A MODEL OF THE UNIX TIME-SHARING SYSTEM UNDER DISK SATURATION

By

BARRY JEFFREY BRACHMAN

B.Sc., The University of Regina, 1981

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1983

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date Sept. 10, 1983

Abstract

A deterministic model of the UNIX time-sharing system under disk saturation is presented and a performance experiment on a PDP-11/34 is conducted to establish the validity and accuracy of the model. The basis of the model is that the ratio of mean file system access rates between two different systems can provide, under certain circumstances, a useful performance comparison between the two systems. Given several workload and system parameters as well as the elapsed time to perform the workload, the model predicts from a second set of parameters the elapsed time to perform the second workload.

Workload, hardware, and internal system parameters are identified and tools are constructed to record these parameters. A controlled experiment, using a synthetic workload, is then conducted. The results are analyzed and the model is evaluated. The model is extended in response to regularities discovered in the measurement data. Sample applications of the model are given. The suitability of the tools developed and the methods used are discussed.

Table Of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank Dr. Sam Chanson for his ideas and guidance, careful readings of the thesis, and financial support through research assistantships.

I also thank Dr. Son Vuong for both his suggestions and his readings of the drafts.

I am grateful to the Department of Metallurgical Engineering for giving me unlimited access to their computer facility and its resources.

Finally, I wish to thank my family and friends for their support and encouragement during my research.

1. Introduction

1.1 Overview

Time-sharing systems have long been implemented using the memory management technique of process swapping. Swapping involves exchanging one or more processes loaded in main memory with one previously written to secondary storage. Normally an entire process is swapped out. The technique is commonly used on systems without virtual memory capabilities; the entire process often must be resident to be executed. Although the use of virtual storage has become widespread, many small and medium size time-sharing systems still use swapping.

A primary task of computer system performance evaluation is to predict the performance of a computer system under a given workload. The improvement study is concerned with modifying an existing system to increase its performance or decrease its cost, or both. Upgrading and tuning are two types of improvement study. Upgrading is replacing or adding one or more hardware components, while tuning is adjusting the system parameters to adapt the system to its workload. This thesis concentrates on the predictive aspect of performance evaluation (See Dowdey et al. [12] for an example of such a study).

Because performance is a subjective concept, a precisely defined descriptor called a performance index is used to represent a system's performance or some of its aspects. Objective measures of performance include throughput rate, response time, and turnaround time. Performance analysis involves determining the values of performance indices for given values of the installation's parameters. The performance information required by a study may be obtained from the system itself or from a model of it.

A number of major obstacles stand between the analyst and both the development and evaluation of a useful model. First, operating systems are large and complicated programs, incorporating various strategies to control resource sharing. They are generally not designed with future performance studies in mind. Events within the system occur asynchronously. Many complicated, *ad hoc* methods are often used and the performance of these methods in combination and under diverse conditions is often unclear [21].

A second major difficulty in the evaluation of computer systems is the lack of a quantitative understanding of the relationship between workload and performance. In performance evaluation, the real workload is characterized by a workload model which is used to drive a model of the real system. The workload model specifies the characteristics of the resource demands placed on the devices by programs. It is not always clear what a system's "typical" workload is because the workload often changes from minute to minute and from day to day. The analyst is sometimes interested in evaluating a system's performance under workloads differing from the "normal" workload; e.g., the performance of the system under heavier than usual load. Characterizing a workload requires determining with enough detail which of its many aspects influence the system's performance and then creating a workload model composed of a set of quantifiable workload parameters. When the production workload cannot be used or does not meet the study's needs, a synthetic workload must be constructed and validated. Synthetic workloads are compact and reproducible while giving the analyst more control over the experiment. These advantages are often achieved at the cost of representativeness.

1.2 Basic Goals, Approach, and Motivation

The basic goal of this research is to discover the major relationships between swap-

ping activity, hardware characteristics, workload characteristics, and system performance under the Version 6 UNIX[1] time-sharing system. A simple model of the computer system is hypothesized, then a controlled experiment is conducted to test the validity, accuracy, and range of the model.

First, a model of the system is postulated based on major workload and performance parameters obtained from earlier work [13,38] as well as from observations of the system under load. Tools to measure and extract pertinent workload, hardware, and system information are developed. Several synthetic workloads are constructed and a controlled experiment is conducted, varying several workload parameters. The results are then analyzed and compared to the performance predictions obtained from the model. Once the accuracy and domain of validity of the model are determined, the effect a different number of processes, a different main memory size, or different disk characteristics would have on system performance can be evaluated. Such information could be useful in deciding whether the addition of more main memory or a disk system upgrade, for example, would be a more cost-effective solution to a performance problem.

A primary motivation for this project is to arrive at a better understanding of how swapping influences performance. Some work has been done in this area (see Section 3.1), but most research on memory management techniques in a time-sharing environment has been concerned with paging systems [9]. It is hoped that better insights into swapping may be useful in comparing swapping to paging and perhaps any similarities may allow some of the results obtained from one scheme to be applied to the other.

All experimental work was performed on a UNIX system. The UNIX system is a popular swapping system[2] and is becoming the *de facto* standard for 16-bit computers.

---

[1] UNIX is a trademark of Bell Laboratories.

[2] The Berkeley UNIX implementation uses paging.

It is hoped that the results of this work will be useful to UNIX installations in particular and perhaps to other swapping systems as well.

It is obvious that swapping in itself is not a "bad" memory management technique. If swapping time and overhead are made negligible (say by the introduction of a solid-state, DMA-based swapping device or by a bank switching mechanism), then it effectively becomes a physical address extension mechanism while not relieving the virtual address limitations.[3] Note that if swapping could be performed "instantly", the system would do so after each quantum expiration or whenever a process blocked, whichever came first. As will be seen in Section 3.2.2, if reasonable scheduling assumptions are made, CPU bound workloads can execute without swapping significantly impacting performance. It is less apparent how swapping affects the performance of the small-to-medium-sized machines where disk and memory resources are often at a premium and where performance is often sensitive to small changes in the workload characteristics. This thesis is concerned with the analysis of swapping on these small systems and the development of a conceptual model which may be used to estimate performance changes due to various hardware or workload changes.

Chapter 2 contains a description of the hardware and software used in this study. Chapter 3 describes a model of the system which will be evaluated in Chapter 5. Chapter 4 deals with the description of the workload and the experimental method. The results of the experiment are detailed in Chapter 5 and an analysis is conducted. Conclusions are presented in Chapter 6.

A basic understanding of operating systems is assumed throughout this thesis, although prior knowledge of UNIX is not necessary.

---

[3] The same can be said about a solid-state paging device.

2. System Description

2.1 The Hardware

The host machine for this project is a Digital Equipment Corporation PDP-11/34A [27]. It has 64K words (2 8-bit bytes per word) of main memory (MS11-LB). Memory may be expanded to the maximum of 248K bytes. Secondary storage consists of 2 RL01 cartridge disk drives which share a single controller [28]. The PDP-11 allows both the CPU and disk controllers to be simultaneously active; i.e., disk I/O can be overlapped with processing. Although the controller used permits a rudimentary overlapped seek capability, this was not used. The characteristics of the RL01 disk drive are given in Table I. The system has four interactive terminals, three of which are connected to a DZ11 serial communications interface. Other peripherals include a plotter, DUP11 serial controller, IEC11 (IEEE-488) bus controller, and a serial line to the campus network.

RL01 Characteristics

Surfaces:	2
Tracks/Surface:	256
512-byte Blocks/Track:	20
Track-To-Track Seek:	15 msec.
Ave. Latency Time:	12.5 msec.
Ave. Seek Time:	55 msec.
Ave. Access Time:	67.5 msec.
Transfer Rate:	512.5K bytes/sec. (peak) 2.35 μsec./byte (ave.)

Table I

2.2 The Operating System

The operating system in use is a locally modified version of the Sixth Edition UNIX, the earliest distributed system. Overviews of the system can be found in Ritchie and Thompson [31], while detailed descriptions of the internals can be found in [22] and [38]. Many of the local modifications to UNIX are not significant to this study; those that are will be presented shortly. Few UNIX installations run a copy of UNIX that has not been altered in some way and few have the same hardware configuration. Because UNIX systems run primarily on PDP-11 minicomputers, the maximum number of users is usually less than 30. The maximum process size is almost 64K bytes on the smaller machines and twice that on PDP-11s with separate instruction and data spaces. Because there are now several versions of UNIX distributed by Bell Labs, any nonqualified reference to UNIX in this paper will imply the Sixth Edition version. It is believed that the other swapping versions of UNIX are similar enough to the Version 6 system that many of the results of this project apply to them as well, although we do not have access to these systems to verify our belief.

The following sections provide a brief overview of those areas of UNIX applicable to this thesis.

2.2.1 Processes

Ritchie and Thompson [31] define an image to be "a computer execution environment". The environment includes a memory image, general register values, status of open files, etc. A process is defined to be the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory. During the execution of other processes it may be swapped-out to secondary storage. The existence of a process implies appropriate entries in the system data structures [22]. UNIX permits the user to initiate a number of concurrent processes. This particular feature was

utilized by the synthetic workload, which is described in Section 4.2.

The user-memory part of an image is divided into three logical segments: text, data, and stack. Generally, all three segments are contiguous in main memory. A swapped-out image is stored contiguously (logically) on the swapping device. It is possible to separate the program code and to make it sharable among all users. In this case the read-only program text need never be swapped-out.

2.2.2 Processor Scheduling and Swapping

Since both scheduling and swapping are important to this work they will be discussed in detail. The information is a summary of Thompson [38].

Process synchronization is accomplished by having procedures wait for events. Events are, by convention, addresses within the kernel of tables associated with those events. Processes are suspended when, for example, they wait for the completion of an I/O request or when they require a resource which is (temporarily) unavailable. When an event occurs, any processes waiting for the event are awakened. Although this method is quite simple, it can lead to large inefficiencies. If, for example, a number of processes are waiting for the same resource to become available and all have been swapped-out, then when the resource becomes available all such processes will eventually be swapped-in. Only the first waiting process swapped-in will get the resource and the remainder will wait again, possibly being swapped-out.

The system call **sleep**, which suspends the caller for the number of seconds specified by the argument, can be especially inefficient. Suppose that one process on a busy system frequently **sleeps** for a short period and that a second process has issued a **sleep** call with a large argument. Each time the first process awakens, the second process will also awaken, be swapped-in, determine that its **sleep** has not yet expired, and

subsequently be swapped-out. **Sleep** has been replaced in newer versions of UNIX by a subroutine of the same name and two new system calls, **pause** and **alarm**, which are more efficient. The system keeps track of **alarm** requests for each process so that a process awakening from a sleep will not disturb other processes which have called the **sleep** subroutine.

An integer priority is associated with each process. The priority of a system process (i.e., the scheduler or a user-process executing within the kernel) is assigned by the code which issues the wait on an event. The priority is roughly inversely proportional to the reaction time that one would expect for such an event. Disk-related events have high priority, terminal input or output have low priority and time-of-day events have very low priority. Thompson [38] claims that the difference in system process priorities has little or no performance impact. All user-process priorities are lower than the lowest system priority. Since high-priority processes are chosen first by the scheduler, waking any system process will preempt any currently running user-process. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time in the last real time unit assigned a low user priority. The system-process reverting to a user-process will continue to enjoy the high priority until it makes a system call, initiates a trap, or upon the next one-second clock interrupt, whichever occurs first. There is no time quantum as such, but it will generally be at most one second and looping user processes will be scheduled round-robin.

To support more processes than main memory can simultaneously contain, UNIX utilizes the memory management technique known as swapping [8]. A process which cannot be entirely loaded into main memory resides on secondary storage until it can replace, or be exchanged with one or more loaded processes. The user data segment, the system data segment and the text segment are swapped-in and swapped-out as needed.

The user data segment and system data segment are kept in contiguous primary memory to reduce swapping latency. Allocation of both main memory and swap space is performed by the same first-fit algorithm. Memory compaction is not performed so that while there may be enough total free memory, there may not be enough contiguous main memory to load a process. This adds to the swapping load. When a process grows (UNIX permits a process to dynamically modify the size of its data area), a new piece of primary memory is allocated. If there is not enough primary memory, the swapping mechanism is used and the process is eventually swapped-in with its new size. The swapping mechanism may also be invoked when a process "forks" a copy of itself.

The scheduler is implemented as a kernel process. It examines the process table, looking for the oldest swapped-out process which is ready to run. If there is sufficient primary memory, the process is swapped-in and the scheduler continues to look for a process to swap-in. If there is insufficient free main memory, the scheduler searches for a loaded process which is waiting for a "slow event" (i.e., a low-priority system event which tends to take much longer than the execution of a swap, for example, terminal I/O). If such a process does not exist, the time that the swapped-out process has been on disk is compared to an integer constant, say *MINTIMEOUT*. If the process has not been out at least *MINTIMEOUT* seconds, the scheduler waits. If the swapped-out process satisfies this condition, a search is conducted for the oldest resident process. If this process has not been loaded for at least *MINTIMEIN* seconds, the scheduler waits.[4] Otherwise this oldest process is swapped-out, its memory is freed, and the scheduler goes back to look for a process to swap-in.

The scheduler waits for a swap-in or a swap-out to complete. Only one scheduling

---

[4] In the distribution system $MINTIMEOUT = 3$ seconds and $MINTIMEIN = 2$ seconds. In this work both were 3 seconds.

swap can be submitted at a time. If the swap device is connected to the same controller as a file system device, swaps are serviced in the same manner as any other disk request. Thompson [38] observes that if the swapping device is also used for file storage (as it is on most small systems), the swapping traffic severely impacts the file system traffic. If the capacity of the drive is small, the swap area is usually placed on the inner-most tracks of the disk, increasing the seek time involved in a swap.

2.2.3 The Block I/O System

The block I/O system considers a device to have a number of 512-byte memory blocks. These devices are accessed through a common buffering mechanism [29]. The system maintains a number of 512-byte buffers in the kernel address space,[5] each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the appropriate device driver is called to fill up the renamed buffer, and the data are made available. Write requests are handled similarly; however the write is performed by marking the buffer and physical I/O is deferred until the buffer is renamed. If less than a complete block is being written, the block must be read first. A user-process, **/etc/update**, executes a system call every 30 seconds which causes all delayed write operations to be flushed from the queues.

When physical I/O is required, the appropriate device driver is called to link the buffer into its queue and to execute the read or write operation. Normally the UNIX

---

[5] On this system, 20.

RL01 device driver performs the requests in a first-come first-served manner. The system under study, however, was modified to use the LOOK disk scheduling policy [36,37].

3. The Model

A model is a representation built to study a system and consists of a certain amount of organized information about it. Models representing the same system may differ in their purpose, standpoint, or amount of detail. An advantage of modeling over direct measurement is that it allows the analyst to obtain results when the target system is not available; i.e., in the design of new computer systems, in improvement studies where the effects of new hardware or a reconfigured system are of interest, or where it would be too costly or impractical to give the analyst direct control.

Verbal and somewhat superficial models of a real system are called conceptual models. Conceptual models are the basis of measurement techniques and of two major types of modeling techniques: simulation methods and analytic methods. Graham [15] gives a brief overview of modeling. An analytic model is composed of mathematical relationships which represent the system. Although many simplifying assumptions may often be made about the workload parameters and the operating system's characteristics, a model can still provide satisfactory results at a comparatively low cost. Queueing models [14,19] and models based on operational analysis [4,10] are two such analytic methods. Grenander and Tsao [16] assert that queuing models alone are insufficient to study and evaluate computing systems, but that they help in understanding the behaviour of the system.

A simulation model reproduces the behaviour in the time domain of a system according to some conventions which establish a correspondence between various aspects of the model and of the system [14]. Simulation is more flexible but is also more costly than analytic techniques. Simulation may be used when the analytic equations are insoluble or an analytic description is unclear. It too suffers as the model grows more complex. It is often necessary to validate the results of simulation with analytic methods or

direct measurement because it is difficult to know if the simulation program itself is correct. The results of simulation are not always easy to interpret with respect to cause and effect relationships between load and system parameter and performance measures, while closed-form analytic solutions often yield such insights [24]. Hybrid simulation uses an analytic model to replace part of the system being modeled, simplifying the overall simulation.

The development of a model often produces insights which have greater implications than the performance study since it requires the analyst to seek out a single, unified picture of what may often appear complicated and inconsistent. Models not only provide quantitative results from given values of parameters, but also may play a qualitative role by developing the analyst's intuition about actual systems.

A drawback to modeling is the possible lack of fidelity to the modeled system. Any model must be validated before it can be used to produce the information needed for evaluation. There is no better way of confirming confidence in a model than by experimentation. Of course measurement cannot be used when the modeled system does not exist or is not available.

3.1 Related Work

Most modeling and analysis of memory management techniques for time-sharing systems have been directed towards paging rather than swapping. Many studies are concerned with balancing on-line load with batch load; i.e., the tradeoff between turnaround time and system utilization. Work which is related to scheduling methods designed to reduce swapping will be mentioned here along with modeling work.

Scherr [33] developed one of the first swapping models, constructing a simple model of the Compatible Time-Sharing System (CTSS). This early interactive system used

swapping and supported only a single completely loaded process. An interaction was described by the amount of processor time required during the interaction, including system overhead and unoverlapped file system access times, as well as the program size. No overlapped processing and channel operation was allowed. Both simulation and continuous-time Markov models were described. In the Markov model, Scherr took the mean processor time per interaction to include the necessary mean swap time per interaction since the processor was idle while swapping. Distribution based calculations were used to determine both swapping and file access times in the simulation. The think time distribution and the CPU service time distribution were assumed to be exponentially distributed. Given the number of active users, the mean response time predicted by the analytic model was accurate when compared to the actual measurement data.

Coffman [7] analyzes time-sharing algorithms which are designed primarily to reduce swapping. The observation made is that to reduce overhead, the time quantum should decrease until a certain point with increasing load, then increase to reduce swapping. A multiple quantum algorithm is presented. The algorithm retains the general structure of the round-robin discipline and the multiple-level waiting time distributions except under heavy load when it avoids saturation conditions by increasing the time-slice. This increases the efficiency of system operation.

Bernstein and Sharp [2] propose a processor scheduling algorithm designed to reduce swapping. They claim it is desirable to incorporate into the scheduler a specification of a minimum level of acceptable service for each class of program. Scheduling and swapping decisions are made with respect to a family of policy functions which specify the level of service to be offered to various classes of users. They also suggest that under heavy load both the minimum memory residence time (*MINTIMEIN*) and the minimum time spent out on disk (*MINTIMEOUT*) be increased by a quantity they

term the hysteresis. Hysteresis could also be adjusted according to the size of a process.

In [20], Kleinrock solves a queueing model to calculate the expected swap time expended on all customers in a system of queues. Both time-shared systems and processor-shared systems are considered. Processor-shared systems are time-sharing systems in which the quantum size is allowed to approach zero. In order to apply the results, the expected wait in queues conditioned on a service requirement must be known for time-shared systems which account for swap time.

Nielsen [26] studies the effectiveness of program relocation, rotational delay minimization, and swap volume minimization techniques. Nielsen uses a set of prototypes in his simulations with each job type based on seven distributions reflecting a job's resource needs. The effectiveness of faster swapping devices is shown to depend upon the overall balance of the system.

A simple model of a multiprogramming system is developed by Tsujigado [39] and the CPU idle time due to swapping is analyzed. The relationship between the number of channels used for swapping and the rate of idle time generated in the processor is derived.

Shemer and Heying [34] analytically model a swapping system designed for batch and time-sharing users and compare the results with empirical measurements. The model parameters are obtained from performance statistics and mean values. The expected response to "typical" interactive user demands is predicted from the number of concurrent users. Empirical data is used to substantiate the validity of the assumptions employed in the model and to determine any correlation between measured performance and the results of the model.

A limiting resource model of the EXEC8 time-sharing system is developed by Strauss [35] under the assumption of swapping saturation. Swapping saturation is the

initiating of one swap operation as soon as the previous swap completes. This simple conceptual model predicts the steady-state performance for both interactive and batch users as a function of configuration and workload. Mean values obtained from direct measurement are the inputs to the model. The results from the model agreed well with measured performance values outside the model's range, although no measurements were available to verify the model under swapping saturation.

Chen [6] studied a closed queueing network with state dependent routing probabilities for swapping based systems. The model predicts mean system response time, and utilization and queue length of an individual resource. The probability that a program must be swapped-in is expressed as a factor of the system state and system parameters. An approximate solution is obtained from an iterative algorithm using Buzen's method [3]. The results of the model are compared with both the classical model and measurement data.

3.2 Initial Work

An initial attempt at understanding swapping behaviour was directed at investigating the efficacy of a performance parameter called the mean time between swaps (MTBS). We define this to be the mean amount of real time which elapses between complete swaps. This parameter is supposed to be analogous to the mean time between page faults on a virtual memory system [1]. A prediction of MTBS from hardware and workload parameters would allow swapping rates to be predicted for simulation purposes. The MTBS could also be used by a queuing network model to obtain estimates of mean throughput and response time. It could also perhaps lead to a load control policy based on the shape of the swap lifetime curve, just as the paging lifetime curve is used for such purposes on a virtual memory system [11]. Although swapping is a system-wide

activity while paging activity is a per process characteristic, it was speculated that there are enough similarities between the two techniques that such an investigation could prove worthwhile.

A major difference, however, is that while a high page fault rate is never desirable, it is not clear that a high swapping rate is undesirable. As long as swapping does not interfere with other disk activities and the CPU idle time due to swapping is negligible, then swapping might as well be performed as frequently as necessary. Ideally, a process should be swapped-out when it no longer needs the CPU. Another difficulty with this approach is that the MTBS seems to depend upon both the total number of processes and the process sizes in a complicated manner. Although some empirical results were studied, it did not appear that research along these lines should be continued for the reasons noted above.

A subsequent effort entailed a development of a model which could estimate the ratio of throughput with swapping overhead to the throughput without swapping overhead, given the ratio of "memory quantum" requirement to the mean one-way swap[6] time. The memory quantum is equivalent to the time a process *should* be loaded if the system is not swap saturated. The model could also account for the percentage of time during the execution of a workload in which the CPU is idle due to swapping. It might be assumed that a "large" CPU idle time due to swapping is undesirable. This idle time, however, may or may not be significant, depending upon the nature of the workload. If most processes are I/O bound, then the CPU will have a low utilization in any case. If the system is essentially compute bound, the idle time is significant to system throughput since productive work could potentially be performed during the swap. The

---

[6] A one-way swap is a swap-in or a swap-out.

determination of the memory quantum on an actual system was found to be difficult. Because of these problems, the model was abandoned.

3.3 The Basic Model

This section describes a simple deterministic model of a swapping system which is of use when the workload falls within the range of several workload and hardware parameters. Two extreme cases will be considered: the CPU bound case and the highly interactive, disk I/O bound case. To simplify the model, we will assume that the processes have the same CPU requirements and process size.

3.3.1 Workload and Hardware Parameters

A number of workload parameters will be introduced to characterize the workload in quantitative terms. Several significant hardware characteristics will also be selected and defined.

Hardware Parameters:

1. *MM*: the amount of main memory available to users in "clicks". Each click is a 64-byte block of primary memory (the minimum allocation size for PDP-11 memory management hardware). The value of *MM* excludes the size of the resident portions of the operating system.

2. *Tswap*: the mean length of time, exclusive of queuing delays, for a one-way swap. *Tswap* is defined to be the sum of the mean access time and the mean transfer time as a function of the size of the swap. Where an empirical result is available for this value it should, of course, be substituted for the theoretical value. When using a measured result for *Tswap*, it should be observed that the total number of processes also contributes to *Tswap* since longer seeks may be required.

3. *Tblock*: the mean time, exclusive of queuing delays, for one file system access (i.e., a 512-byte read or write).

Workload Parameters:

4. *PN*: the number of active processes. A number of relatively idle system programs (**/etc/update**, **/etc/init**, **/bin/sh**, the workload startup process, and the measurement process) will be excluded from this quantity.

5. *PS*: the swappable process size, in clicks. This value includes the user text and data segments, the stack and the user's system data.

6. *NL*: the maximum number of simultaneously loaded processes. This is defined to be:

$$NL = \left\lfloor \frac{MM}{PS} \right\rfloor$$

7. *PC*: the mean CPU requirement between I/O requests (i.e., the uniprogramming duration of a process' CPU burst). This value is not used by the basic model, but is used in the experiment to determine how *PC* influences the validity of the model.

8. *Nswaps*: the sum of the swap-ins and the swap-outs during a measurement period.

3.3.2 The CPU Bound Case

The case when most or all of the workload is CPU bound will be briefly mentioned here. It is assumed that round-robin scheduling is used and that processes are usually swapped-out after receiving their quantum time, Q. To simplify the following discussion, it will be assumed that processes are all of equal size. If the time quantum is also assumed to be at least as large as the time to replace a loaded process with one from secondary storage and that there are always at least two processes loaded ($NL > 1$), then since all swapping can be overlapped with computation (except for

scheduling/swapping code overhead and DMA related memory cycle competition), swapping will have virtually no impact on performance. (Note that response time can possibly be decreased without degrading throughput if Q is reduced to $2 * Tswap$; i.e., if the time-slice is equal to the time to ready a swapped-out process for execution. As Q decreases beyond this point, "thrashing" will occur because the amount of productive work performed between swaps decreases.) Two predictions based on the round-robin scheduling and equal process size assumptions may be made and experimentally verified for two systems, A and B, as follows:

1. If T_A is the running time on system A of a fixed amount of work when no swapping is required or when all swapping is overlapped with computation, then if $PN > 1$ and $NL=1$, the time to complete the same amount of work on system B will be $T_A + Tswap * Nswaps$ because no overlap is possible. The general case for arbitrary $Tswap$, Q, N, and NL under a compute workload will not be discussed here but the same reasoning may be applied to the general case. On an actual system, Q would normally be much longer than $2 * Tswap$.

2. Given $PN > 2$ compute bound processes of which $NL_A > 1$ are simultaneously loadable and $Tswap \leq Q / 2$, the time T_A to complete a fixed amount of productive work should be equal (within error) to the time T_B to complete the same work when $1 < NL_B \leq NL_A$. This implies that as long as there is sufficient memory to allow one process to execute while another is being swapped, swapping does not significantly impact system performance. It should take twice as long to complete double the amount of work (a linear increase in execution time with increasing PN). Adding extra memory to such a system will not improve performance nor will substituting a faster swap device since the CPU is the bottleneck.

3.3.3 The Basic Model

The model makes several assumptions about the characteristics of the workload, the operating system, and the hardware. The workload is assumed to be of a highly interactive nature, generating many file system requests and with relatively short CPU demands relative to the swap time. The volume of this type of workload is further assumed to be sufficient to cause the system to approach swapping saturation, although this requirement will be relaxed later. It is assumed that the shared swapping/file system controller becomes the limiting resource (i.e., bottleneck) to system performance. Many simple models are based on a limiting resource assumption of this type [35]. Such models are conceptually and analytically simple, but their usefulness depends on the validity of the original resource assumption.

It is assumed that round-robin processor scheduling will be employed, that only one scheduler initiated swap-in or swap-out can be queued at any one time, and that swap requests are serviced in the same manner as ordinary file system requests. It is also assumed for the time being that a single controller is shared by the disk devices involved (i.e., swapping is not overlapped with file system traffic). Since the disk subsystem is assumed to be the major bottleneck, other peripheral devices are ignored by the model. Many small to medium sized UNIX facilities meet these assumptions. The highly interactive nature of the workload (e.g., editing, graphics, plotting, file manipulation, communications, etc.) allows the swapping saturation assumption to be met, because processes waiting for a slow (relative to a disk) device are usually swapped-out. Also, given a sufficiently large workload of this type, think times may be ignored because the next interaction is often ready when the current one finishes.

The basic concept behind the model is that the ratio of the number of file system requests (i.e., useful work) queued between swap requests (overhead), adjusted for I/O

times, gives an approximation of the ratio of times to complete the same amount of useful work or the same number of interactions. If more file system accesses can be performed between swaps then more productive work is being done per unit time. The model assumes that, on average, each process has one file system request in the disk queue. The basis for this assumption is that when the CPU requirement of a process is short compared to the time to execute a one-way swap, a new request will be queued before the current swap-in or swap-out completes. Since the workload is highly interactive, as soon as a swap-in or swap-out completes, another will be queued. Note that as NL decreases, the swapping rate should increase until swap saturation because there is less file system traffic between swaps.

To illustrate a simple case, suppose that in configuration A there are four loaded processes and that in configuration B there are eight loaded processes. Assume that in either configuration the time to swap and the time to perform a file system request are respectively the same. Then because B performs about twice as much useful work between swaps as A, B will take less time to perform the same amount of work.

The analytic model is:

$$Ratio_{BA} = \frac{T_B}{T_A} = \frac{NL_A * NL_B * Tblock_B + NL_A * Tswap_B}{NL_A * NL_B * Tblock_A + NL_B * Tswap_A} \tag{3.1}$$

where T is the time to complete the workload.

The model predicts the total time to accomplish a given amount of work, once a measurement of the same type of workload under (possibly) different conditions is known (PN is the same in both cases). From this value, the mean system throughput can be determined ($X = C / T$) and the mean response time is $R = T / C - Z$ [10], where C is

the number of completed "interactions" and Z is the mean think time. For the interactive user, response time is usually of primary concern. Hellerman [17] suggests that a better performance indicator than response time is a statistic on the response relative to the service time required for the request (sometimes called the "stretch factor" or "dilation"). This is defined as the ratio of response to service time, or its reciprocal. The output of the model, the relative change in execution time (per workload), is analogous to the dilation statistic (per process). It is assumed that decreasing the time to perform a file system access will have no significant effect on the mean queue length; the disk controller will still be the bottleneck. Note that the utilization of the controller must be high (greater than about 80 percent) on both of the systems used in Equation 3.1.

The primary advantages of this model over previous models are its simplicity and deterministic nature. Chen's model [6], for example, requires the mean number of disk I/O requests per interaction to estimate routing frequencies. The probability of a process being swapped-in or swapped-out must also be estimated. Other modeling work is primarily concerned with the balancing of interactive load and batch load to improve system utilization.

Because the model developed in this chapter is deterministic, neither probabilities nor a separate queueing network solution step is required. There is also no need for any of the assumptions used in the theory of stochastic processes. The concepts behind the model are easy to understand and the model is simple to evaluate. Since the model is based on operational quantities, its assumptions can be tested and verified.

A weakness of the model is that it relies upon the assumptions of disk saturation and highly interactive workload.

4. The Experiment

4.1 Overview

In this chapter, the experiment conducted will be described in detail. Having decided to perform an experiment to evaluate the model, the first step is to design the parameters for a synthetic workload and validate the workload. Tools are then developed to measure, extract, and process the system data. The description of both the system modifications and the measurement program are included in this section. The design and implementation of the experiment are discussed.

The objective of any experimental investigation is to learn more about the system being investigated; i.e., to study the effect of variation of certain factors or the relation between certain factors in a system. The major difficulty in evaluating computer performance is the large number of factors in the problem under investigation. Extraneous factors which cannot completely be controlled influence the outcome of the evaluation. The utility of an experimental design is to eliminate these factors if possible, or to minimize their effects by arranging the experiment so that the effects may be expected to cancel and partially cancel each other in the analysis of the resulting empirical data. Experimentation is an iterative process. An initial conjecture leads to a design, the design leads to the experiment, and the experiment provides data that are analysed before forming a new conjecture.

4.2 Synthetic Workload and Model Parameters

When measuring the performance of a system, either the production workload or an artificial workload can be used to drive the system. The production workload, or a segment of it, has the advantage of being more representative. Because, however, the

workload varies with the time of day, the day of the week, etc., the question of a particular measurement's representativeness may arise. Artificial workloads offer reproducibility, flexibility, and are potentially more compact. They are, however, more expensive to construct, less representative, and require a dedicated system to run on.

In this work, the production workload was not suitable for direct measurement although monthly command frequency data suggested the more popular commands. For the purposes of this thesis, the synthetic workload is the most desirable choice because a controlled environment is needed to make comparisons between different configurations with the same workload. The changes in performance with changing workload characteristics are also of great interest. A prototype program which became the basis of the synthetic workload was therefore constructed.

4.2.1 Construction of the Prototype Process

A prototype process consists of 200 "interactions", each made up of the following:

1. A Compute Loop - the duration of this computation is a command line argument to the process when it is invoked.

2. Disk I/O Operations - a number of reads, writes, and seeks are usually performed, although on occasion no operation will be performed. Note that because of the buffering strategy, physical I/O may be delayed or may not be required.

3. Pause - to simulate think time, the process is put into a low priority wait for a short time. This would normally cause the process to be swapped-out. This was implemented by setting an alarm clock signal and then issuing a read request which could not be satisfied.

4. Terminal Output - several 25-character-long strings are printed. The first character of the string identifies the writing process. An interactive workload usually involves a great deal of terminal input and output. Because terminal input is difficult to simulate in a synthetic workload, terminal output was used to cause additional process suspensions. When the terminal output buffer fills, the process is suspended at a priority slightly lower than the input priority until the characters have been sent to the terminal.

5. Process Size - there is a dummy array in the prototype process which can be statically varied to alter the size of the process. The process size includes the stack and system data segments.

The amount of computation, disk I/O, and terminal output, as well as the duration of the pause are all distribution driven. This adds variability to the workload while keeping averages constant. In addition, the seed for the random number generator and the file used for I/O were different for each process. To create the synthetic workload, the prototype process was duplicated on a non-sharable basis to give the desired number of processes.

4.3 Description of the Measurement Tools

In order to extract data on the operation of a computer system over a period of time, measurement monitors are used to record the values of certain variables considered to be significant for evaluating system operation. Rose [32] discusses measurement procedures in detail. Monitors use either an event trace policy or a sampling policy and are implemented using hardware, software, or both. Event trace monitors record information when a particular event occurs while sampling monitors record information at

specified time intervals. The event trace policy usually obtains more detailed information over a shorter period of time. Data are written to disk or tape for subsequent use by data reduction software. Large amounts of data are often collected in the course of a measurement session and it is important to insure that the collection process does not interfere significantly with the system being observed. The sampling policy initiates data collection activities when a real-time clock signals the end of an interval. The interval, or time between successive sampling events, is usually constant. A less detailed description of the system is obtained over a longer period of time using sampling.

There are three basic implementations of measurement monitors, each of which may use either event trace or sampling policies. Hardware monitors are devices attached to the computer's circuitry at various points to examine statuses or count events. These monitors have the advantages of permitting high event rates, making precise measurements, and not interfering with the system being measured. They cannot, however, reveal the causes behind events and they lack selectivity of measurement. Software monitors are measurement routines inserted into the system software to record events and statuses. They can generally record any information available to the operating system and offer a great deal of flexibility in the selection of the measurement data. Because they require resources, they may interfere with the system operation which is to be measured. If a sampling policy is used, the sampling frequency has to be large enough to provide statistically good results, but must not unduly influence the measurements. The hybrid implementation combines hardware and software but has the disadvantage of requiring the monitor software to be able to communicate with the external hardware monitor.

A number of changes were made to the operating system to conduct the experiment. When the experimental version of UNIX is booted, it allows the user to specify

the amount of free user memory to be made available. This allows the parameter *NL* to be controlled by adjusting *MM*. The **pause** and **alarm** [18] system calls had previously been added to the system as well as the Version Seven **sleep** subroutine which uses these system calls. The user-process (**/etc/update**) which flushes the buffer system uses these new calls to avoid the inefficiencies detailed earlier caused by the original **sleep** system call. A measurement program called **iostat** was written which is similar to the Seventh Edition program of the same name [18]. Code was added to the kernel which combined an event trace policy with a sampling policy. Also, **/etc/update** was made into a pure text program so that it would never be swapped-out. The kernel was modified to sample system states at each clock interrupt (60 Hz) as well as to record various transactions. The clock interrupt handler was modified to record the status of either disk drive (busy or not busy) and the state of the CPU (busy user-mode, busy system-mode, or idle) each time it was called. The distribution of disk controller queue lengths was also maintained. Within the section of the system used to manage the block I/O buffers, code was added to count the number of cache hits, the number of reads, read-aheads, and writes. The routine to set up a swap was changed to count the number of swaps and the number of core clicks which were swapped. The disk driver was modified to count the number of requests directed to each drive and the number of words transferred by each drive. It was also responsible for marking the status of either drive for the purpose of sampling by the clock driver. The total number of seeks for either drive was maintained. The terminal driver recorded the number of characters read and the number printed. All the data were stored in a contiguous data structure so that the **iostat** user-process could access them easily.

This particular implementation of performing measurements was chosen because of its simplicity and because of constraints imposed by the system hardware (or lack

thereof). Measurement code could have been designed to write data to secondary storage, for example. This would allow much more detailed information to be extracted and studied. Since neither a tape drive nor an additional disk unit was available, simple operational data would have to suffice. The interference to the system caused by writing data to the disk devices being observed was deemed to be too large. It would also increase complexity and introduce another potential source of error.

Some of the quantities recorded are necessary to derive parameters for the model; others (such as system state distribution) were thought to be useful for the insights they might give into the experimental results and for possible future research.

The output of **iostat** consists of the number of disk reads, read-aheads, cache hits, disk writes, the mean queue length, and the disk queue length distribution. The total number of requests to each disk and the total amount transferred to and from each disk are printed. The mean access time, the mean transfer time, and the mean number of bytes per request are calculated from measurement data and from the manufacturer's device specifications. Disk utilization and the time the CPU spends waiting for I/O to complete are printed as well as a system-state/disk-drive-state matrix. The user may select various subsets of this information.

When **iostat** is invoked, it simply reads the measurement data from kernel space and notes the current time of day. It then suspends itself for a given amount of time using the **sleep** subroutine. When it awakens, it calculates the elapsed time, reads the data again, and displays performance parameters based on the difference between the two sets of data and the real time which elapsed between the readings. If **iostat** is interrupted before the given time period expires, it ignores the original time period and uses the elapsed time to the interrupt. Since the pause puts the process in a low priority wait, **iostat** will be swapped-out during the measurement session and therefore will not

interfere with it. The measurements within the kernel consist of simple increments and assignments and would contribute little to the system overhead.

The normal system clock is accurate to the nearest one sixtieth of a second. A more accurate clock is available within the DUP11 synchronous line interface. The maintenance clock in this interface has a resolution of between one and two milliseconds. A device driver and a system call were added to the kernel in a preliminary experiment to allow the maintenance clock to be used as a stopwatch. The clock was calibrated using the system clock and then the maintenance clock was used to develop distributions for the various *PC* intervals. The code was originally written to make more precise measurements and to pass data back to a user-process. A mechanism was developed whereby part of main memory could be reserved for collection of data. This implemented a simple, low interference trace facility.

4.4 Design of the Experiment

The workload and hardware parameters that the basic model requires were outlined in Section 3.3.1. When there are interactions among the parameters, it is necessary to use a factorial design for the experiment. In a factorial design all values of each parameter must be varied with all values of the other parameters. The model parameters become the factors of the experiment; i.e., those quantities which are explicitly varied. The different values of a factor which are used are termed the levels of a factor. The influences which are not of interest and which must be held constant during the experiment are called secondary factors. This section describes the levels chosen for the factors and the reasoning behind their choice. A complete factorial experiment was followed for the main levels, while several individual experiments were run with different levels in order to probe for the points where the model breaks down. Also, some combi-

nations of levels/factors were not of interest because no swapping would occur (i.e., $PN \leq NL$). The factors and levels are summarized in Table II.

Experimental Factors and Levels	
Factor	Levels
MM	650, 1000, 1280 (clicks)
PN	4, 5, 6, 8, 10, 12 (processes)
PS	157, 314 (clicks)
PC	12 (msec.)

Table II

Since the natural workload was not suitable for the experiment's workload or for determining reasonable levels for the factors, results from a previous experiment helped to provide some realistic values. Downing [13] measured the production workload on a similarly configured UNIX system. This workload was highly interactive. The mean number of processes, the process size distribution, and the distribution of uninterrupted CPU intervals were determined. It was found that on a busy system there were usually about 2 ready processes and from 12 to 14 blocked (i.e., waiting) processes. Most processes were smaller than 200 clicks (6K words) with up to 5 percent over 500 clicks in size. The mean uninterrupted CPU interval of that production workload was found to be 12 msec. Although the *PC* findings were from a differently configured system, the same levels were used in this work since they were found to be less than, close to, and greater than the one-way swap service times used.

The level of 12 msec. for the *PC* factor in the factorial experiment was chosen. Several supplementary runs were made with larger *PC* values to determine when the CPU demands begin to violate the model's workload assumptions. The uninterrupted

CPU requirements were implemented by calling a function having a fixed delay several times based on a distribution. All *PC* distributions used in the experiment are given in Table III.

Distribution of Compute Loops (Percent of Occurrences)				
	Number of Iterations			
PC (msec.)	5	20	200	5000
12	3	36	100	100
30	1	19	97	100
60	1	5	90	100
186	1	5	57	100
354	1	5	15	100

Table III

The maximum level for main memory is limited by the maximum amount of free memory available on the system. The lower level is based on providing just enough memory so that two of the larger processes could be simultaneously loaded. Several runs were made with a process size of over 400 clicks so that swap times could be increased.

The distribution for the disk activity was adjusted so that the utilization of the controller exceeded 90 percent when the maximum level of *PN* was used. It could be argued that the amount of file system traffic generated (about 6 blocks read/written per interaction) was perhaps greater than that which would be expected from a typical interactive process. For example, no think times exceeded 3 seconds. This sacrifice in representativeness was necessary because there was insufficient main memory to create a high utilization when the larger process sizes were run and because it became too difficult experimentally (and too lengthy) to use a level of more than 12 processes. Many

similar installations run at the maximum main memory capacity (more than twice the user memory available on this system), naturally creating high controller utilization.

The internal system parameters, such as the number of buffers, the scheduling constants, priorities, etc., were held constant throughout the experiment.

The swapping and file system accesses were directed to separate disk drives to make the mean time to perform either type of disk request easy to determine. It is common for a UNIX installation to assign user files to a different drive than that used for swapping. It would not violate any assumptions made by the model if this were not the case. All terminal output was sent to the same 9600-baud CRT terminal.

Ideally, it would be better to control *NL* and *Tswap* separately. This would allow runs to be made where a slower disk could be studied with the same number of loaded processes. Because of the introduction of the extra factors and the extra complexity, this enhancement was not implemented. Further confidence in the model would be obtained by conducting an additional experiment on a differently configured UNIX system.

The identical process size of each of the prototypes is, of course, not realistic. This simplification made the *NL* parameter easy to calculate and *Tswap* easier to measure, reducing the complexity of the experiment. Calculating *NL* in an actual system involves scanning through the system process table, where the mean process size can also be determined for the calculation of *Tswap*.

4.5 Implementation

The implementation of this experiment involved changing the operating system as described in Section 4.3, writing the user-process measurement program, and creating the prototype processes. Before running the experiment, the length of a session had to be determined. It is assumed that the accuracy of the values gathered within the system

are ergodic; i.e., that the accuracy of the estimations increases as the number of observations in the series increases. A sample workload was run several times until the number of iterations within each prototype was enough to make elapsed times for each run reasonably close. The value of 200 iterations provided experimental runs not excessively long and with differences of less than 5 percent in elapsed time between identical workloads.

To reduce the possibility of introducing error, to simplify the implementation, and to minimize possible variations in the start-up procedure, each run was initiated by a command file and a special start-up program. The start-up program simply read from the command file, invoked each prototype process with its arguments, and then waited until the last process completed before displaying the elapsed time. Once the workload had stabilized, the **iostat** program was invoked from a second terminal. The stabilization criterion was the same for each workload. Since each process identified itself when it printed its terminal output, it was possible to tell when each of the processes had begun to execute. At this point **iostat** was invoked. Upon completion of the start up program, **iostat** was manually interrupted, causing it to display the performance data and then terminate. After a run was completed, the scratch files were reset to their original contents.

5. Results and Analysis

The results of the experiment are presented and the accuracy and utility of the model are examined in this chapter.

5.1 Experimental Results

The measured and calculated values for swap times and block I/O times are given in Table IV and Table V, respectively.

Mean Swap Times			
PS (clicks)	Measured (msec.)	Calculated (msec.) Min	Max
157	50.7	91.1	118.6
314	69.8	114.7	129.7
425	93.7	146.4	161.4
470	106.5	153.2	168.2

Table IV

Mean Block Read/Write Time (*Tblock*)	
Measured (msec.)	Calculated (msec.)
39.8 ±2.5	68.7

Table V

The large difference between the measured block I/O times and the times obtained from the manufacturer's specifications can be attributed to the physical closeness of the

scratch files used by the prototype processes and to the LOOK disk scheduling algorithm.

The measured swap times above are also lower than the calculated times due to the locality of the seeks. In fact the difference between the mean measured values and the mean calculated values is approximately the mean seek time. The measured times are approximate because they vary with *PN*; as *PN* changes, the number of track-to-track seeks may change. The measured values are used whenever possible in subsequent results.

The results of the factorial part of the experiment are presented in Table VI. The measured value *Qdisk* is the mean queue length at the disk controller, including the request currently being serviced. The experimental error in the measurement of *Qdisk* was found to be ± 0.1.

Results of the Factorial Experiment
$PC = 12$ msec.

PS	*MM* (clicks)	*PN*	*NL*	*Qdisk*	*Nswaps*	Controller Utilization (%)	Elapsed Time (secs.)
157	650	12	4	4.8	3554	95	717
157	650	10	4	4.7	3016	95	613
157	650	8	4	4.6	1937	91	439
157	650	6	4	4.4	1202	88	358
157	650	5	4	3.4	660	78	315
157	1000	12	6	7.0	2336	96	637
157	1000	10	6	7.2	1889	94	530
157	1000	8	6	6.3	1031	89	423
157	1280	12	8	8.6	1833	95	623
157	1280	10	8	8.2	1477	93	521
157	1280	8	8	6.5	438	86	403
157	1280	6	6	3.9	5	77	311
157	1280	5	5	2.9	4	68	298
314	650	12	2	2.7	4095	96	1032
314	650	10	2	2.8	3590	95	838
314	650	8	2	2.7	2819	94	670
314	650	6	2	2.7	2027	91	510
314	650	5	2	2.6	1602	90	433
314	650	4	2	2.3	1064	81	353
314	1000	12	3	4.1	3815	96	920
314	1000	10	3	4.2	3262	97	761
314	1000	8	3	4.2	2500	95	613
314	1000	6	3	4.1	1630	92	453
314	1000	5	3	3.6	1119	86	387
314	1000	4	3	2.7	531	71	307
314	1280	12	4	4.7	3493	96	854
314	1280	10	4	4.8	2865	97	721
314	1280	8	4	4.3	2093	87	669
314	1280	6	4	4.5	1316	91	422
314	1280	5	4	3.9	691	82	346
314	1280	4	4	2.4	198	64	289

Table VI

Table VII shows the results of several runs where the CPU interval was increased from 12 msec.

Supplementary Experimental Results

PC (msec.)	*PS* (clicks)	*MM*	*PN*	*NL*	*Qdisk*	*Nswaps*	Controller Utilization (%)	Elapsed Time (secs.)
30	157	1280	12	8	8.2	1793	95	610
60	157	1280	12	8	8.1	2046	93	665
186	157	1280	12	8	6.1	2773	89	797
354	157	1280	12	8	3.8	3240	75	1025
354	157	1280	5	5	1.6	4	40	534
30	314	1280	12	4	4.7	3598	96	882
60	314	1280	12	4	4.6	3813	96	908
186	314	1280	12	4	3.7	3935	89	994
12	425	1280	12	3	3.8	4029	98	1045
30	425	1280	12	3	3.8	4031	98	1072
60	425	1280	12	3	3.6	4050	95	1095
186	425	1280	12	3	2.9	4053	86	1205
12	470	1280	12	2	2.9	3921	97	1047
186	157	650	12	4	3.4	4425	83	987
12	157/314	1280	12	~5	5.7	2939	94	724

Table VII

One of the supplementary runs in Table VII consisted of 6 processes of size 157 and 6 of size 314 clicks. The mean number of loaded processes was estimated to be 5.

5.2 Analysis of the Model

Figure 1 illustrates the output of the model versus measured values for several

different sets of parameters. Table VIII shows the standard deviation[7] using five of the measurements to predict a sixth.

Standard Deviation in the Predictions		
Measured Elapsed Time (secs.)	Mean Prediction (secs.)	Standard Deviation (secs.)
623 ±31.2	684.2	32.5
637 ±31.9	680.6	30.0
717 ±35.9	735.1	38.3
854 ±42.7	787.3	30.0
920 ±46.0	871.2	41.5
1032 ±51.6	1042.8	55.5

Table VIII

The linear trend in elapsed time with changing *PN* is not directly taken into account by the model; *PN* is assumed to be constant for both workloads used by the model. In Figure 2, a graph is plotted of elapsed time versus *PN*. The linearity in the elapsed time is due to the saturation of the disk controller. The relationship between the bottleneck device and both response time and throughput is derived by Denning and Buzen [10]. The fact that the linearity is observed at the lowest *PN* values ($PN = 4$) suggests that the controller is approaching saturation even at that point.

A second extension to the model involves changes of the *PC* parameter. As *PC* increases, the queue length at the disk controller decreases. This occurs because there is more computation done between disk I/O requests and since *PN* is not changed, the mean queue length must decrease. Similarly, as *PC* decreases, the queue length must increase (up to a point) since requests are being added to the queue after a shorter

[7] The standard deviation was calculated using the mean of the predictions as the true mean and the predictions as the samples.

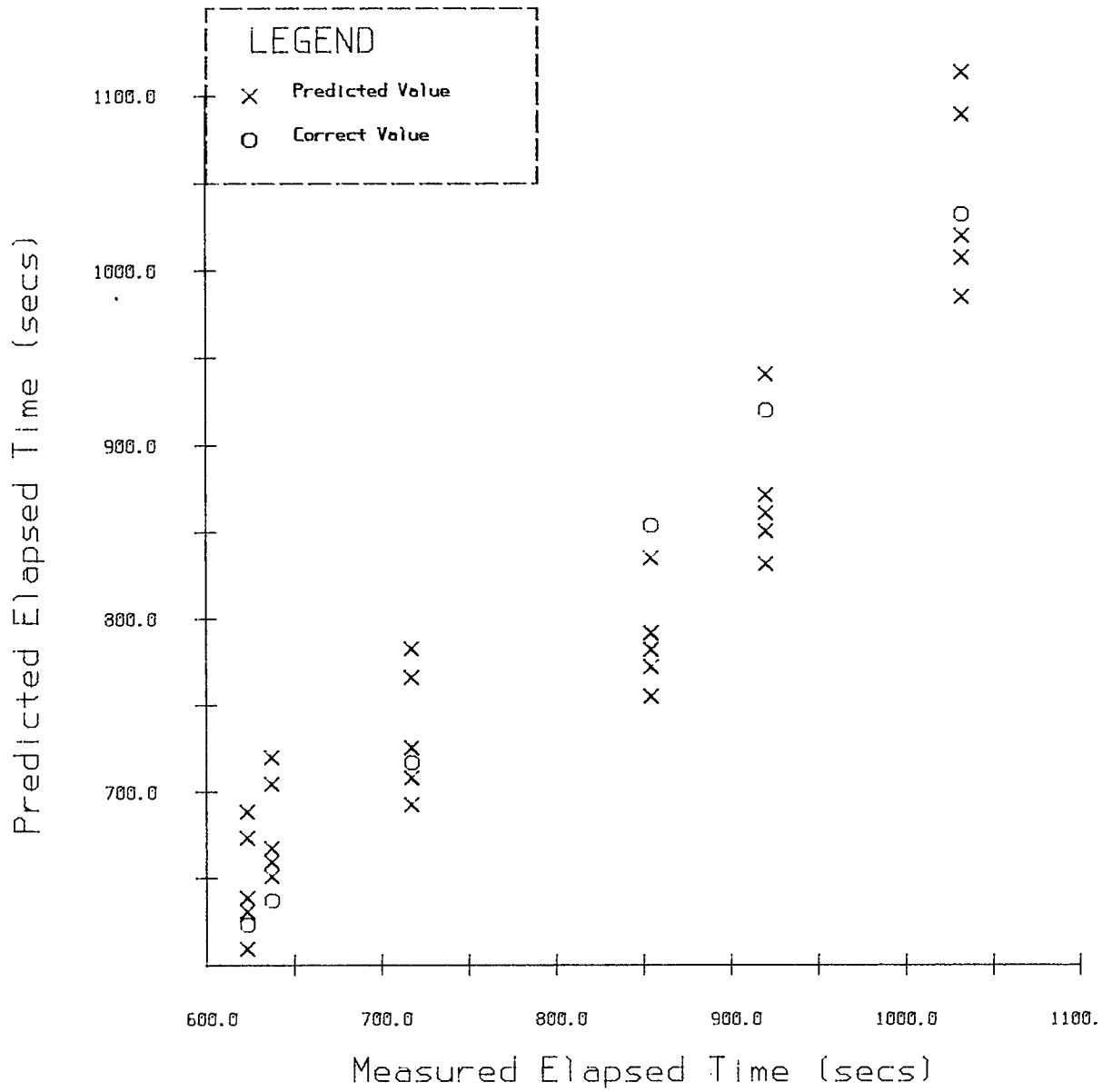
LEGEND
Predicted Value
Correct Value
1100.0
1000.0
900.0
800.0
700.0
Predicted Elapsed Time (secs)
600.0
700.0
800.0
900.0
1000.0
1100.0
Measured Elapsed Time (secs)

Figure 1

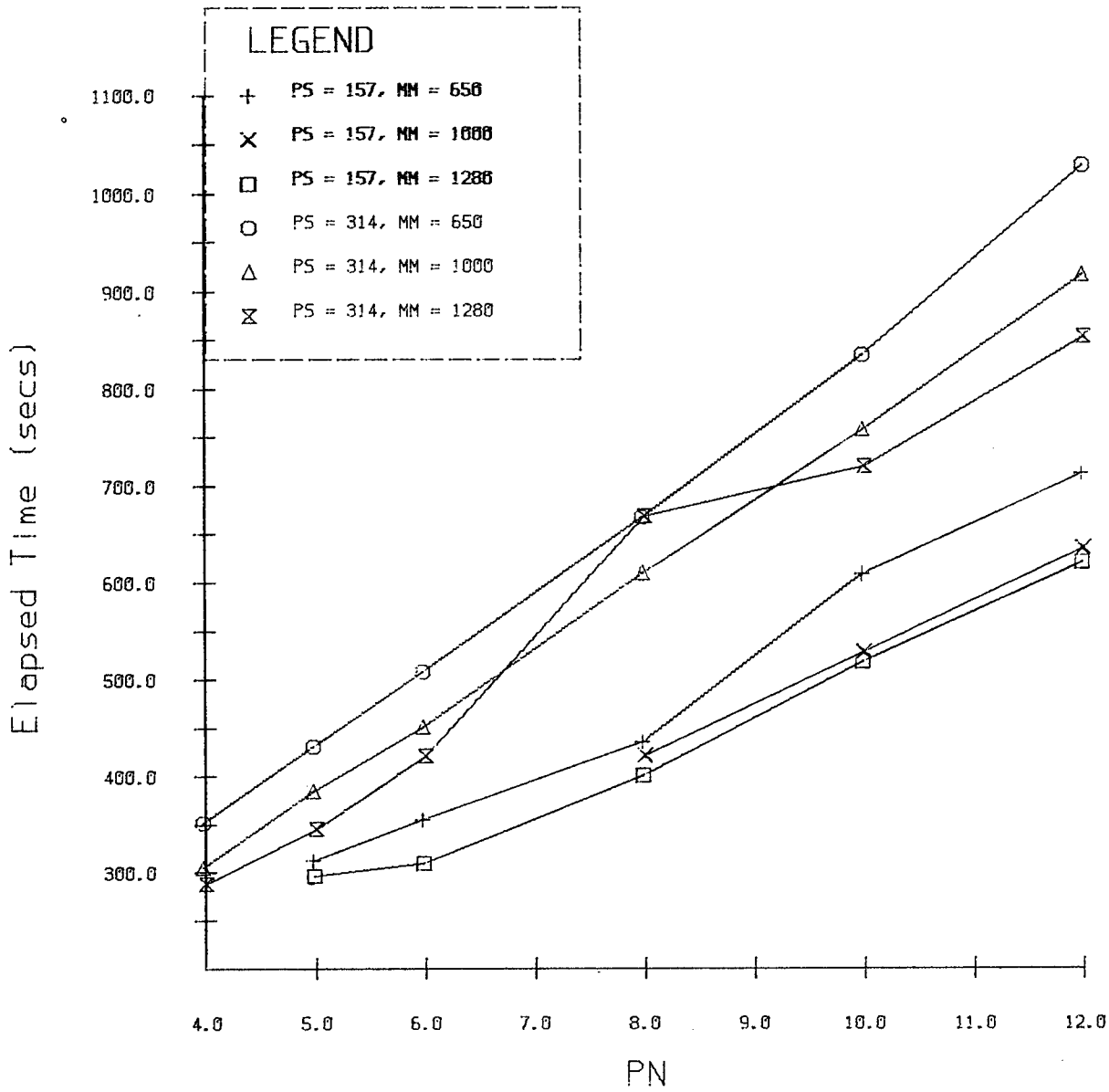
LEGEND
PS = 157, MM = 650
PS = 157, MM = 1000
PS = 157, MM = 1280
PS = 314, MM = 650
PS = 314, MM = 1000
PS = 314, MM = 1280
Elapsed Time (secs)
1100.0
1000.0
900.0
800.0
700.0
600.0
500.0
400.0
300.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
11.0
12.0
PN

Figure 2

period of time. The increase becomes most significant when *PC* exceeds *Tswap*. It was discovered that the ratio of queue lengths provides a good approximation of the ratio of elapsed times between two workloads which are the same except that one has a large *PC* and the other a small *PC*.

For example, during the run (157,12,12,4)[8] the mean queue length was observed to be 4.8 and the elapsed time was 717 seconds. During a different run (157,186,12,4), the queue length at the controller was 3.4 and the elapsed time was 987 seconds. The ratio 4.8 / 3.4 is, within experimental error, 987 / 717.

The mean queue length, however, must be known for the workload with the larger *PC* since there appears to be no simple way to predict the change in queue length with a change in *PC*. The relationship seems to break down in the run where $PC = 354$ msec.; the prediction from this data greatly exceeds the measured elapsed time. Presumably this is because the disk saturation assumption is violated or perhaps another factor which is not taken into account by the model starts to become influential when *PC* is sufficiently large.

The mean queue length is influenced by both the buffering mechanism and by **/etc/update** which flushes the buffers periodically. Because of this, the observed swapping rate does not agree well with the rate predicted by the model. It also appears to be difficult to predict what the queue length at the controller will be as *PC* is increased beyond *Tswap*. This would probably make the model, as it is, of little use when dealing with compiling-type workloads.

5.3 Sources of Error

[8] The notation (*PS,PC,PN,NL*) will be used in the following discussion to describe the experimental parameters.

There are several potential sources of error in the experiment. Differences in both the allocation of file space and swap space between runs introduce a certain amount of error. It is possible that some events were missed because sampling was done at 60Hz. Errors in the manufacturer's specifications are also possible. A certain amount of human error is introduced by the need to manually interrupt the measurement program when the workload finishes. These sources of error probably account for the 5 percent fluctuation in elapsed time mentioned earlier. A few experimental runs were redone several times when the accuracy of the first run was in doubt. Either mean values were used in such cases or the questionable result was discarded.

5.4 Applications

The following hypothetical situations are presented to indicate areas where the model and its extensions may find application. In each example it is assumed that both workloads are fairly uniform and consist mainly of interactive programs or many short programs which tend to saturate the disk controller.

Suppose there was a UNIX system which had 1280 clicks of available memory, an average of 10 user processes, 4 of which could, on average, be resident. Assume that *Tblock* is 0.05 sec., *Tswap* is 0.07 sec. and that the think time is negligible. The installation would like to estimate the effect of doubling the user memory on mean response time. Substituting into Equation 3.1,

$$Ratio = \frac{8 * 4 * 0.05 + 4 * 0.07}{8 * 4 * 0.05 + 8 * 0.07}$$

$$= 0.87$$

or a 13 percent decrease in elapsed time.

A similar method could be followed to determine the influence of a change in *Tblock* and/or *Tswap*. Suppose the installation would like to estimate the effect of decreasing *Tblock* to 0.04 sec. and *Tswap* to 0.055 sec. Substituting into Equation 3.1,

$$Ratio = \frac{4 * 4 * 0.04 + 4 * 0.055}{4 * 4 * 0.05 + 4 * 0.070}$$
$$= 0.80$$

or a 20 percent decrease in elapsed time.

Suppose that swapping effects were eliminated altogether by keeping all processes loaded or by the addition of extremely fast swapping hardware. From Equation 3.1,

$$Ratio = \frac{10 * 4 * 0.05 + 4 * 0.00}{10 * 4 * 0.05 + 10 * 0.07}$$
$$= 0.74$$

the model predicts a 26 percent decrease in elapsed time.

Because of the saturation assumption, the increase or decrease in elapsed time with a change in *PN* may be assumed to be linear; e.g., doubling *PN* should approximately double the completion time. If *Tblock* was decreased to 0.04 sec. and *PN* was increased by 50 percent, then

$$Ratio = \frac{4 * 4 * 0.04 + 4 * 0.07}{4 * 4 * 0.05 + 4 * 0.07} * \frac{15}{10}$$
$$= 1.28$$

and so the elapsed time would be expected to increase by about 28 percent.

If the mean queue length at the controller is known or can be estimated for a change in the CPU requirements, then the ratio of queue lengths can be used to estimate the change in elapsed time. Suppose, for example, that the installation would like to estimate the effect of decreasing *Tswap* to 0.06 sec. while at the same time increasing *PC* from 0.012 sec. to 0.05 sec. and increasing *NL* to 6 by the addition of memory. The queue length at the disk controller in the new configuration will be estimated to be $NL + 1$, since *PC* will be less than *Tswap*. The expected change would be approximately:

$$Ratio = \frac{4 * 6 * 0.05 + 4 * 0.06}{4 * 6 * 0.05 + 6 * 0.07} * \frac{5}{7}$$

$$= 0.63$$

6. Conclusions

6.1 The Model and Validation

A simple, deterministic model of a disk-saturated swapping system was developed and validated through a controlled experiment. The model was found to provide a good approximation of the ratio of elapsed times between two workloads with differing parameters. As predicted by theory, the change in elapsed time was linear with the change in the number of processes when the disk controller approached saturation. When the CPU requirements of the processes were increased, within limits, the ratio of queue lengths at the disk controller was found to be a good approximation of the ratio between elapsed times.

Only one hardware configuration was used to keep the size of the empirical validation reasonable. A synthetic workload had to be created because the natural workload was unsatisfactory for validation purposes. Several sample applications were described. An important characteristic of the model is that since it is deterministic, there is no need to determine device routing probabilities. Although the model is simple, it can be used to determine the change in performance with changes in the hardware configuration. Since the size of main memory is constant, the mean number of processes and the mean process size can be determined from only one pass through the system process table. The mean disk I/O times can be easily measured from within the system or can be calculated using the manufacturer's specifications.

6.2 Tools and Method

The data measurement and extraction tools for the experiment proved very satisfactory for the nature of the information being collected. The tools are straightforward

and because they are written in C, are quite portable and easily understood. The overhead introduced by the tools is negligible.

The factorial nature of the experiment, chosen to test the model uniformly, proved satisfactory when supplemented by several additional runs. As mentioned above, testing the model under various hardware configurations is desirable, but turned out to be impractical. Similarly, additional factors would significantly increase the size of the experiment.

6.3 Domain of Applicability

Perhaps the greatest simplification in the experiment was the use of an homogeneous synthetic workload. While this does not reflect reality, one experiment where the process sizes were not identical gave a result close to that predicted by the model (see the last entry, Table VII). Although more work should be done on this, it is felt that the model will continue to be useful as long as there are not large differences in the CPU requirements of the processes which make up the workload. Additional work could investigate the effect pure, sharable code has on the accuracy of the model.

Regarding workload, the domain is limited to highly interactive environments where CPU requests are all much less than one second; i.e., where the workload consists mainly of editing, file manipulation, and execution of programs which are either short or which interact with the terminal. Obviously the model is restricted to systems which employ swapping.

Another departure from reality is the absence of terminal input. This was not considered too significant because a good approximation using terminal output was used.

Because newer versions of UNIX use similar scheduling and swapping strategies, the model could be applied to them as well. Testing this hypothesis, however, was not

possible.

6.4 Future Research

The results of this study could be used as a basis for research into the following:

1. A more general model, not relying on disk saturation,
2. Expanding the model to account for large PC values so that compiling-type workloads can be considered,
3. An examination of the differences in performance between swapping and paging on a small machine where the disk is the limiting resource,
4. Testing the model on a different hardware configuration, perhaps one with overlapped file I/O and swapping capability,
5. Studying the effectiveness of a processor scheduling algorithm which attempts to balance the characteristics of the loaded processes to reduce the swapping load.

1. BELADY, L.A., AND KUEHNER, C.J. Dynamic space-sharing in computer systems. *Communications of the ACM 12*, 5 (May 1969), 282-288.

2. BERNSTEIN, A.J., AND SHARP, J.C. A policy-driven scheduler for a time-sharing system. *Communications of the ACM 14*, 2 (February 1971), 74-78.

3. BUZEN, J. P. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM 16*, 9 (September 1973), 527-531.

4. BUZEN, J. P. Fundamental operational laws of computer system performance. *Acta Informatica 7*, 2 (1976), 167-282.

5. CHANDY, K. M., AND SAUER, C. H. Approximate methods for analyzing queueing network models of computer systems. *Computing Surveys 10*, 3 (September 1978), 281-317.

6. CHEN, P. P. Queueing network model of interactive computing systems. *Proc. of the IEEE 63*, 6 (June 1975), 954-957.

7. COFFMAN JR., E. G. Analysis of two time-sharing algorithms designed for limited swapping. *Journal of the ACM 15*, 3 (July 1968), 341-353.

8. DEITEL, H. M. *An Introduction to Operating Systems*, Addison-Wesley, Reading, Mass., 1983.

9. DENNING, P. J. Virtual memory. *Computing Surveys 2*, 3 (September 1970), 153-289.

10. DENNING, P. J., AND BUZEN, J. P. The operational analysis of queueing network models. *Computing Surveys 10*, 3 (September 1978), 225-261.

11. DENNING, P. J., KAHN, K. C., LEROUDIER, J., POTIER, D., AND SURI, R. Optimal multiprogramming. *Acta Informatica 7*, 2 (1976), 197-216.

12. DOWDY, L. W., AGRAWALA, A. K., GORDON, K. D., AND TRIPATHI, S. K. Computer performance predictions via analytic modeling–an experiment. *Conference on Simulation, Measurement and Modeling of Computer Systems*, August 1979, 13-28.

13. DOWNING, R. *A Performance Experiment on a UNIX System,* M.Sc. Thesis, University of British Columbia, 1979.

14. FERRARI, D. *Computer Systems Performance Evaluation,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

15. GRAHAM, G. S. Queueing network models of computer system performance. *Computing Surveys 10,* 3 (September 1978), 219-224.

16. GRENANDER, U., AND TSAO, R. F. Quantitative methods for evaluating computer system performance: a review and proposals. In *Statistical Computer Performance Evaluation,* W. Freiberger (Ed.), Academic Press, New York, N.Y., 1972, 3-24.

17. HELLERMAN, H. Discussion of session II. In *Statistical Computer Performance Evaluation,* W. Freiberger (Ed.), Academic Press, New York, N.Y., 1972, 99-200.

18. KERNIGHAN, B. W., AND MCILROY, M. D. *UNIX Programmer's Manual,* Seventh Edition, Vol. 1, Bell Labs, January 1979.

19. KIENZLE, M. G., AND SEVCIK, K. C. Survey of analytic queueing network models of computer systems. *Conference on Simulation, Measurement and Modeling of Computer Systems,* August 1979, 113-229.

20. KLEINROCK, L. Swap-time considerations in time-shared systems. *IEEE Transactions on Computers C-29,* 6 (June 1970), 534-540.

21. LAZOWSKA, E. D. The benchmarking, tuning, and analytic modeling of VAX/VMS. *Conference on Simulation, Measurement and Modeling of Computer Systems,* August 1979, 57-64.

22. LIONS, J. *A Commentary on the UNIX Operating System,* University of New South Wales, 1977.

23. MCKINNEY, J. M. A survey of analytical time-sharing models. *Computing Surveys 1,* 2 (June 1969), 105-216.

24. MUNTZ, R. R. Analytic modeling of interactive systems. *Proc. of the IEEE 63*, 6 (June 1975), 946-953.

25. MUNTZ, R. R. Queueing networks: a critique of the state of the art and directions for the future. *Computing Surveys 10*, 3 (September 1978), 353-359.

26. NIELSON, N. R. An analysis of some time-sharing techniques. *Communications of the ACM 14*, 2 (February 1971), 79-90.

27. PDP-11 Processor Handbook. Digital Equipment Corporation, 1981.

28. Peripherals Handbook. Digital Equipment Corporation, 1981.

29. RITCHIE, D. M. The UNIX I/O system. Bell Labs Internal Memorandum, 1974.

30. RITCHIE, D. M. A retrospective. *Bell System Technical Journal,* Part 2, July/August 1978, 1947-2969.

31. RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *Bell System Technical Journal,* Part 2, July/August 1978, 1905-2930.

32. ROSE, C. A. A measurement procedure for queueing network models of computer systems. *Computing Surveys 10*, 3 (September 1978), 263-280.

33. SCHERR, A. L. *An analysis of time-shared computer systems,* Research Monograph No. 36, The M.I.T. Press, Cambridge, Mass., 1967.

34. SHEMER, J. E., AND HEYING, D. W. Performance modeling and empirical measurements in a system designed for batch and time-sharing users. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 17-26.

35. STRAUSS, J. C. A simple thruput and response model of EXEC8 under swapping saturation. Proc. AFIPS 1971 FJCC, Vol. 39, AFIPS Press, Montvale, N.J., pp. 39-49.

36. TEORY, T. J. Properties of disk scheduling policies in multiprogrammed computer systems. Proc. AFIPS 1972 FJCC, Vol. 41, Part I, AFIPS Press, Montvale, N.J., pp. 1-21.

37. TEORY, T. J., AND PINKERTON, T. B. A comparative analysis of disk scheduling policies. *Communications of the ACM 15,* 3 (March 1972), 177-284.

38. THOMPSON, K. UNIX implementation. *Bell System Technical Journal,* Part 2, July/August 1978, 1931-2946.

39. TSUJIGADO, M. Multiprogramming, swapping and program residence priority in the FACOM 230-60. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 223-228.