similar to the single disk chain, except the λ parameter is scaled by the number of disks in the system.

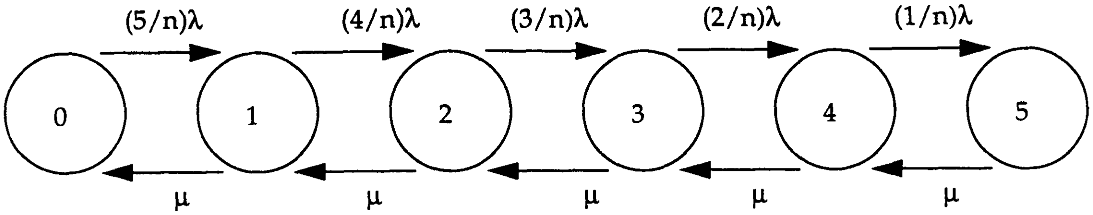

Figure 6. Markov Chain for Disk Queue: Multiple Disks

Because of this, we can use the formulas from Section 2.3.1 to obtain values for N and T, the average length of the queue and average time spent in the system, respectively. We simply replace all occurrences of λ with λ/n, and μ with μ_t:

$$N = \sum_{k=1}^{M} \frac{\dfrac{k\,(\lambda/n\mu_t)^k}{(M-k)!}}{\displaystyle\sum_{j=0}^{M} \frac{(\lambda/n\mu_t)^j}{(M-j)!}}$$

$$T = \sum_{k=1}^{M} \frac{\dfrac{k\,(\lambda/n\mu_t)^k}{(M-k)!}}{\displaystyle\sum_{j=0}^{M} \frac{(\lambda/n\mu_t)^j}{(M-j)!}} \quad \frac{n}{(M-N)\,\lambda}$$

2.3.3 Multiple Disk System - Asymmetric Access

The symmetric access model presented above allows us to calculate several important performance related values for a voice mail disk system with more than one drive. However, it assumes that the software accesses all disks with equal frequency. Unfortunately, this is not the case. A significant amount of frequently accessed data resides exclusively on one disk. Thus, a larger percentage of the total number of I/O requests are routed to this disk. This contradicts the assumption used in the symmetric access model, because the arrival rates for the disk queues are not equal.

In our multiple disk system model, each disk drive has a separate queue implemented in software. In the symmetric access version of this model, it was possible to calculate an average response time that held for every disk queue, because arrival rates were identical. In the nonsymmetric model, however, this is not the case. Each queue may have a different average arrival rate, and so each queue may have different average total delay, T_t, Therefore, the average total delay must be calculated separately for each queue.

In the actual system, the entire database is maintained on disk number zero. Thus, all database disk requests are necessarily routed to disk zero. To attempt to compensate for this, the system software stores other types of data less frequently on disk zero than on the other disks. To approximate this mathematically, we need to define some constants. Let db = the percentage of all disk requests that are database requests. Let z and d be integers such that z/d is the ratio of disk zero non-database requests to disk (1,2...) non-database requests. With these defined, we can find λ_z and λ_d, the arrival rates to disk zero and the remaining disks, respectively:

$$\lambda_z = (db)\,\lambda + \frac{z}{z + d(n-1)}\,(1-db)\,\lambda$$

$$\lambda_d = \frac{d}{z + d(n-1)}\,(1-db)\,\lambda$$

As always, n is the number of disks in the system, and λ is the aggregate average arrival rate for all disk requests (per active channel). With these values, we can obtain N_z and N_d by using the formula for N in Section 2.3.1 and substituting the appropriate value for λ. To find T_z and T_d, we again use Little's law, but there are now two λ_A's:

$$\lambda_{Ad} = (M - N_d)\,\frac{\lambda_d}{n-1}$$

$$\lambda_{Az} = (M - N_z)\,\lambda_z$$

We can then find T_z and T_d, the average total delay for disk zero and the other disks, respectively.

2.4 Bus Queue

We have calculated the average values above, and those in the previous section, based on the total service rate, μ_t. However, we have yet to calculate that value. It has two components: the service rate of the actual disk drive, μ_d, and time spent on the bus or waiting for the bus, T_b. Given these values, we can find the total service rate:

:

$$1/\mu_t = 1/\mu_d + T_b$$

This section develops a queuing model for the bus that will give the average time spent waiting for the bus, T_b.

The bus, as a server, does not have an actual queue per se. Instead, jobs wait to be transferred from the disk buffer of whatever drive they are using. The rate of arrivals to the bus queue is equivalent to the combined throughput of the disk queues. Since (in a stable system) the throughput is equal to the average rate of arrivals, we define the arrival rate to the bus server as follows: Symmetric access-- $\lambda_b = (M - N)\,\lambda$, Asymmetric access-- $\lambda_b = \lambda_{Ad}(n-1) + \lambda_{Az}$.

A job in this case consists of a I/O request (a disk read or write) that has data waiting to be transferred over the bus. The server (the bus itself) processes the job by completing the transfer. Then it can proceed with another request, and transfer more data.

2.4.0.1 Normal Case

In a normal operating situation, the bus queue can be modeled as a simple M/M/1 queue. This is because, in the steady state, the average arrival rate to the bus must equal the combined throughput of all the disks, which in turn equals the average arrival rate to all the disks.

For an M/M/1 queue, the average length of the queue N is given by the simple equation(λ_b will be referred to as λ):

$$N = \frac{(\lambda/\mu)}{1 - (\lambda/\mu)}$$

Once we have N, we can calculate the average time spent waiting in the queue, T. This is given by Little's Theorem:

$$T = \frac{N}{\lambda_A}$$

The parameter λ_A is the average arrival rate, which in this case is simply equal to the combined throughput of all the disk queues, which we recently defined as λ. So, the average time spent in the bus queue is:

$$T_b = \frac{(1/\mu)}{1 - (\lambda/\mu)}$$

Therefore, given the average disk request rate to the system, the average disk service rate, we can calculate the average number of jobs in the bus queue and the average delay due to the bus.

2.4.0.2 Worst Case

In many cases, we are interested in the worst case scenario when dealing with performance. For the bus queue, the worst case situation is a little different from the standard case.

For the bus queue fully utilized, each drive on the SCSI bus must processes requests as if it had an infinitely full queue of requests waiting to be serviced. In this case, requests would "arrive" after they finish being processed by the disk. Since one request begins processing as soon as another is finished, requests arrive to the bus at the rate of μ_d, the service rate of the disks involved.

The most accurate representation of this would be a queue similar to the one used for the disks. In other words, the queue has exponential arrival and service distributions, one server, and a finite population available to arrive at the queue. When a request arrives at the bus queue, the disk it is on can no longer process another job, and so no longer contributes to the arrival process.

The Markov chain for the system would look similar to that of the disk queue, except the number of states is determined by the number of disks, n, rather than the number of channels. In this case λ is equal to the service rate of the disks involved, and μ is the service rate of the bus. The Markov chain is shown in Figure 7 below:

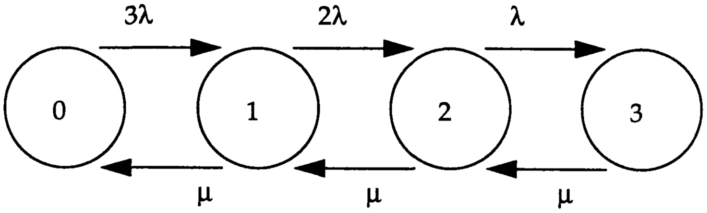

Figure 7. Markov Chain for Bus Queue (3 Disks)

Using the equations from 4.2.6, we replace occurrences of the variable M (number of channels), with the variable n (number of disks). This gives us the average number of jobs in the system, N:

$$N = \sum_{k=1}^{n} \frac{\dfrac{k\,(\lambda/\mu)^k}{(n-k)!}}{\displaystyle\sum_{j=0}^{n} \frac{(\lambda/\mu)^j}{(n-j)!}}$$

From this, we can calculate the average delay due to the bus queue, T, as follows:

$$T_b = \left(\sum_{k=1}^{n} \frac{\dfrac{k\,(\lambda/\mu)^k}{(n-k)!}}{\displaystyle\sum_{j=0}^{n} \frac{(\lambda/\mu)^j}{(n-j)!}} \right) \left(\frac{1}{n\lambda \;-\; \lambda \displaystyle\sum_{k=1}^{n} \frac{\dfrac{k\,(\lambda/\mu)^k}{(n-k)!}}{\displaystyle\sum_{j=0}^{n} \frac{(\lambda/\mu)^j}{(n-j)!}}} \right)$$

2.4.0.3 Worst Case: Asymmetric

If all the disk drives on the bus do not have the same service rate, the situation becomes vastly more complicated. If we look back at figure 7, we see that the Markov chain for the bus queue is structurally identical to that of the disk request queue in figure 6. This structure depends on the symmetry of disk request service rates. Recalling the specifics of the bus queue, each disk in the system is considered to be a potential job which will "arrive" at the queue when an I/O request is completed by the drive. Requests arrive with a rate equal to the service rate of that drive, because there is always a request waiting to be processed. When the service rates for all the disks are equal, we can treat the disks as identical to one another and are not concerned with which disk is being utilized. Thus, when there are three disks free, we know that a single request arrives at the bus queue at a rate equal to three times the service rate for a single disk. With two disks free, requests arrive at twice the rate of a single disk. For this reason, the bus queue was modeled as an M/M/1/L/M queue, mathematically identical to disk request queue.

With asymmetrical request rates, however, this model no longer holds. Since the service rates for each disk are different, the amount each disk contributes to the total arrival rate is different. Thus, when two disks are free, we cannot calculate what the average arrival rate for a request is unless we know *which* two disks are free. Therefore, we can't model the bus queue as a M/M/1/L/M queue. We must take a step back and look at how we constructed the equation for the M/M/1/L/M queue in the first place. The Markov chain for the asymmetric bus queue would have the structure presented in Figure 8 below.

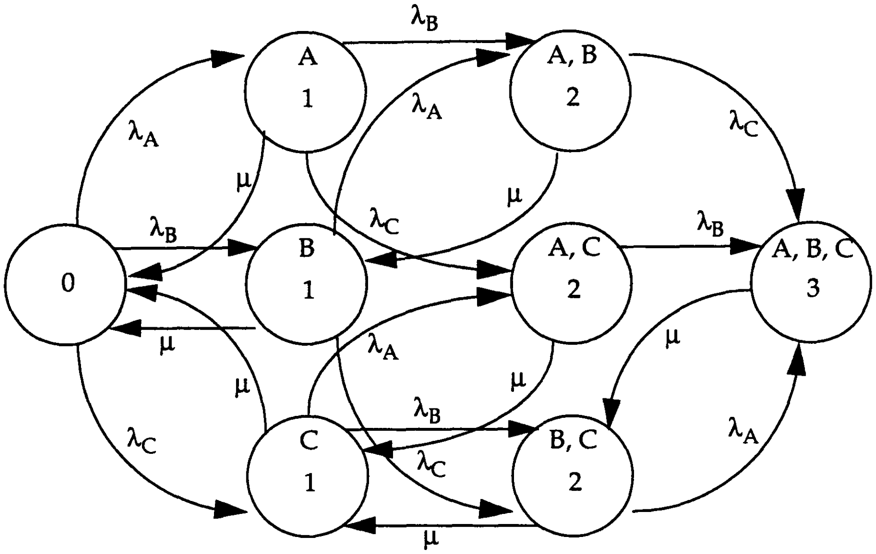

Figure 8. Markov Chain for Bus Queue (3 Disk Asymmetric)

Figure 9 shows a markov diagram for a 3 disk system in which the disk are labeled A, B, and C. These disks contribute unequal arrival rates λ_A, λ_B, and λ_C respectively. The bus server has service rate μ, and when there is more than one drive waiting for the bus, the drives are serviced in alphabetical order. This is consistent with the manner in which SCSI bus arbitration is handled. Each state is labeled with a number and a sequence of letters. The number indicates how many disks are currently waiting to be serviced by the bus. The letters indicate *which* disks are waiting to be serviced.

The transitions between states are relatively simple. For instance, from state 0, the system goes to state 1A at rate λ_A, to state 1B at rate λ_B, and to state 1C at rate λ_C. From state 1A the system can go to state 2AB at rate λ_B, or state 2AC at rate λ_C. It also returns to state 0 at rate μ. A Markov chain can be constructed in this manner for a system with an arbitrary number of disks.

Unfortunately, this Markov chain no longer represents a birth-death system. Therefore, we can no longer use the general birth-death equations to describe the behavior of this queue. Instead, we must rely on the basic mechanics of Markov Chain mathematics and attempt to find the steady state behavior of the system. If we can accomplish this, we can calculate the relevant values, such as mean queue length and mean delay.

The first step in solving for steady state probabilities is to put the system in matrix form. Since we are dealing with a continuous time system, we must compute the system's *transition rate matrix.* The non-diagonal elements of the matrix describe the rate of transition from one state to another. An entry in row i and column j denotes the rate of transition from state i to state j. The diagonal elements of the matrix are equal to the additive inverse of the sum of the rates of transition out of a given state. In other words, the entry in row i, column i is zero minus the sum of the rates of transitions out of state i. Figure 9 below shows the transition rate matrix for the Markov chain in figure 8.

$$\begin{array}{r|cccccccc}
 & 0 & 1A & 1B & 1C & 2AB & 2AC & 2BC & 3ABC \\
\hline
0 & S_0 & \lambda_A & \lambda_B & \lambda_C & 0 & 0 & 0 & 0 \\
1A & \mu & S_1 & 0 & 0 & \lambda_B & \lambda_C & 0 & 0 \\
1B & \mu & 0 & S_2 & 0 & \lambda_A & 0 & \lambda_C & 0 \\
1C & \mu & 0 & 0 & S_3 & 0 & \lambda_A & \lambda_B & 0 \\
2AB & 0 & 0 & \mu & 0 & S_4 & 0 & 0 & \lambda_C \\
2AC & 0 & 0 & 0 & \mu & 0 & S_5 & 0 & \lambda_B \\
2BC & 0 & 0 & 0 & \mu & 0 & 0 & S_6 & \lambda_A \\
3ABC & 0 & 0 & 0 & 0 & 0 & 0 & \mu & S_7
\end{array}
\qquad
\begin{array}{ll}
S_0 & = -(\lambda_A + \lambda_B + \lambda_C) \\
S_1 & = -(\lambda_B + \lambda_C + \mu) \\
S_2 & = -(\lambda_A + \lambda_C + \mu) \\
S_3 & = -(\lambda_A + \lambda_B + \mu) \\
S_4 & = -(\lambda_C + \mu) \\
S_5 & = -(\lambda_B + \mu) \\
S_6 & = -(\lambda_A + \mu) \\
S_7 & = -(\mu)
\end{array}$$

Figure 9. Transition Rate Matrix

From this matrix, it is possible to calculate the steady state probabilities of being in each state. To understand how, it is necessary to understand what the transition rate matrix actually means. If we have a state probability vector, **p**(t), which describes the probability of being in each state at time t, the following equation describes the relationship between the transition rate matrix, **Q**, and **p**(t):

$$\mathbf{p}(t)\,\mathbf{Q} = \frac{d}{dt}\,\mathbf{p}(t)$$

Multiplying the probability state vector by the transition rate matrix gives us the instantaneous rate of change of probabilities at time t. Since we are interested in the steady state probabilities of the system, this is a very convenient equation. We merely need to find the state probability vector for which the instantaneous rate of change is equal to zero, and that is the steady state probability vector. In other words, the steady state probability vector, π, satisfies the following equation:

$$\pi\mathbf{Q} = \mathbf{0}$$

So, if we solve for π, then we have accomplished our goal of calculating the steady state probabilities of the system. To do this, we simply multiply out the matrix equation to obtain the simultaneous system of linear equations shown in figure 10.

$$[\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7, \pi_8] \begin{bmatrix} S_0 & \lambda_A & \lambda_B & \lambda_C & 0 & 0 & 0 & 0 \\ \mu & S_1 & 0 & 0 & \lambda_B & \lambda_C & 0 & 0 \\ \mu & 0 & S_2 & 0 & \lambda_A & 0 & \lambda_C & 0 \\ \mu & 0 & 0 & S_3 & 0 & \lambda_A & \lambda_B & 0 \\ 0 & 0 & \mu & 0 & S_4 & 0 & 0 & \lambda_C \\ 0 & 0 & 0 & \mu & 0 & S_5 & 0 & \lambda_B \\ 0 & 0 & 0 & \mu & 0 & 0 & S_6 & \lambda_A \\ 0 & 0 & 0 & 0 & 0 & 0 & \mu & S_7 \end{bmatrix} = [0, 0, 0, 0, 0, 0, 0, 0]$$

$$S_0\,\pi_1 + \mu\pi_2 + \mu\pi_3 + \mu\pi_4 = 0$$
$$\lambda_A\pi_1 + S_1\pi_2 = 0$$
$$\lambda_B\pi_1 + S_2\pi_3 + \mu\pi_5 = 0$$
$$\lambda_C\pi_1 + S_3\pi_4 + \mu\pi_6 + \mu\pi_7 = 0$$
$$\lambda_B\pi_2 + \lambda_A\pi_3 + S_4\pi_5 = 0$$
$$\lambda_C\pi_2 + \lambda_A\pi_4 + S_5\pi_6 = 0$$
$$\lambda_C\pi_3 + \lambda_B\pi_4 + S_6\pi_7 + \mu\pi_8 = 0$$
$$\lambda_C\pi_5 + \lambda_B\pi_6 + \lambda_A\pi_7 + S_7\pi_8 = 0$$

Figure 10. Equation System for Steady State Probabilities

We now have a system of 8 linear equations in 8 variables (π_1 through π_8). These equations are not linearly independent but can be solved in terms of a single variable. Since we are dealing with probabilities, all the π's must sum to 1. We can use this fact to uniquely solve the system, and the result is the steady state probability vector.

Now that we have the steady state probabilities, we can compute the average queue length in the system, and thus the average delay. If we recall, in a simple birth-death queue, the mean queue length, N, is given by the following formula:

$$N = \sum_{k=1}^{\infty} k\pi_k$$

Although the principle is the same, this equation cannot be used directly in this particular case. In a birth-death queue, state k is simply the state in which the queue has k jobs waiting in it. It is for this reason that formula above holds. Therefore, we simply define for our queue a series of variables that have the desired characteristic: Let P_k be defined as the set of states in which our queue has k jobs waiting. Then, we will define π_k' to be the steady state probability that the system has k jobs waiting. The mean queue length N would be given by the following equation:

$$N = \sum_{k=1}^{\infty} k\pi_k'$$

We can easily calculate π_k' because it is merely the sum of the steady state probabilities of each state in the set P_k. Using the example above, we can see the following:

P_0 = [state 0]	$\pi_0' = \pi_1$
P_1 = [state 1A, state 1B, state 1C]	$\pi_1' = \pi_2 + \pi_3 + \pi_4$
P_2 = [state 2AB, state 2AC, state 2BC]	$\pi_2' = \pi_5 + \pi_6 + \pi_7$
P_3 = [state 3ABC]	$\pi_3' = \pi_8$

Finally, we use π_0' through π_3' to calculate N. Then, using Little's law, we use N to calculate T_b.

2.4.1 System Parameters → Model Parameters

The system parameters given in section 3 are not in any of the equations of the working model so far; in fact they are hardly even mentioned. However, the system parameters will map onto model parameters in some way or another, allowing useful system comparisons to be made.

The parameters of our mathematical model are:

1) Number of channels

2) Number of disks

3) Average call length

4) Average disk service rate

5) Average bus service rate

6) Average rate of call arrivals

7) Average rate of disk requests per active channel

The first three parameters are simple enough: disks and channels are direct system parameters, and average call length is fixed based an typical behavior.

The fourth parameter is not so direct, but still relatively simple. The average disk service time is given by the following formula:

- Average Service Time = Average Seek Time + Average Latency + #bits transferred * media transfer rate

Parameter number 6, average call rate, can be set arbitrarily, based on how many people the system serves. Here, we set the call rate so that the system has a specified percentage of blocked calls.

The last parameter, and certainly one of the most important, is a simple number that is very hard to come by. It is difficult to measure such a number on a real system, which is one of the several reasons a simulator was constructed.

Finally, the fifth parameter, bus service rate, deserves some discussion. In reality, the mechanics of a data transfer are a little bit more complicated than they have been shown to be thus far. The path that data follows during a disk access (read or write) is shown below in Figure 11.

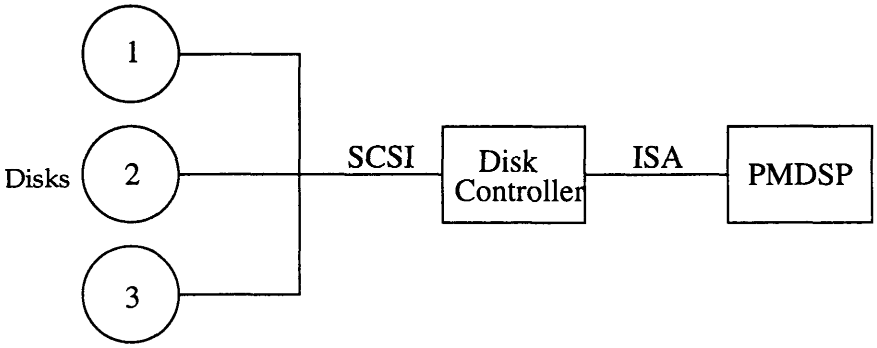

Figure 11. Data Transfer Path

The software queues located in the disk driver actually hold pointers to SCSI command blocks, which specify the disk and memory addresses to be involved in the data transfer. As discussed above, when one of these jobs is processed the appropriate disk will disconnect from the bus while the disk head moves to the correct address. When data is ready to be transferred, the drive will wait for the SCSI bus to be free and then attempt to reconnect. Once the drive has reconnected, the data will be transferred.

The transfer is carried out by the disk controller, which becomes the ISA bus master. There is a 16 byte bidirectional FIFO cache on the MSC card that acts as an interface between the ISA bus and the SCSI bus. The controller uses burst mode in a 4 microseconds on, 4 microseconds off pattern. In the "on" period, the controller either transfers the contents of the FIFO (up to 16 bytes) across the ISA bus, or transfers 16 bytes from memory to the FIFO. This depends upon whether the disk access being processed is a read or a write. In the 4 microseconds "off" period, the CPU has control of the ISA bus. Thus, the CPU is assured time to access system memory or other devices on the ISA bus.

Due to the mechanics of the data transfer, the "bus transfer rate" is more complicated than just the SCSI bus bandwidth. It is a function of the disk transfer rate, SCSI bus bandwidth, the ISA bus bandwidth, the disk controller bandwidth, and the disk controller latency. A simple and fairly accurate approximation of this number is simply the minimum of all the above rates.

3.0 Simulational Model

3.1 Introduction

Although most performance models rely on mathematical queuing models, many times these equations and formulas require restrictive assumptions and cannot capture the complexity of the system being modeled. They are concise and elegant, but for those very reasons cannot always be applied to real world systems that are neither concise nor elegant.

Throughout the mathematical model, both arrival and service processes were assumed to have exponentially distributed times. Without such assumptions, the model would become significantly more complicated, if possible at all. However, the actual system is not obliging in this respect; neither the arrival nor service process is exponential. In the case of service, the times are more closely modeled as a truncated Gaussian process. In the case of the arrivals, the distribution is more complicated. It is a merger of streams of requests from each channel, which are somewhat random but deterministic to high degree. This makes the accuracy of the mathematical models questionable.

Another need for the simulator exists: at least one important queuing model parameter is not readily available without simulation- the average rate at which an active voice channel makes requests to the system. Without a reasonably accurate number for this parameter, the mathematical model has little utility.

3.2 Architecture Overview

Central to the understanding of this program is the fact that it is a **time based** simulator. It keeps track of time through a global variable (variable: ptime). Each execution of the main loop covers one unit of time (in this case, one millisecond). When the pass is complete, the time is incremented and the loop begins another execution. Each part of the program does its work one time unit at a time, most of them through the use of counter variables. Throughout this document, it is assumed that work is divided over multiple executions of the main loop, and this may not be mentioned explicitly.

Upon start-up, the program reads input data from the specified files and initializes the global variables. It the begins execution of the main loop, which calls seven functions during each pass. The heart of the simulator, however, is contained in just two of these functions. One function implements the Request Generator (function: req_gen). The other implements the Request Processor (function: do_process).

The Request Generator is responsible for simulating disk traffic to the rest of the simulator. To obtain accurate results, the Request Generator attempts to issue disk requests in a manner similar to a system under normal load. In order to this, it is functionally divided into two parts: the Call Generator (function: call_gen), and the Call Processor (function: do_call_process). The call generator initiates calls at a specified average rate, and assigns them to a free channel. Then, the Call Processor simulates these calls, one channel at a time, issuing requests when necessary.

When a request is issued, it is placed on a disk queue for one of the disks in the system. These queues are effectively the interface between the Request Generator and the Request Processor. It is possible to almost completely change the Generator without affecting the Processor. In fact, this feature was used to validate the Request Processor portion of the program by using a much simpler piece of code to place requests on the queue.

A small piece of code referred to as the Controller searches for free disks. If there is a request enqueued for a free disk, it sets up the Disk Process, which is from that point on dealt with by the Request Processor.

The Request Processor basically acts as a state machine controller, with the state being stored in the Disk Processes. The code cycles through the Disk Process for each disk. Depending on the current state of the process, the type of request, and the simulation time, action is taken. A simple example: if the Process is in state Seek, and the disk head has arrived at the correct cylinder, the Request Processor will change the state to Rotate so that the appropriate sector comes under the disk head.

When the Request Processor has finished a particular request, it records the vital statistics of the system's performance during the execution of that request in a list of data.These statistics have been recorded throughout the course of the requests processing, by simply storing the simulation time at certain points and comparing them to obtain timing results. When the simulation is finished, the Data Processor (function: process_data) writes the results to two output files (file: sim.out, delay).

The other four functions called in the main loop take care of various small, but important, tasks. One keeps track of the rotation of the disks (function: do_rotation). Another is responsible for bus arbitration when more than one device wants to use the bus (function: do_arbitration). A third takes care of the elevator ordering algorithm that is used by the voice mail to increase efficiency (function: do_elevator). The final function takes samples of system values such as queue lengths so that the Data Processor can calculate average values (function: total).

Figure 12 below graphically illustrates the program's high level flow.

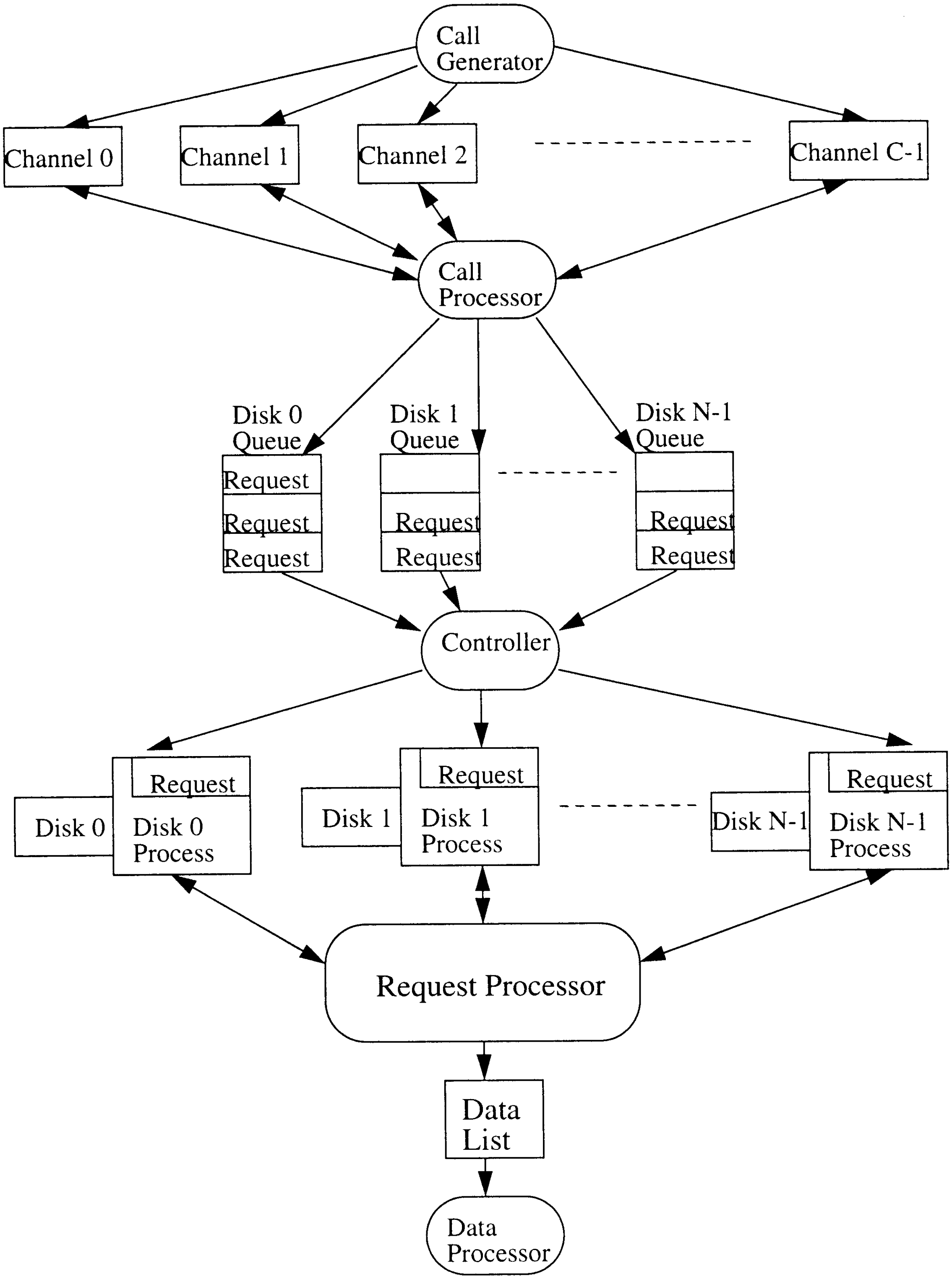

Figure 12. High Level Program Flow

3.2.1 Request Generator

The Request Generator (function: req_gen) is responsible for generating the disk requests that the Request Processor processes. The code is broken down functionally into two pieces: the Call Generator (function: call_gen), and the Call Processor (function: do_call_process)

The Call Generator's basic function is to initiate user calls to the system at an average rate of CALL_RATE. The value of this parameter is read from a stat file at the beginning of program execution. A call is initiated every time a counter (variable: nctime) reaches zero. When the call is initiated, the Call Generator calculates a time until the next call (function: next_call_time), which is an exponentially distributed random number based on the parameter CALL_RATE. Since the time is given in simulator time units, the counter variable is set equal to this value and is subsequently decremented once each execution of the main loop until it equals zero. This continues until the simulation is finished.

When a call is initiated, a free channel is assigned to handle it (function: free_chan). If there are no free channels available, a blocked call is recorded. Otherwise, the program chooses a particular type of call to implement (function: get_call), selecting from a table of calls (array: call_list[ ]) that was initialized from a file. The function chooses from this table based on the probabilities corresponding to those calls, given in another table (array: cprob[ ]). Once a call has been selected, a pointer to it is stored in a data element (structure: call_proc) that keeps track of calls. One such structure is stored for each channel in the system, in a table (array: channel[ ]) indexed by channel number. Next, the state of the channel is set to notify the Call Processor that the call is just beginning.

From there, the Call Processor (CP) takes over. The status of each channel is checked. Based on that status, a particular function is called. If the channel is inactive or waiting for a request to be processed by the disk system, then nothing is done. If a channel is just beginning to service a call, various variables are set appropriately and the state is changed so that the CP knows the channel is now issuing requests. If a channel is currently issuing requests, the CP calls the function issuing_proc.

The function issuing_proc's basic purpose is to issue requests when necessary according to the call that is currently being handled. In the simulator, a "call" is an array of "call phases". Each call phase has three parts: number of requests, type of requests, and average time between requests. For some phase types, the time predetermined, and the stored value is ignored. There are also two special types of call phases that allow the measuring of elapsed time between two points in a call. All the calls in the call table are read from a file at the beginning of program execution, allowing for maximum flexibility.

A request is generated (function: gen_req) every time a counter (variable: nrtime) reaches zero. The function will check to see if there are any more requests left in the current phase (element: channel[].remaining). If so, it resets the counter to equal the time until the next request (function: next_req_time), and changes the channel's status so the CP will know it is waiting for a request to be processed. It then decrements the number of requests remaining in the current phase. If, on the other hand, there were no more requests remaining in the current phase, the procedure advances to the next call phase. If there are no more call phases, the call is finished and the channel is released. Otherwise, the number of requests remaining is set equal to the number of requests specified in the new call phase, and the counter and channel status are set in the same manner as above.

When it is time to issue a request, the function issuing_proc calls the function gen_req, which actually creates the request and puts it on the queue. It checks the type of the current call phase and uses that to decide what type of request to generate. It relies on subroutines to get random LBA's, disk numbers, and other aspects of the request. In the current implementation, the types and characteristics of call phases are as follows:

<u>Database</u>

Disk #: 0
Length: 1
Operation: random
Inter-request time: 10 ms

Play/Record

Disk #: First req- random | Thereafter - same as first
Length: 7.5 Kbytes
Operation: Read / Write
Inter-request time: First req- 10 ms | Thereafter - 2.5 seconds

Vocab

Disk#: random
Length: 2 Kbytes
Operation: Read
Inter-request time: .30 seconds

Other

Disk#: random
Length: 5.5 Kbytes (Uniformly distributed, 1-11)
Operation: random
Inter-request time: as specified

Note: Times or lengths with distributions given are averages. Also, Inter-request time is counted from the time one request is finished until the next is issued.

3.2.2 Request Processor

The Request Processor (function: do_process) is responsible for simulating the disks servicing requests that have been set up in the Disk Processes.

The RP's basic architecture resembles a state machine. The top level procedure invokes a function once for each disk in the system (function: do_1_process). This function checks the state of the request's processing, and calls the appropriate sub-function based on that state.

Each disk in the system has a Disk Process (DP) data structure (structure: process) that is used to hold state information. These structures are kept in a table (array: pprocess[ ]) indexed by disk number. The DP holds several things: A) a copy of the request being serviced (element: pprocess[ ].r), B) the name of the state the process is currently in (element: pprocess[ ].state), C) a state counter (element: pprocess[ ].statecnt), and D) four data count variables that keep track of the simulated location of the data for the request. The data can be stored on the disk media (element:pprocess[ ].media), in the disk cache(element: pprocess[ ].dcache), in the memory (element: pprocess[ ].memory), or in the controller cache (element: pprocess[ ].ccache). In the current implementation only the first three are used. In many cases, data will be stored in two of the three locations simultaneously (during transfers).

The state counter and the four data count variables are used to keep track of the time spent in each state, and when to make transitions. The counter is used for states that are not involved with transferring data, namely Seek and Rotate. The counter is initialized upon entrance to one of these states, and is decremented until it reaches zero, when a state transition is made. For states that transfer data, the data count variables are adjusted each time step depending on what transfer is taking place. For instance, during ReadM, or read from media, the disk is reading data of the physical media and storing it in the on disk cache. Thus, depending on the internal transfer rate of the disk involved, data is subtracted from the media count and an equal amount added to the disk cache count. At all steps of the process, the RP knows where the data is. This allows state transitions to be made at the correct times; i.e. when the cache is full or there is no more data to be read.

When a request is finished, a procedure (function: finish) is called to log the performance data in a data list for the appropriate disk. These lists are simply stored in a table (array: pdata[ ]) indexed by disk number.

A state transition diagram is given in Figure 13.

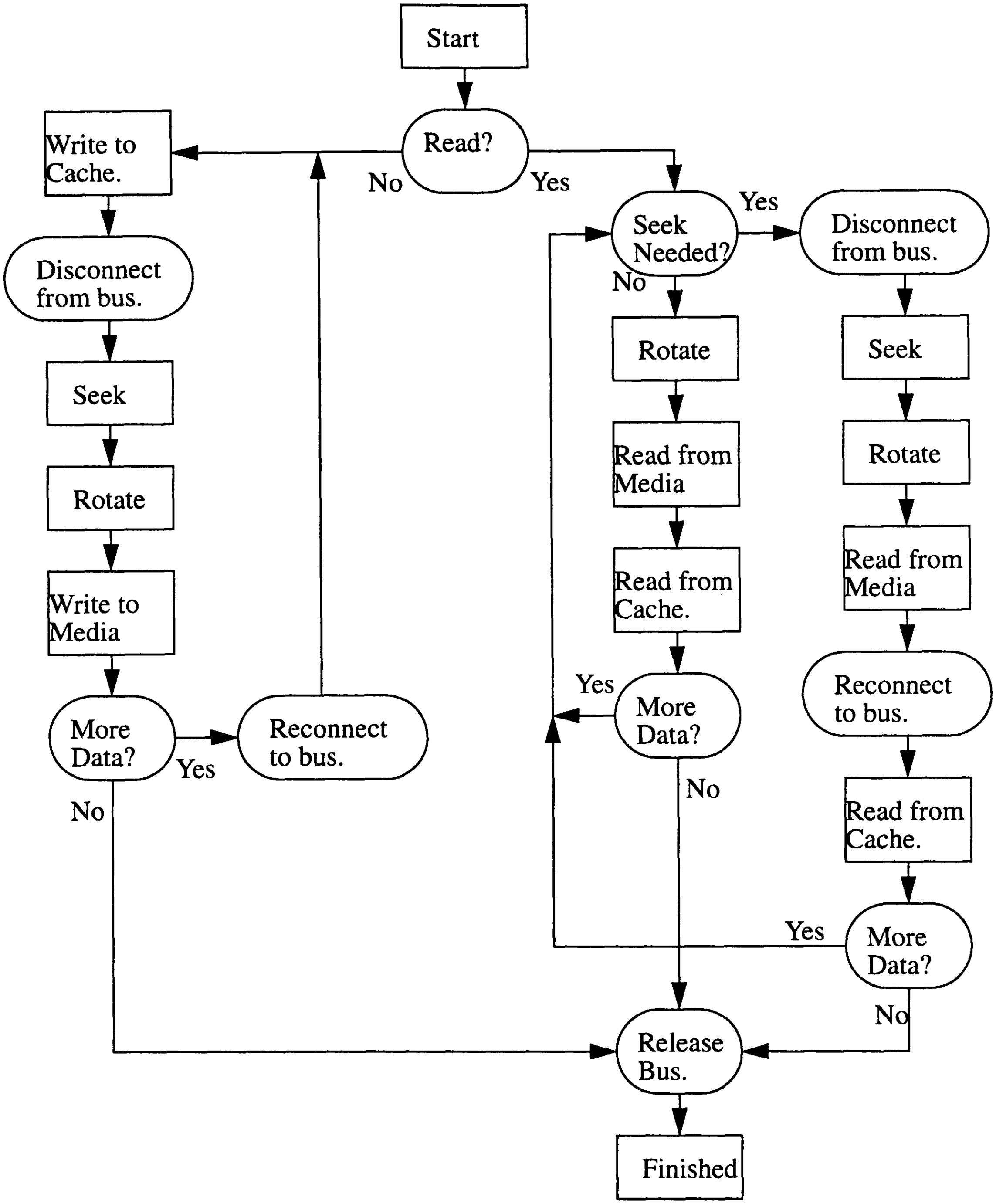

Figure 13. State Transition Diagram

3.2.3 Other Functions

There are several other important pieces of code that are not functionally a part of either the Request Processor or the Request Generator.

3.2.3.1 Arbitration

Arbitration occurs when more than one disk attempts to access the bus at the same time. In this simulation, the function do_arbitration is executed each time step, and assigns bus uses in all cases (even if only one disk is attempting to connect).

The procedure first checks that the bus is free. If it is not, nothing further can occur and the function returns. If it is free, the program checks the status of the controller. If the controller wants the bus, it has first priority. If it does not, the procedure simply checks each disk in turn. The first one that wants the bus wins. The disk (or the controller) that has assumed control of the bus is notified through a status variable related to each device. If a device does not win the bus, it simply continues to try until it does.

3.2.3.2 Elevator Algorithm

The elevator algorithm (function: do_elevator) is implemented by maintaining two separate queues for each disk. These queues are simply stored in a two dimensional table (array: pqueue[ ][ ]) indexed by disk number. At any given time, one queue will be receiving (enqueuing) requests from the Request Generator, and the other will be issuing (dequeuing) requests to the Request Processor. Each disk has a binary selector variable(array: qout[ ]) that keeps track of which queue is performing which function.

Whenever the RP adds a request to the input queue, it uses one of two functions: insert_ascending, or insert_descending. Both perform the obvious function, ordering requests as they are enqueued. The appropriate function is used based on a table of binary direction selectors (array: dir[ ]), one for each disk. This variable keeps track of which direction the queue should be ordered.

Once the output queue is empty, the program switches the queues around, so that the fully ordered input queue now becomes the output queue. The RG then begins to enqueue requests in the new input queue, ordering them in the opposite direction, while the RP works on emptying the other queue. In this manner, the disk head moves from the inside of the platform to the outside and back, minimizing the large seek times.

3.2.3.3 Data Processing

The data processing is performed at the end of the simulation by the function process_data. The first thing this function does is to traverse the lists of primary simulation data stored in a table (array: pdata[ ]). The table stores a list of data (structure: data_list) for each disk. These lists are made up of data elements (structure: stat) that contain data for one processed request. The elements contain 4 numbers: seek time, latency, time spent in queue, and time spent in service (disk + bus). By traversing these lists, the program can calculate averages for these values, and some composite values, for each disk. Also, by dividing the number of requests by the total simulation time, it can calculate request arrival rates for each disk.

There are several other data structures kept during the simulation. One is list of integers (variable: bustimes) representing delays due to waiting for the SCSI bus. Another list keeps track of call durations. The simulator stores an integer equal to sum of the number of channels active at each millisecond; dividing this number by the simulation time gives average number of channels active. Finally, the number of blocked calls during the course of the simulation is recorded.

The above averages and statistics are written to the file "sim.out". Optionally, a variable in the stat file allows the program to write only the average response time across all disks. This allows easy viewing of large numbers of simulation runs.

Once process_data has done all this, it calls the function do_delays before returning. This function is responsible for correlating the delay data recorded by the Call Processor from the use of "delay points". Delay points are two special call_phases; one specifies the start of the delay period, the other specifies the end. The Call Processor simply records the delays between any pair of delay points it finds, and stores them in a list (variable: delays).

The program first traverses the list and sorts the data into a one thousand bracket histogram. This is simply done by dividing the delay by two, rounding it off and using the resulting number to index an array element that is incremented. Since the delays are recorded in milliseconds, the histogram brackets are each two milliseconds wide.

Once this is done, the program uses seven percentages specified in the stat file to determine where these delays lie. For example, if the first percentage specified is 99, the program will find a time that approximately 99% of the delays are less than. It does this for each of the seven percentages, so that an accurate view of the delay profile is obtained. These numbers are written to the file "delay".

3.2.3.4 Function Dependency Diagrams

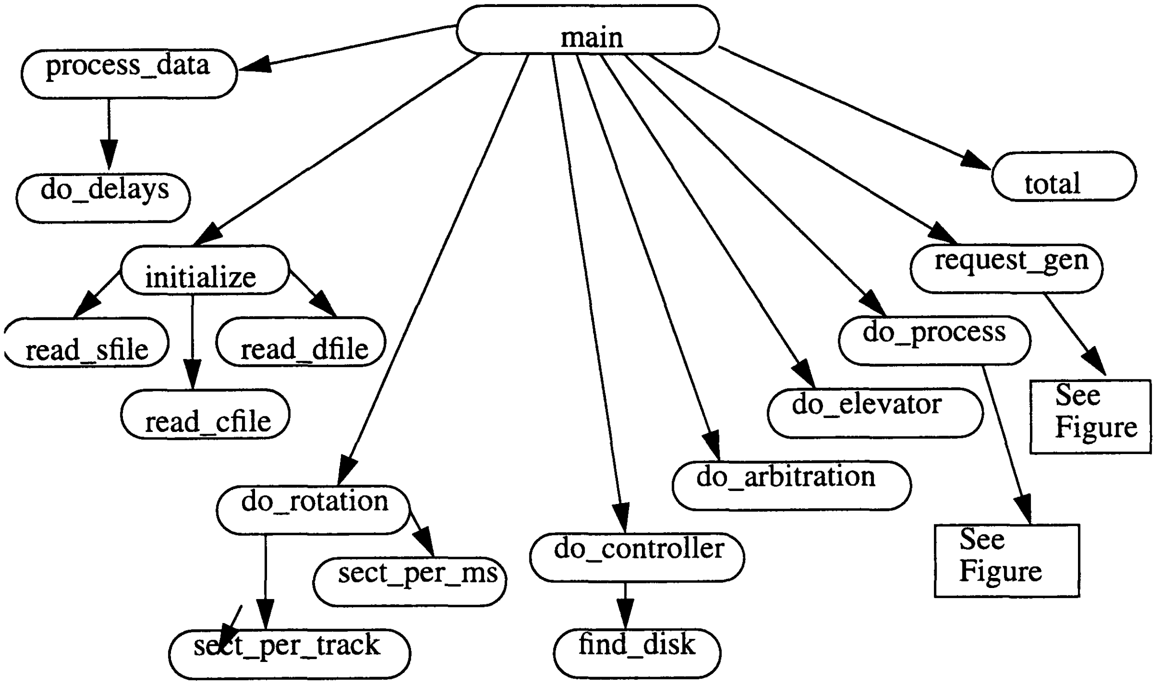

Figure 14. Main Dependency

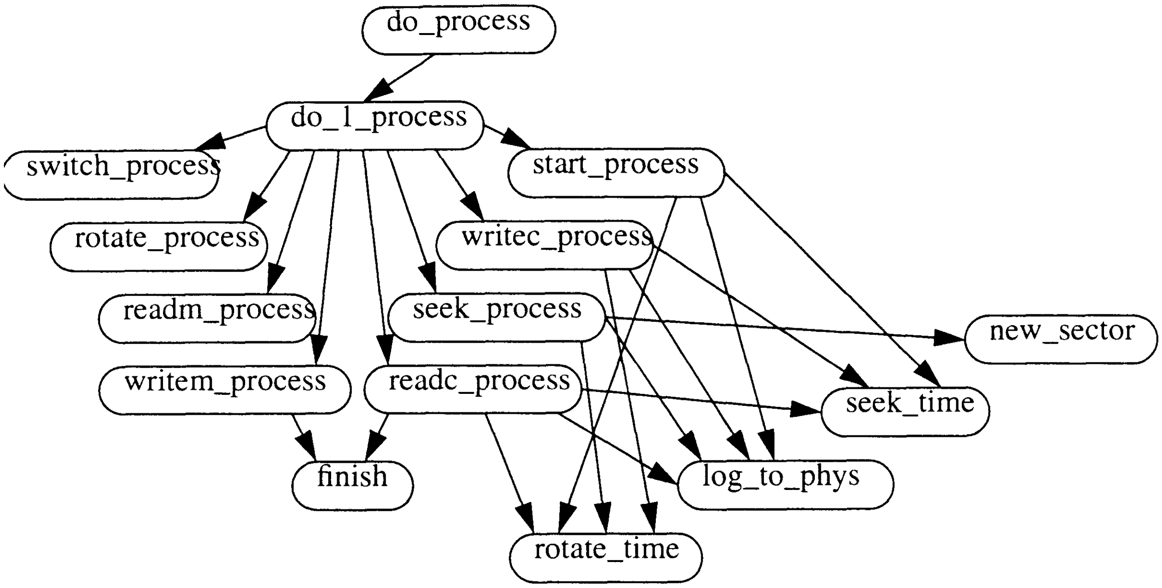

Figure 15. Do_Process Dependency

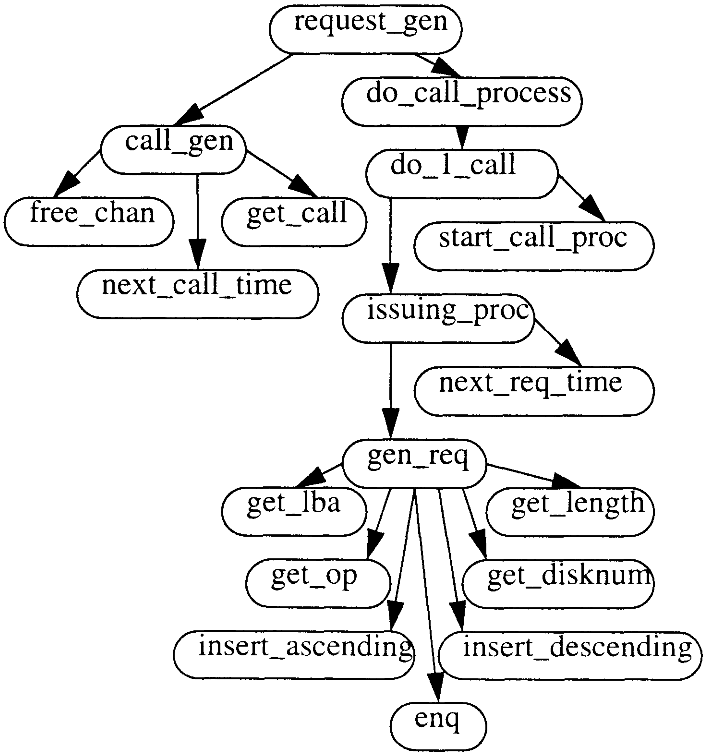

Figure 16. Request_Gen Dependency

4.0 Simulator Verification

4.1 Introduction

To verify the simulator, various outputs of the program were compared with measurable variables in the actual system. The voice mail system currently uses two disk drives: the relatively slow Lee drive, and the faster Lightning drive. For both of these drives, the simulator produced averages very close to the manufacturers specifications.

Another verification checks on internal consistency. The simulator reports the average delay in the queue, the average arrival rate of disk requests to the queue, and the average length of the queue. As mentioned previously in the mathematical model, Little's Law states that the product of the first two must equal the third. By making some test simulator runs, this was checked; in all cases, the law held.

These tests, although somewhat helpful, are not quite enough. One of the primary outputs of the simulator is average total response time, and it needs to be verified that this number is reasonably accurate. Thus, measurements of average response times in the real system need to be made and compared with the simulator output.

The simulator measures total response time from the moment the software issues a disk request to the moment it is transferred across the SCSI bus into main memory. Unfortunately, it is not possible to measure this value in the real system.

A simple program was run on a test system that created multiple tasks which sent requests to the queue. The program creates a variable number of tasks, each of which sends requests of fixed length to the disk driver. Once the outstanding request is processed, the program issues another. Thus, each task has one request outstanding at any given time. The program measures the time each request took from the moment it was issued to the moment it was finished.

In the test scenario, the program was run with 1 through 4 disks on the system, varying the number of tasks, or "channels", from 8 to 56 each time. This was performed with both the Lee and Lightning drives.

The simulator could not be compared to those results without modification; it was designed to simulate the actual system. The difference between the actual system and the test system was the manner in which disk requests were issued. In the actual system, requests are issued in response to incoming user calls. In the test system, each "channel" simply issues requests to the disk system one after another. The simulator was modified to mimic this behavior.

Note: Because the following results do not test the Request Generator, it was tested separately to see that it produces requests in the way it should.

4.2 Original Run

The results of the first tests of the simulator are shown below. Each figure has four graphs, with one for each possible number of disks. The graphs plot the number of channels a system has versus the average response time of the disk requests. The simulated results are shown overlaid on the actual results. Figure 17 shows the results for the Lee drive, Figure 18 for the Lightning drive, and Figure 19 for the Lightning drive with elevator algorithm disabled.

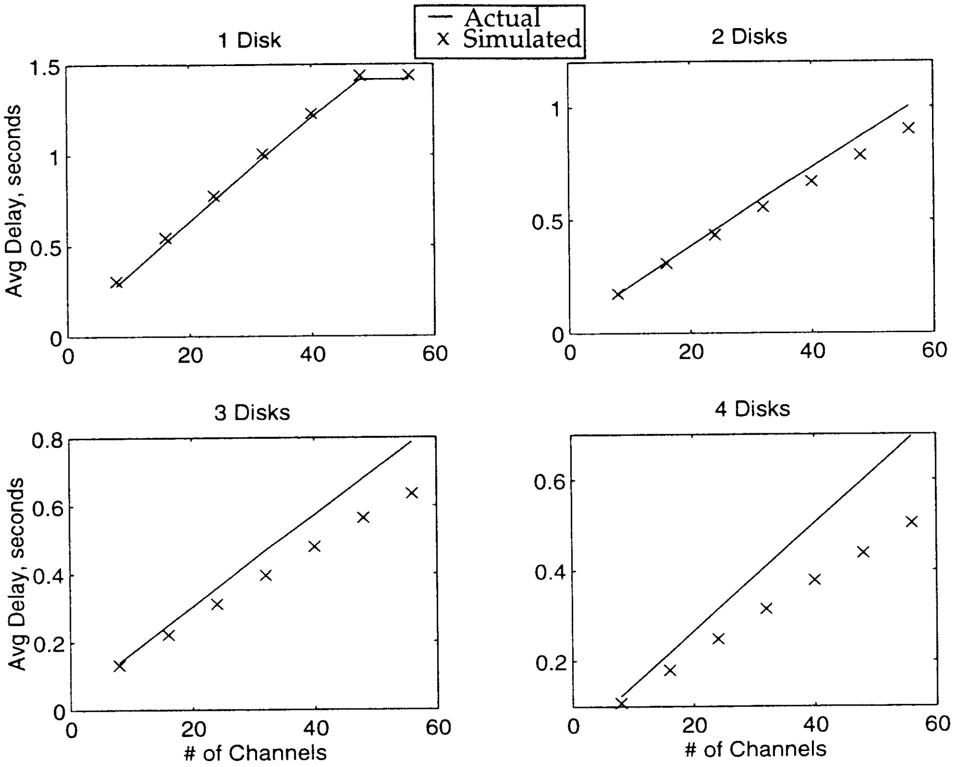

Figure 17. First Test: Lee Drive

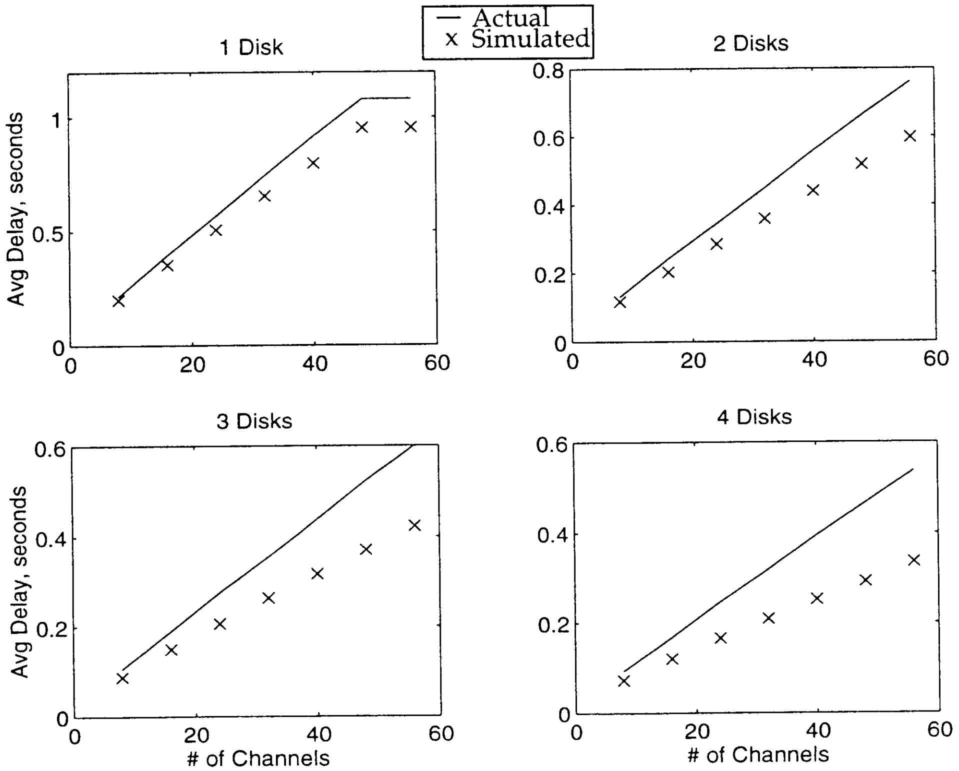

Figure 18. First Test: Lightning Drive

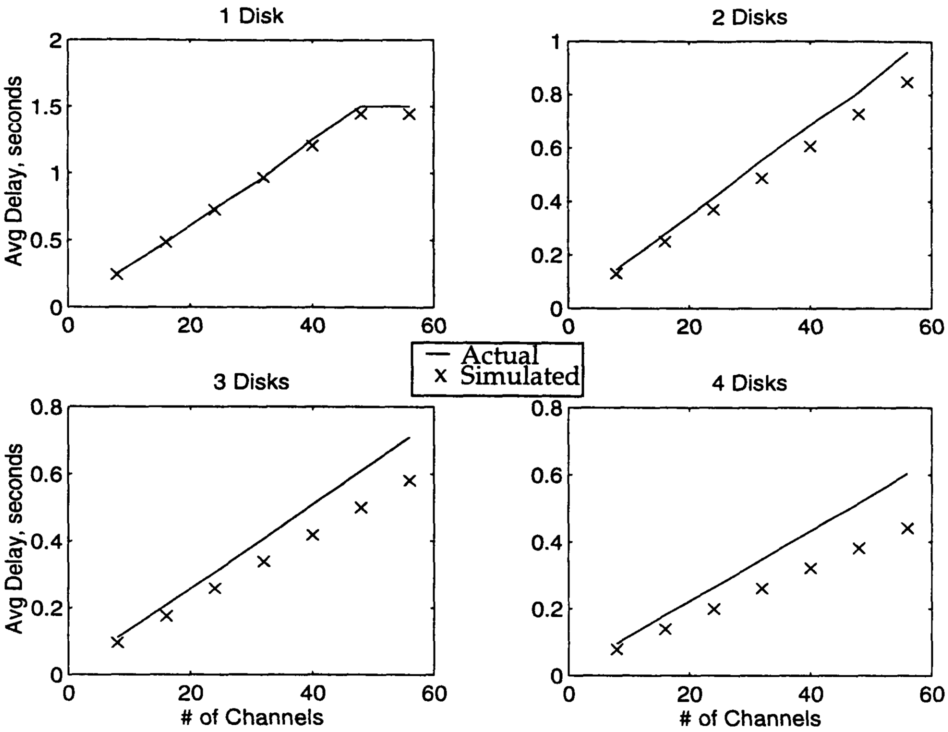

Figure 19. First Test: Lightning, no Elevator

The results show that the simulated results diverge from the actual at high numbers of channels. If the results are examined more closely, it can be seen that this divergence is virtually non-existent in single disk systems, and manifests itself increasingly as the number of disks increase. This indicates some problem with the manner in which the simulator modeled the interaction of multiple disks.

After investigating the various possibilities, it was concluded that the simulator was missing some small overhead that is present in multiple disk systems, and furthermore that this overhead is proportional to the number of disks on the system. The simulator was modified to add a small delay proportional to the number of disks on the system to each request.

4.3 Modified Run

The results of the modified test run are shown in Figures 20 - 22, in the same format as the originals:

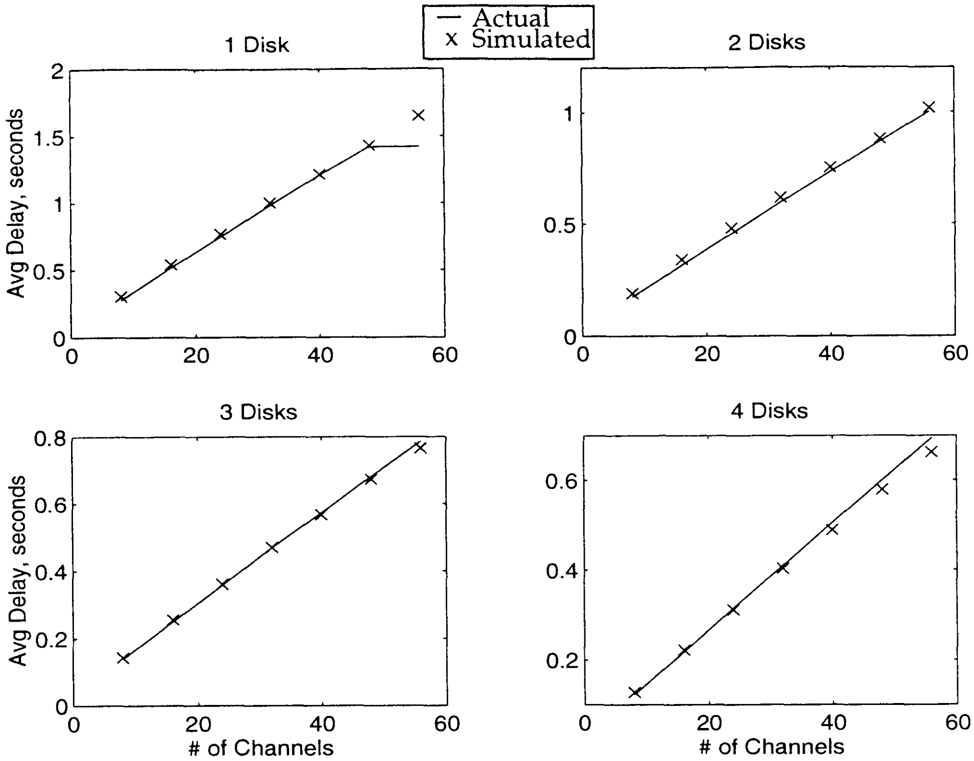

Figure 20. Modified Test: Lee Drive

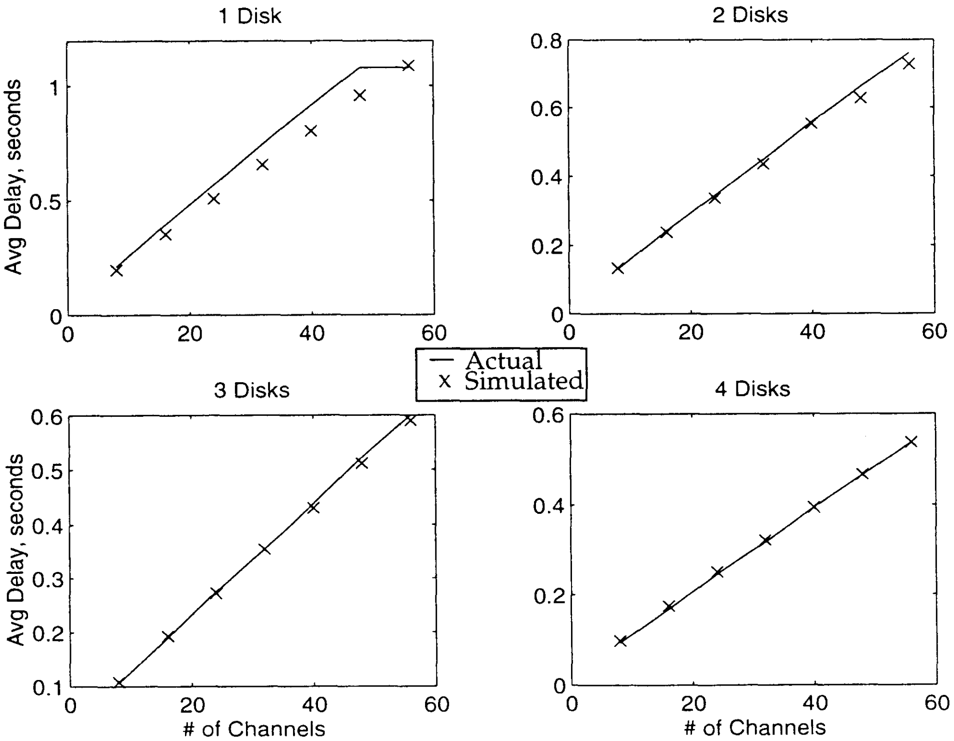

Figure 21. Modified Test: Light Drive

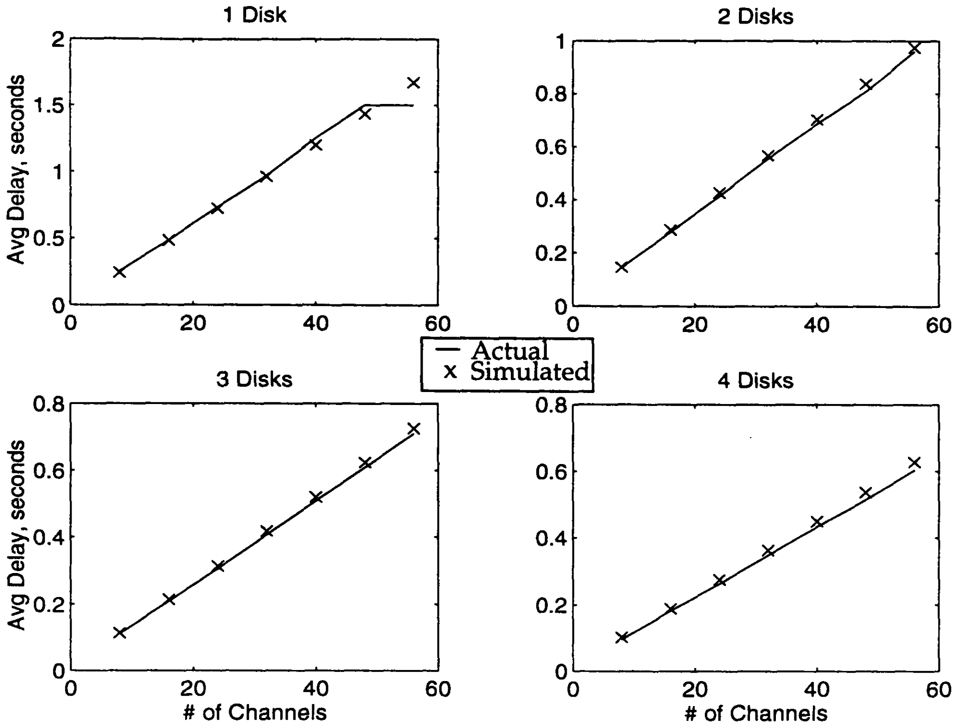

Figure 22. Modified Test: Lightning, no Elevator

The new results show that the modification to the program was effective. In view of the fact that the test checked three different kinds of drive, with from 1 to 4 disks, across a large range of "channels", we can conclude that the simulator models the disk activity of the voice mail system adequately.

5.0 Results

5.1 Introduction

With the model complete, the system was investigated with the goal of determining what factors cause the system to perform poorly. In the case of a voice mail disk system, poor performance is defined as disk service slow enough that users notice significant delays while conducting a voice mail session.

To ascertain under what conditions this occurs, it was necessary to begin with the simulation model. After a suitable simulation strategy was designed and executed, the mathematical models were used to provide a "second opinion". Comparing the results of the two models lent further insight.

It was assumed that 3 variables dominate the average response time of disk requests: speed of disks, the rate at which disk requests are made, and the number of disks on the bus. Simple calculations indicate that the bus is not a bottleneck.

- Speed of Disks: At the current time, disks do not operate much faster than the Lightning drive that was used to verify the simulator. It can also be assumed that a disk slower than the Lee drive would not be used in a functioning system. Thus, the simulations examined a slow drive (Lee) and a fast drive(Lightning).

- Disk Request Rate: The rate at which disk requests are made is not a directly controlled variable. It is a function of four things: number of channels on the system, average rate of calls into the system, average length of calls, and the average rate at which an active call issues disk requests. The final two factors are externally defined parameters that are held constant throughout the simulation set. The first factor, number of channels, is an important system parameter and is highly variable over different systems and over time. Thus, it is used as a simulation variable. Finally, the average rate of calls into the system is also very important because it determines the utilization of the available channels. Real voice mail systems are fitted with a certain number of channels based on expected usage patterns. It is therefore necessary to determine what call rates should correspond to certain numbers of channels. To quantify this, the mathematical model is used.

- Number of Disks: Since the SCSI bus allows a maximum of 6 drives, the systems are simulated with from 1 to 6 disks in each case.

Thus, the simulations are varied across two main variables: number of channels and number of disks. In every case, simulations are performed for both the Lee and Lightning drives. For each system configuration, the simulator determined a time value for which 98% of the response times (measured from user command to system response) fall below. For this thesis, a particular system behaves "acceptably" if 98% of all the delays measured are below one second. This somewhat arbitrary determination is based on actual operating conditions and serves the purpose of demonstrating the utility of the simulations.

5.2 Call Rate Determination

As mentioned in the preceding section, it is necessary to determine a relationship between the number of channels on the system and the call arrival rate to that system.

On an actual system, the number of channels is chosen according to the expected usage of the system, that is, the expected average call rate. This determination is made to ensure that the system has enough channels to handle the traffic, rarely rejecting users for want of an open channel. Thus, the call rate is tied to the number of channels during practical use. The simulation must have a similar relationship. To establish this relationship, a return to the mathematical model is needed.

In a system with a finite number of channels, the probability of the all the channels being busy at any given time can be calculated. The formula is taken from Section 2.2, repeated here for convenience (k is defined as the number of channels in the system):

$$\pi_k = \frac{\frac{(\lambda/\mu)^k}{k!}}{\sum_{j=0}^{k} \frac{(\lambda/\mu)^j}{j!}}$$

In this case. λ is the average call arrival rate, μ is the inverse of the average call length. Whenever the system is full, there is a danger that another call will arrive before one of the other k calls is finished. The probability of this happening is a relationship between the two rates, λ and μ, and is given by:

$$\frac{\lambda/\mu}{(\lambda/\mu) + k}$$

If we multiply this two equations together, we find the probability that a call will arrive when all the channels are occupied. If that probability is fixed, the equation can be set equal to a constant, and numerically solved for λ/μ as k is varied. Since we know how long the average call length is, and hence what μ is, we can find λ for any given number of channels.

For the purposes of this thesis, it was assumed that system usage would be fixed so that a call would be blocked approximately 1% of the time:

$$.01 = \frac{\frac{(\lambda/\mu)^k}{k!}}{\sum_{j=0}^{k} \frac{(\lambda/\mu)^j}{j!}} \frac{\lambda/\mu}{(\lambda/\mu)+k}$$

Using a math program, the appropriate λ (call arrival rate) was determined for a range of k(# of channels). Table 1 shows the resulting arrival rates and expected number of active channels for particular values of k.

k	λ	Active Channels
16	18	11
32	48	22
48	78	33
64	108	44

Table 1.

5.3 Main Observation

The simulation was run with values of 16, 32, 48, and 64 channels. The system performance was measured by compiling data on certain delays that occurred during the execution of each call. The delays measured represent the response times a user would experience after giving the voice mail a command. Unlike values dealt with previously, it is not the mean value of these delays that is of concern. Instead, it is the tail of the distribution of these delays. A certain percentage of the delays must fall below a particular value for acceptable performance.

For the purposes of this thesis, acceptable performance was defined as being 98% of all delays below 1 second. The results of the original simulation runs are shown in Figures 23-26. Figure 23 shows two graphs, one for each of the two disks. On each graph, a different plot line represents different number of channels (k) in the systems. Each line plots the 98% delay points versus the number of disks on the system. That is, a point on the graph, for example (1.6 seconds, 3 disks), signifies that the simulator found 98% of all delays to be below 1.6 seconds for the 3 disk system.

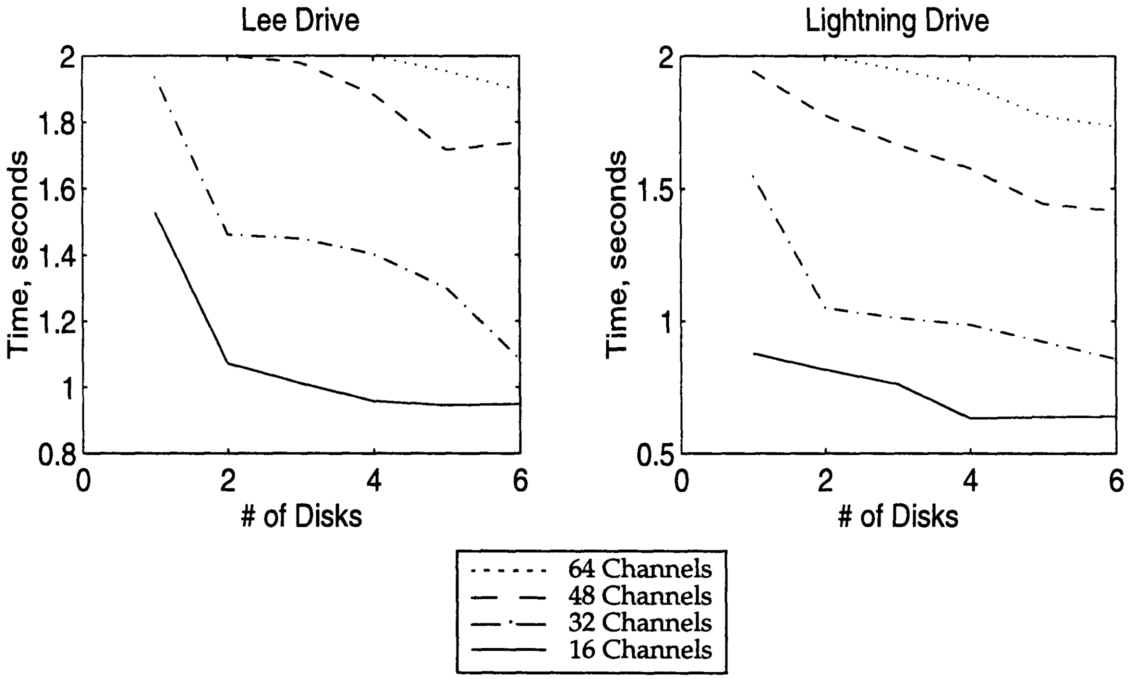

Figure 23. 98% Delay Points

At 48 and 64 channels, none of the systems meet the previously defined requirement of 98% of delays below one second. In the case of the Lee drive, at 64 channels only the 5 and 6 disk systems come in below the 2 second mark (the simulator does not report values higher than 2 seconds). With 6 disks, the maximum allowable on the SCSI bus, 98% of the delays were just below 1.9 seconds. The 32 channel systems do not perform much better, with the 32/6 (32 channels/ 6 disks) system measured at 1.1 seconds. The Lee drive only performs acceptably in 16 channel systems, with more than 2 drives.

In the case of the Lightning drive, which was until recently one of the fastest drives available, the 64/6 (64 channels/6 disks) system comes in at 1.74 seconds. The 48/6 measures 98% percent of delays below 1.44 seconds, and even the 32/5 is barely below the one second mark. Looking at 16 channel systems, we see that a one disk Lighting system should suffice,

The Lightning drive outperforms the Lee drive substantially in the 1,2, and 3 disk systems, averaging about 27% lower response times. In the six disk systems, this performance advantage decreases to about 17%.

The data are somewhat unexpected, in that they indicate that no matter how many Lightning drives are on the system, it cannot handle 48 or 64 channels. The first conclusion that might be made is that perhaps a faster drive is needed, so one was simulated. The performance characteristics for a drive approximately 25% faster than the Lightning (making it about as fast as fastest drive available today) were created.

The simulated results of the hypothetical drive for 48 and 64 channels are shown in figure 24. There are two graphs for each number of channels. The lower displays the 98% delay points versus number of disks, while the upper graph plots two variables against the number of disks: average delay across all disks in the system, and average delay for disk 0 alone.

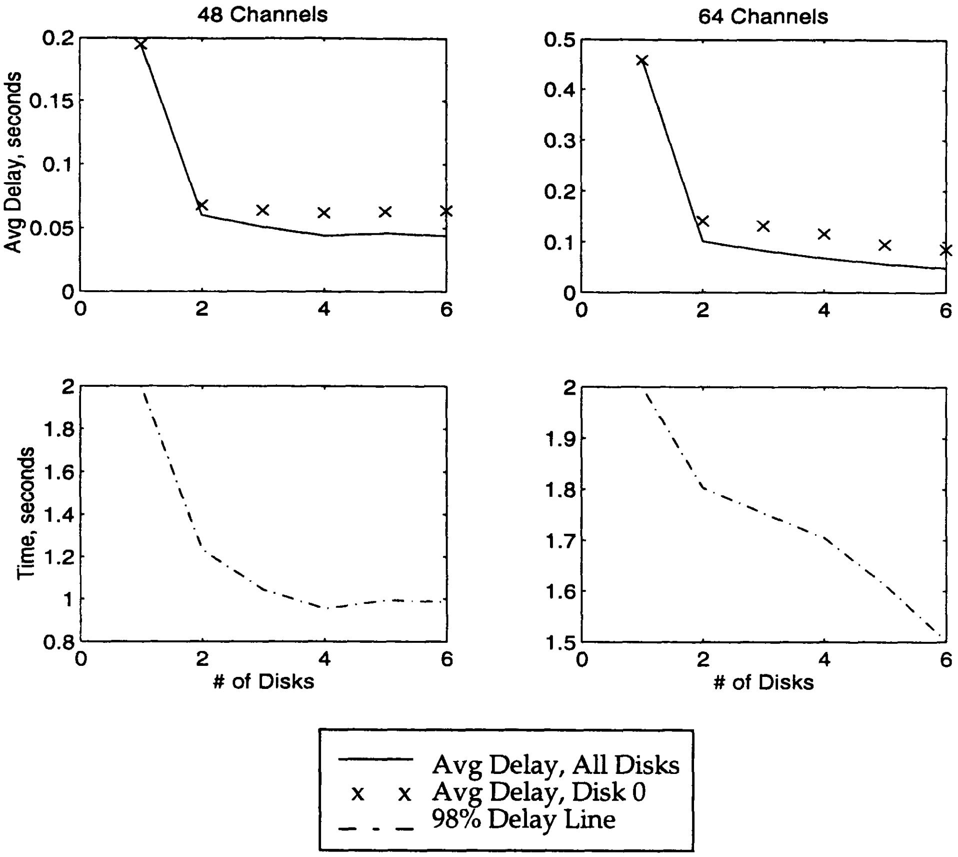

Figure 24. Fast Disk

The fast disk drive fares better, but not much. The simulator measured the 64/6 system with 98% of delays below 1.5 seconds. The 48/4, 48/5, and 48/6 systems all hover around the required 1 second point, but none are significantly below. Thus, it is likely that increasing disk service speed is not a viable solution.

Examining the top two graphs in Figure 24, we discover something important. In the 64/6 system, the average delay across all disks is 48 milliseconds. However, the average delay for disk zero is 77% larger, at 85 milliseconds. If we look at the other fast disk systems, this same phenomenon occurs; disk zero delays are substantially higher than the average.

The reason for this is clear, if one recalls that all the database requests are sent to disk zero. Since these requests make up a sizable percentage of the total request volume (about 35%), disk zero is much more heavily loaded than the other disks. In addition, all of the delay periods (the delays that make up the 98% delay line) contain several database requests. This would seem to explain the poor performance exhibited previously.

If we look at the average response times for the Lee and Lightning, we see that they have similar patterns. The results are displayed in Figures 25 and 26. Each figure shows four graphs, which vary the number of channels simulated. Each graph plots two variables against the number of disks: average delay across all disks in the system, and average delay for disk 0 alone.

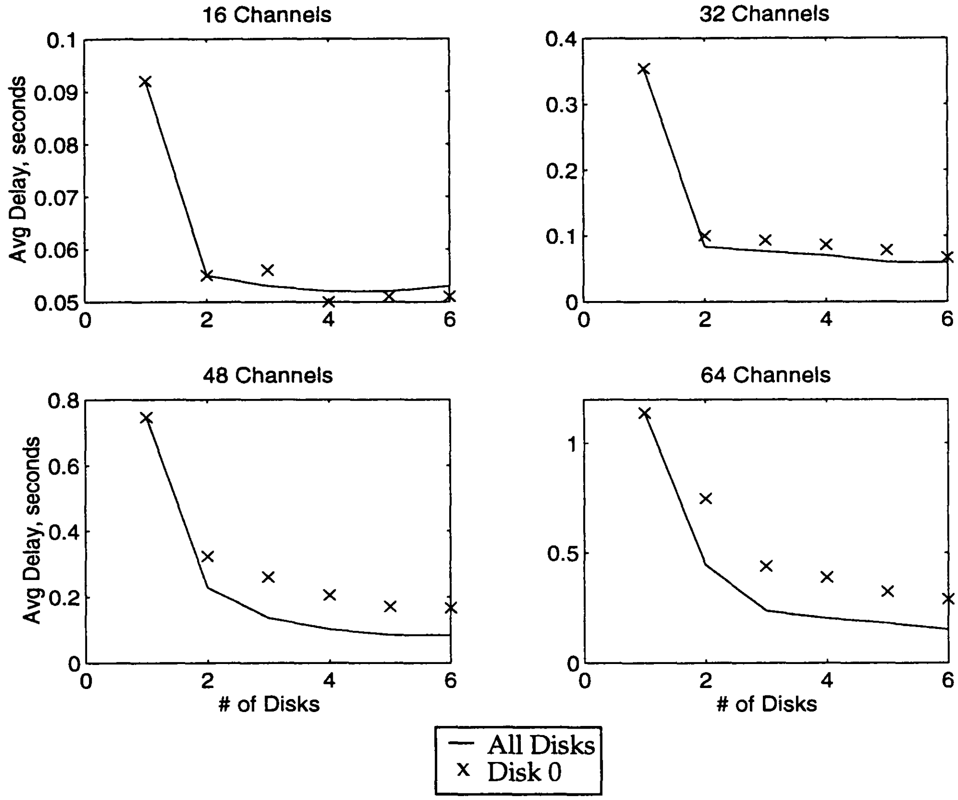

Figure 25. Lee Average Response Times

The Lee results show that in both the 48 and 64 channel systems, disk zero response times are substantially higher than those of the average disk.

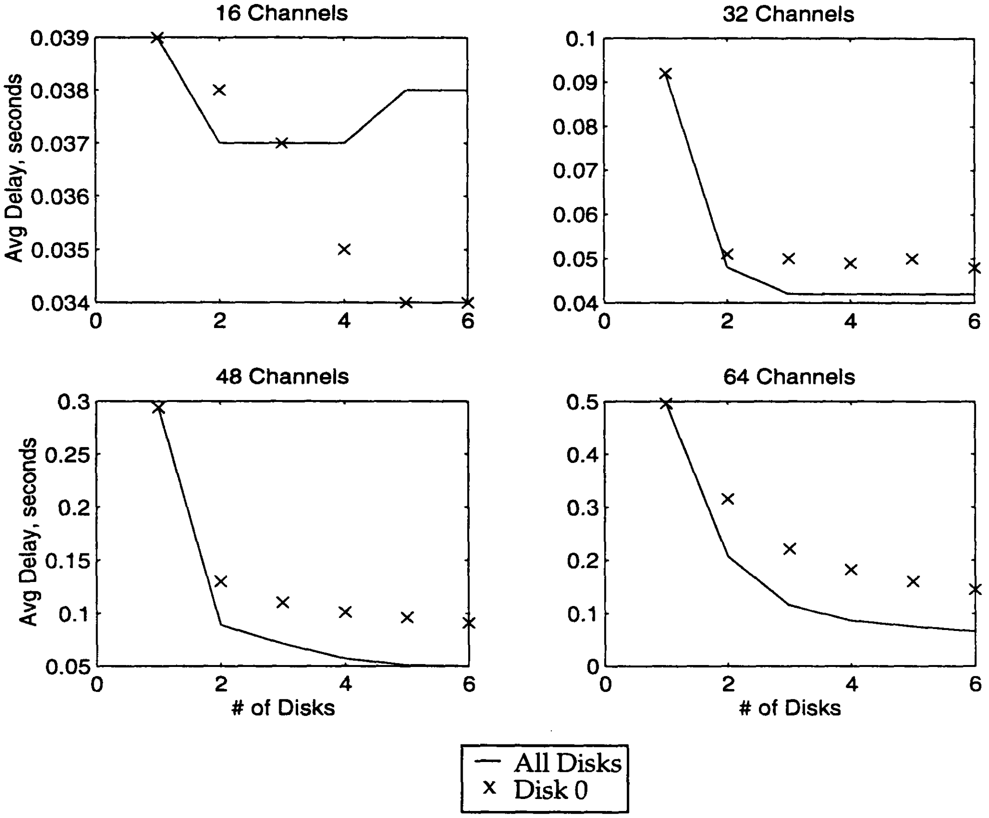

Figure 26. Lightning Response Times

The Lightning results also show that all three of the higher capacity systems have disk 0 response times well above the average. These data combined with previous data produce an important conclusion:

In higher capacity systems, disk 0 becomes a bottleneck and degrades performance sufficiently to cause unacceptable performance, no matter how many other drives are added to the system.

5.4 Improved Design

Since the bottleneck to disk 0 is caused by large volume of database requests, it is likely that alleviating this problem should allow higher capacity systems to perform acceptably. A solution to such a problem is to extend the database to include disk 1, as well as disk 0. As demonstrated by the preceding figures, the marginal increase in performance of going from one disk to two is very large. Thus, going from one database disk to two should be sufficient to alleviate the bottleneck.

The simulator was modified to send database requests to disks one and two with equal probability, and all other requests were weighted away from those disks. The simulation results for the modified simulator are shown in Figures 27 - 30. Figures 27 (Lee) and 28 (Lightning) show 4 graphs each, varied across the number of channels used.Each graph plots two variables against the number of disks: average delay across all disks in the system, and average delay for disk 0 alone.

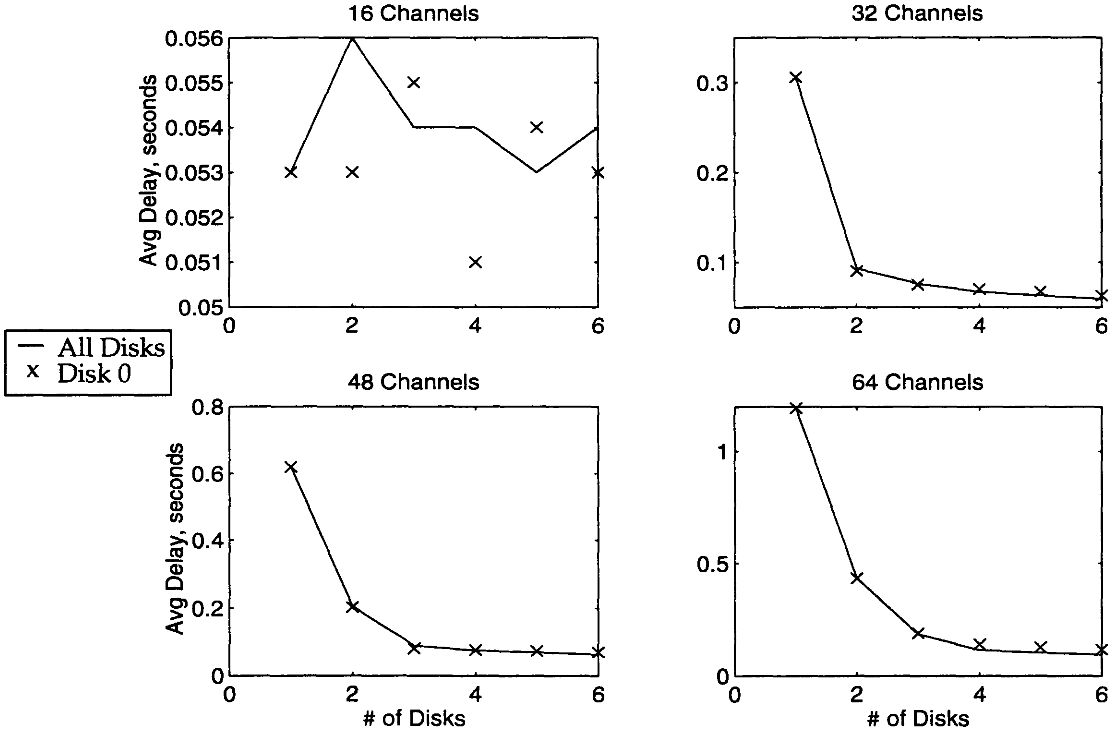

Figure 27. Lee Response Times

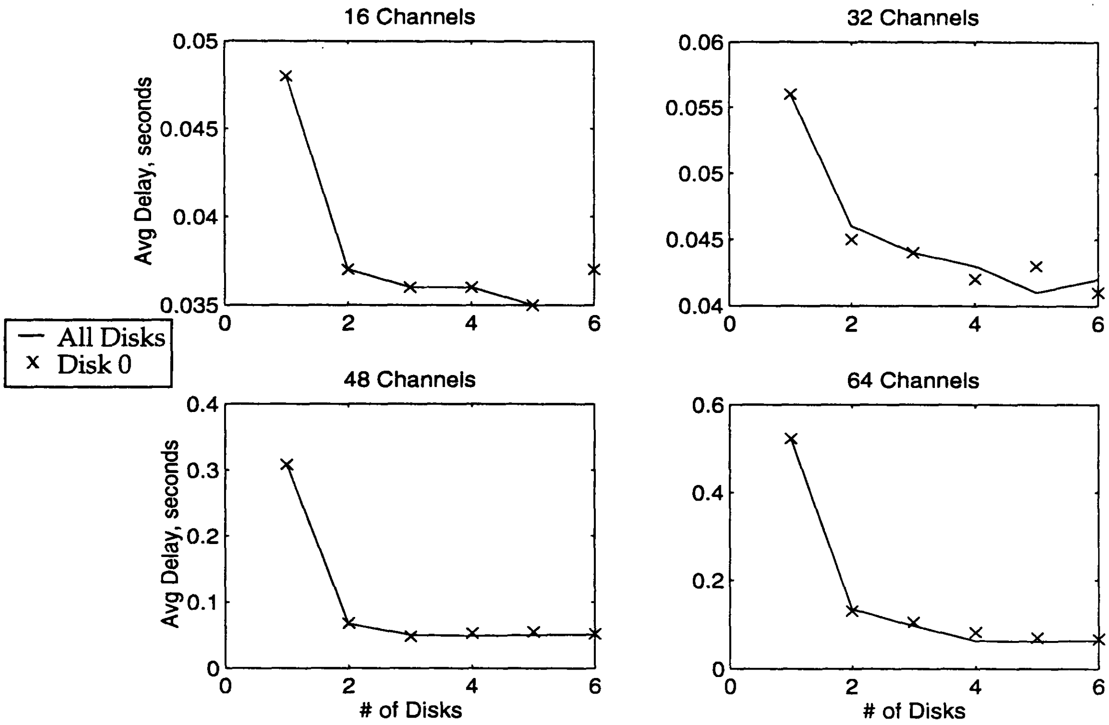

Figure 28. Lightning Response Times

It can be seen from these figures that disk 0 no longer has delays substantially longer than the average. Instead, the delay for disk 0 is close to the average delay across all disks. These averages are slightly lower than in the original simulation for all configurations. As can be seen in the Figure 29, overall performance improves quite dramatically. The figure shows two graphs, one for each of the two disks. In each graph, a different plot line represents different number of channels in the systems. Each line plots 98% delay points versus the number of disks on the system.

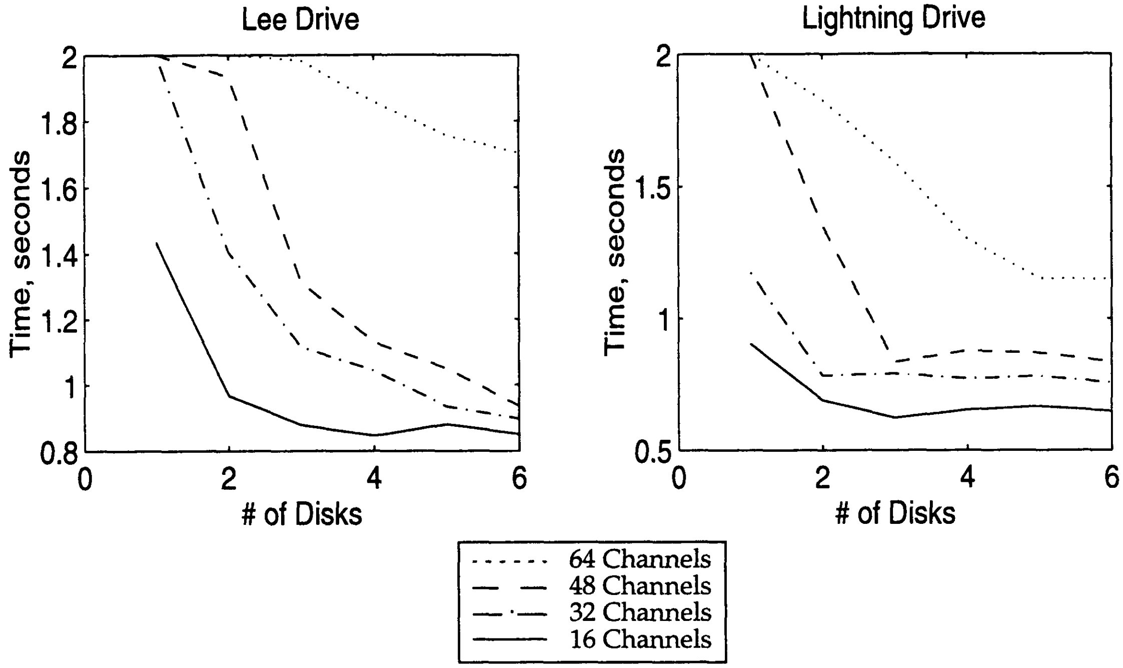

Figure 29. 98% Delay Lines

In the modified simulation, a Lightning drive is able to handle 48 channels with ease; even the 48/3 system comes in substantially below the one second mark The 32 channel systems also perform much better, needing only two disks to handle the load. As in the original simulation, only one Lightning drive is required to handle a 16 channel system.

The Lee drive can also handle 48 channels, but the system needs six disks. The Lee can handle 32 channels with five disks, and 16 channels with two disks.

For both systems, multi-disk performance has improved in all system configurations. However, even with the performance increase, none of the systems are able to handle the load that a 64 channel system creates. The fastest, the Lightning 64/6, comes in at 1.15 seconds. The highest capacity system needs faster disks, and so the hypothetical "fast disk" was simulated in the modified environment, at 64 channels. Figure 30 shows the results of this simulation. It plots the 98% delay points against the number of disks.

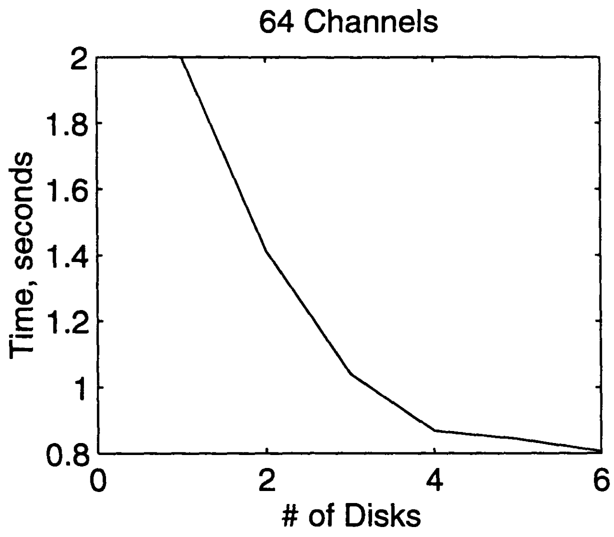

Figure 30. Fast Disk, 98% Delay Line

As Figure 30 shows, the 64/4, 64/5, and 64/6 systems are all measured below one second, with the 64/6 coming in at.8 seconds.

5.5 Comparison: Simulation vs. Mathematical

Once results from the simulator are obtained, it is possible to compare them with results from the mathematical model. They are not expected to match exactly; the simulator was built because the math model contained assumptions, such as exponential distributions, that are not entirely accurate.

To obtain results with the mathematical model, the key parameter λ, or the average request arrival rate per channel, must be estimated. This number signifies the amount of disk traffic each active channel generates, on average. Such a value is very difficult to obtain from an actual system, but it is possible to run the simulator and obtain a reasonably accurate value. The simulator keeps track of the average total request rate and the number of channels active. By simulating a few carefully selected system configurations, it is possible to estimate the value of λ by dividing the second into the first.

Using this value, results were obtained for the same set of data points (system configurations) as in the simulation. The same call rate values were used.

Figures 31 and 32 show the math model results for the Lee and the Lightning, respectively. They show 4 graphs each, varied across the number of channels used. Each graph plots two variables against the number of disks: average delay across all disks in the system, and average delay for disk 0 alone.

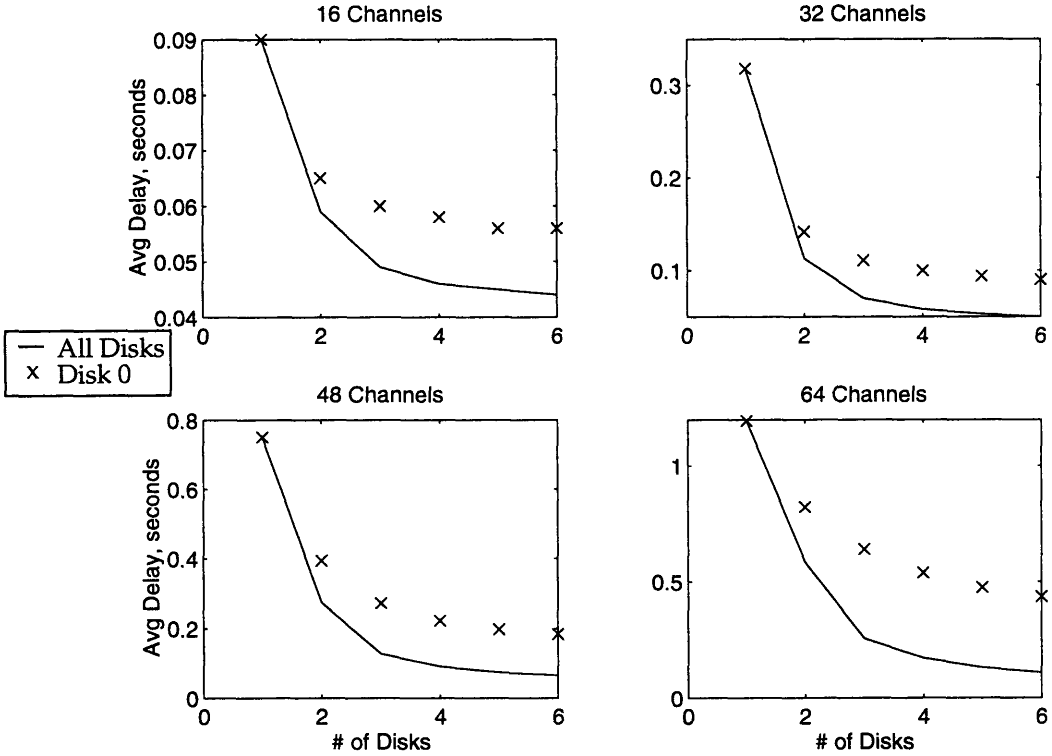

Figure 31. Lee Response Times, Math

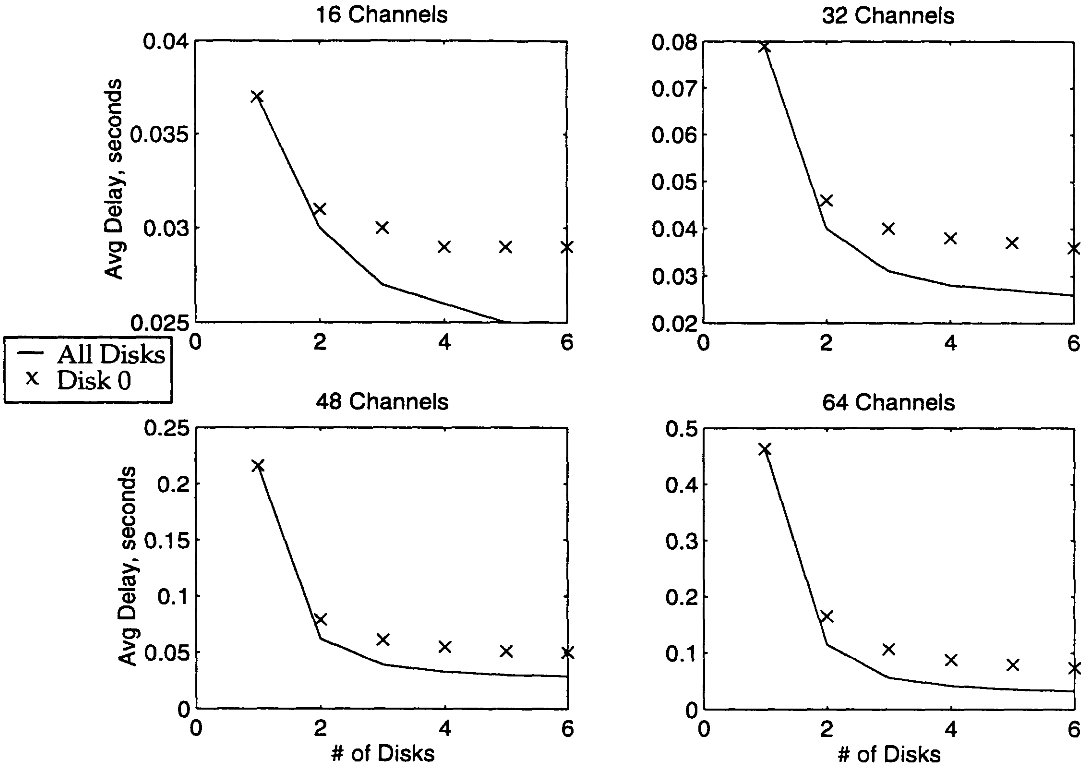

Figure 32. Lightning Response Times, Math

By examining these graphs, it can be seen that the math results have substantially the same pattern as the simulation results, including the above average response times for disk 0. A point worthy of notice is that the math model, even with its inaccurate assumptions, captures much of the important information. It shows a diminishing return with each extra disk, and tells us that disk 0 may cause a problem. Without the simulation, however, it would be difficult to estimate the magnitude of the problem.

Figures 33 and 34 allow quantitative comparisons to be made between the math and simulation results for the Lee and Lightning, respectively. Each figure has four graphs, varied by number of channels in the system. The graphs plot the average response times of both the math and the simulation models against number of disks.

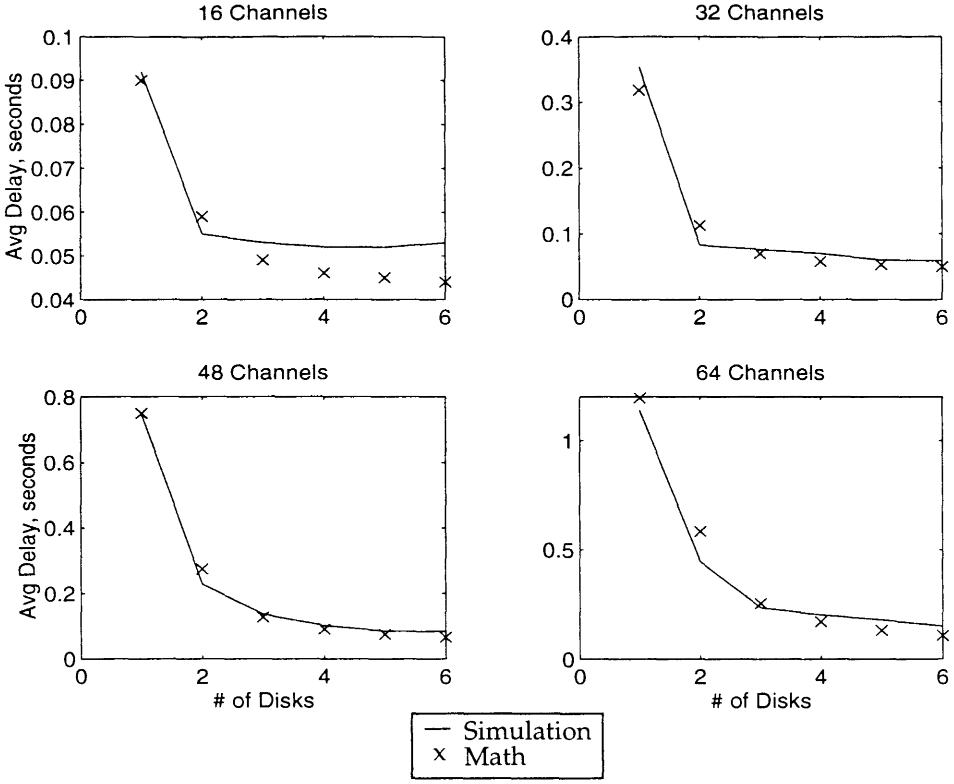

Figure 33. Math Vs. Simulation, Lee

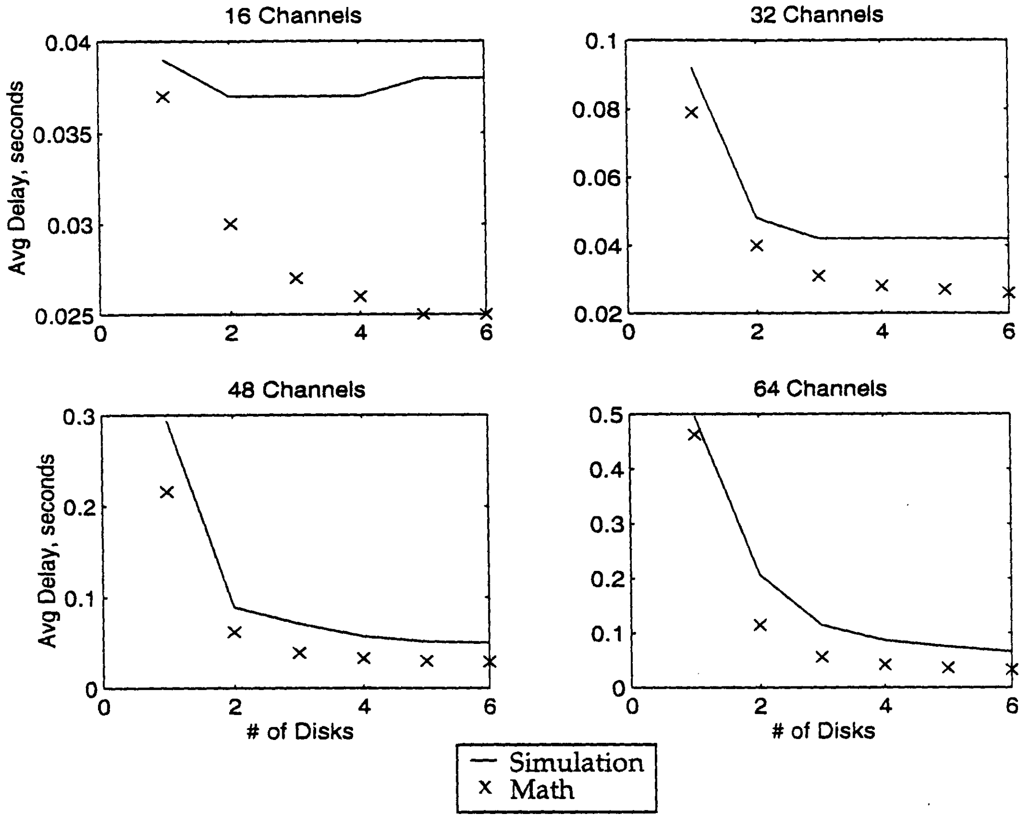

Figure 34. Math Vs. Simulation, Lightning

The plots in Figures 33 and 34 indicate that the two models exhibit all the same trends in approximately the same proportions. However, the mathematical model gives response time significantly lower than the simulation. It is difficult to see in Figure 33, but the Lee 64/6 simulated response time is 154 milliseconds, approximately 50% slower than the 104 millisecond response time from the mathematical model.

This result was to be expected; the simulation takes into account certain overhead that the mathematical model does not. The fact that the shape of the curves are nearly identical is evidence that the simulator results are not out of line.

6.0 Conclusion

This thesis accomplishes two things. One, it creates a performance model that relates the performance of meaningful system components to some useful measure of overall system performance. Two, it uses this performance model to draw meaningful conclusions about the system and feasibility of future systems.

Two types of models are presented. The first is a mathematical model based on queuing theory and Markovian systems, and the second is a software simulator written in the C language.

The mathematical models consists of 3 key parts. First, the "call queue" calculates, for given call arrival rates and average call lengths, the average number of channels active. Second, the "bus queue" calculates how long, on average, requests will have to wait at the SCSI bus. Third, the disk queue, using values generated from both the call and bus queues, calculates the average response time for disk requests. Three types of disk queue were presented: Single disk, multiple disk with symmetric access, and multiple disk with asymmetric access.

The simulation model has two main parts. First, the Request Processor, which simulates the behavior of the disk subsystem in response to any given set of disk requests. Second, the Request Generator, which creates the particular set of disk requests generated by the real voice mail system.

Using the simulator in conjunction with analytical queuing models, this thesis made conclusions about performance under various conditions and system configurations.The main observation of the thesis was the following: *In higher capacity systems, disk 0 becomes a bottleneck and degrades performance sufficiently to cause unacceptable performance, no matter how many other drives are added to the system.*Asolution to the bottleneck was simulated, verifying the main observation while at the same time presenting an option for future systems. It was also shown that the mathematical model captured much of the important information of the simulation, even though some of its underlying assumptions were not strictly correct.