T H E   U N I V E R S I T Y   O F   M I C H I G A N

Technical Report 4

PROGRAM BEHAVIOR AND CONTROL IN VIRTUAL STORAGE COMPUTER SYSTEMS

Tad Brian Pinkerton

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

April 1968

PROGRAM BEHAVIOR AND CONTROL IN VIRTUAL STORAGE COMPUTER SYSTEMS

by
Tad Brian Pinkerton

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in
The University of Michigan
1968

PREFACE

This research was carried out with the cooperation and assistance of many people. In particular, the author wishes to thank Professors B. W. Arden and F. H. Westervelt of the Computing Center, Professor K. B. Irani of the Systems Engineering Laboratory, and Professor B. M. Hill of the Mathematics Department, for many enlightening discussions. Michael Alexander of the Computing Center provided considerable help in the creation of the MTS data collection facility and in interpreting the data obtained with it.

The author is grateful for financial assistance provided during the period of this research by an IBM Information Science Fellowship administered by the Computing Center. Use of the computer for developing data reduction programs and analyzing the MTS data was supported in part by the Computing Center and in part by the Rome Air Development Center, Air Force Systems Command, under contract number AF 30(602)-3953 with the Systems Engineering Laboratory of the Department of Electrical Engineering.

This dissertation was prepared on the computing system studied herein [using text-processing and display editing programs and a special print train] as part of a research project in the use of conversational and graphical computing techniques. This project was supported by the Advanced Research Projects Agency of the Department of Defense (contract number DA-49-083 OSA-3050, ARPA order number 716) administered by the Office of Research Administration.

Finally, the author is especially indebted to Professor B. A. Galler for his patient encouragement, gentle criticism, and continuing professional guidance; and grateful to Professor R. C. F. Bartels, under whose devoted guidance the Computing Center is a dynamic and inspiring environment for research.

Table of Contents

List of Illustrations

List of Appendices

SYMBOLOGY

The following notation is used throughout this paper with the indicated interpretation.

() [ ] {}	sets of parentheses
Q(i,j)	subscription of Q with i and j
A/B	quotient of A and B
C^{q}	exponentiation of C by q
p ==> q	implication of q by p
$\sum_{i=1}^{k}$	summation operator
$\prod_{j=0}^{n}$	multiplication operator
$\int_{x}^{\infty}$	definite integral from x to infinity
	end of proof of lemma or theorem
[14]	bibliographic reference

INTRODUCTION

"One man's constant is another man's variable."

A. J. Perlis

"If a man will begin with certainties he will end with doubts, but if he will be content to begin with doubts he shall end in certainties."

Francis Bacon

I. GENERAL DISCUSSION

The subject matter of this study is the allocation of storage to digital computer programs. The basic point of departure is that of operating system efficiency. We are primarily concerned with the productivity of the system rather than its response; the effectiveness with which it performs rather than the effectiveness with which it services an individual customer.

Most large-scale computing systems in use early in this decade operated in "single-thread batch" mode: a single program used the central processing unit (CPU) and main storage until it was completely finished, whereupon the next program was introduced and allowed to run to completion, etc. During the processing of input-output (I/O) commands for a program, including program loading, the CPU and most of the rest of the system stood idle. As computing facilities grew larger and faster, the disparity between internal processing rates and slow I/O rates caused these more expensive systems to remain idle for an increasing share of the time.

Multiprogramming, or sharing system resources among many programs at once, was introduced to improve the performance of a system. With a large enough main storage, a system could deal with a number of programs in the execution phase concurrently, dispatching a new one to the CPU whenever the previous task paused for I/O activity. Since both processor rates and program sizes continued to increase, however, a large-scale system could still not store enough complete tasks in main memory to keep the CPU busy. This was true despite the fact that more processor time and additional circuit complexity was necessary to allow the system to operate with many programs at once.

In an attempt to ready more tasks for the CPU, systems have begun to store only parts of a program in main memory at any given time with sections removed and added as computation progresses. Programming techniques have been devised to keep track of the various parts of a task in main and auxiliary storage and to provide for their flow back and forth on a demand or scheduled basis [21,24,53]. However, because of the allocation problems introduced by having segments of various sizes moving in and out of a fixed-size main storage device with arriving segments required to be linked to the parts already there, conventional multiprogramming systems still leave a great deal to be desired in their ability to maximize CPU utilization. In fact, due to the overhead introduced by their necessary complexity, these systems may be outperformed in some situations by their simpler predecessors.

The most exciting, but controversial, development in recent storage allocation techniques is the virtual storage concept [4,5,19]. This organization provides each task in the system with an addressable storage space in excess of its total maximum demand. Each task, in effect, can be organized as if it were to run alone on a very large (non-existent) machine. The virtual memory concept is implemented on existing machines by partitioning main and auxiliary real memory and each task's virtual address space into one or more fixed-size units, called pages. Special relocation hardware and programming techniques are then used to translate the virtual address into a real storage address whenever a memory reference occurs during the execution of a task. If the actual address lies in a page stored in auxiliary memory, then execution of the task must be interrupted until the page can be moved to main storage; and we say that a "page fault" has occurred. In principle, it would be better to use variable-size pages, so that the allocation of storage could be made to correspond more directly to program structure. However, the relocation hardware required for fixed-size pages is already complicated, and the additional generality needed for variable-size pages seems difficult to obtain with current techniques.

Development of virtual memory systems has come to be associated with time-sharing, a departure from the traditional "batch run" mode of using large-scale computing systems. Because a number of human beings are simultaneously communicating directly with the system, an additional constraint—that of rapid response to each request for service—is added to the operating system discipline. A machine for time-sharing requires more multiplexing and communications-oriented equipment for handling I/O requests, and more sophisticated treatment of interrupts. To take advantage of economies of scale, such computing systems are usually large enough to require multiprogramming. However, time-sharing itself requires no essentially new storage allocation method: the constraint of fast response simply requires that allocation be efficient, and that control be switched more often among tasks. It also follows that the virtual storage concept is not restricted to time-sharing systems, since it provides a good solution to the storage allocation problem inherent in multiprogramming organizations for conventional systems.

II. THE PROBLEM AREA

We first note that despite rather radical changes in storage allocation techniques in going from a small single-thread batch system to a large, time-shared virtual storage system, the type of storage allocation problem remains much the same. The data and instructions of a task in the batch system were moved into core prior to starting the job and may have been overlaid by other segments as the execution progressed. The CPU waits during I/O were of concern because the CPU had to remain idle. In the virtual storage system, tasks tend to be larger but the amount of main storage devoted to any one of them at a given time may be smaller than that occupied by the batch system task. Thus the flow of information into and out of main storage proceeds at a faster rate, and a greater percentage of tasks are subject to such page-swapping. Task I/O waits do not necessarily idle the CPU, but they may still do so if the system cannot keep enough tasks in main storage so that at least one is usually ready to use the CPU.

We can view the process of executing a task in both the small batch and large virtual storage systems as a sequence of time intervals, in each of which a portion of the system resources are used by the task. During each interval in which a task actually uses the CPU, it uses the most expensive system component and progresses toward completion. Intervening periods occur in which the task is not ready to use the CPU and still uses part of main storage, and perhaps I/O and other devices. Since this time does not contribute toward that task's completion, it is here in particular that extra cost to the system occurs. The storage allocation problem is, simply stated, that of minimizing the costs associated with these interruptions in service.

In the single-thread batch system the cost to the system is easily computed—it is worth as much as an equivalent amount of straight execution. But when the system can make part of the wait time productive by allocating the CPU to another task, the cost becomes a function of the way in which each of the tasks currently in contention for the CPU affects the ready status of the others. Thus the cost depends on the way in which these tasks call for I/O and use main storage, and on the way in which storage is allocated by the system, and hence on the number of tasks which can use main storage at one time. The time-sharing requirement for rapid response simply accelerates the rate at which storage allocation events occur, and thus increases the likelihood that all tasks may be waiting for more space or I/O events at any given moment.

III. THE PROBLEM

This thesis addresses the problem of increasing productivity in a computing system by improving storage allocation. The problem is attacked with stochastic models of system and task behavior. Where possible, storage allocation delays and holding times are characterized by a cost function. The models are used to minimize the total expected value of the cost.

The study begins with a particular problem in a small, single-thread batch system. A large task may require a storage space greater than main memory, and as a result must be divided into segments which are exchanged between main and auxiliary storage as the computation proceeds. A new segment is required whenever an instruction or item of data in a new segment is referenced from the one currently executing. The problem is to divide the program into (not necessarily disjoint) segments in such a way as to minimize the total cost of swapping during execution, using a stochastic model of the program references to its various parts. This situation is not uncommon with programs requiring large matrices or other data structures, e.g. [54]. Since the system cost in this case depends on only one task, it can be expressed as the objective function of a well-defined discrete optimization problem.

The use of a Markov chain for a model of program references was first considered by Karp [39] as an extension of his work with directed graph models. Graph models of computations are also considered by Ramamoorthy [56] and others [48,55,60,64]. The transitive graph probability model of Estrin, Turn and Martin [22,23] is a more comprehensive technique which requires a great deal of numerical evaluation, but includes a model of parallel computations. A nice solution to a special case of the sequential transitive model (using magnetic tape as auxiliary storage) was given by Wonderly [72]. A different way of computing parameters from Markov models has been suggested by Ramamoorthy [57]. Chapter I of this thesis extends the formulations of Karp and Ramamoorthy by providing a simple and general cost function for the optimization problem. Like the transitive model, the only fast solutions to this problem are suboptimal, for the effort required to obtain the full solution is probably not justifiable in practice. The formulation is nevertheless useful in that the cost function provides a way of comparing any proposed set of alternative solutions.

In the graph and stochastic models of programs, a section of data and/or instructions is taken to correspond to a node or state. Parameters obtained from the model then show how these states, and hence the parts of the program, should be grouped together. In certain partitioned multiprogrammed systems where each task is restricted to a fixed-size share of main memory, these models remain relevant; for optimal system performance is attained when each task performs well in its own partition, independent of the other tasks.

Bringing the Markov model to bear on virtual storage systems, we encounter a multilevel problem:

- first, the _a priori_ packaging of data and instructions into one-page units so as to decrease the number of references to other pages, given a model of these references,

- second, the best dynamic allocation of pages of main storage to a single task, given a model of its inter-page references, and finally, the control of storage allocation events by scheduling tasks so that CPU usage is maximized.

The first problem is judged to be poorly served by these models, for it is the single-thread problem above in microcosm, with a more complicated cost function determined by a solution of the other problem levels. Indications are that heuristic techniques such as that reported by Comeau [13], and simple procedures built into compilers and other system components will be competitive at this level. A discussion of the possibilities is given at the end of Chapter I.

For the second problem, the feasibility of using Markov chain models of inter-page references is a controversial issue. Shemer and Shippey [66] use them to estimate the optimal size of the associative store in virtual addressing hardware. Denning [17,18] feels that the accuracy of Markov chain models is questionable and that reliable data for them is too hard to obtain. The current debate stems in part from the following considerations:

- Simulation studies[1], early experimental results[2], and actual experience with virtual storage systems show that they are not operating as well as had been hoped,

- thus the original concept that the new addressing structure and hardware assist could operate efficiently without altering the structure of tasks seems to be optimistic, and

- a Markov model of programs provides a tempting representation because the data is obtainable by prior examination [15,11] or dynamically during execution [57]. It provides an adjustable level of detail and is capable of analytic solution.

Chapter II discusses the application of Markov models to the second problem level. Starting with a demand paging formulation of Lauer [45], a cost function is considered which illuminates the difficulties and possible advantages of using _a priori_ information to improve paging for an

----

[1]For example, see Nielsen [52].

[2]Very little data has been published, but see Fine, _et al._ [28], Coffman and Varian [11], and Comeau [13].

individual program. However, it is an enlightening investigation rather than a solution to the problem.

Chapters III and IV consider the third aspect of the allocation problem in a virtual storage environment: scheduling tasks to achieve overall system balance and a busy CPU. Most studies of this subject in the literature focus on the queueing structures used in scheduling and the response that these techniques provide for individual tasks [8,16,41,42,46,65]. Coffman [10] developed a Markov process as a model for the way in which storage fills and empties with virtual memory pages. Scherr [62] used a similar continuous-time process to represent the number of tasks ready for execution in the simpler CTSS system [59]. Fife [27] and Smith [67] considered Markov decision processes with rewards [33], and investigated optimal control (i.e. scheduling) policies. The technique of Chapter III, which in spirit most closely resembles that of Gaver [30], exploits the form of a particular paging delay distribution (see Appendix B). Assuming a fixed number of tasks, the CPU busy period is obtained with a cyclic queue model, where CPU service is assumed to have a negative exponential distribution. In comparison, Gaver assumed a negative exponential I/O (or CPU not-ready) delay and considered more general distributions of CPU service time, using a recursive method to derive the CPU busy period, again with a fixed number of tasks. The result of Chapter III by no means provides the definitive solution to this complicated problem; but it provides a way of studying the system which allows the delays due to paging to be represented rather accurately for a drum system. It may be used in combination with other models to obtain a better total system model.

IV. EXPERIMENTAL WORK

Chapter IV reports the results of a digital simulation study of task scheduling and storage allocation algorithms for a virtual storage machine. In simulation and theoretical models, there is a pressing need for accurate information about the way tasks behave with respect to CPU and storage requirements in the new systems. This data is required in order to decide what types of models are appropriate, how they will differ from reality, and what parameter values to use for numerical calculations. To the author's knowledge no published material has appeared to date from an actual system operating on a paged, virtual storage computer. Information is only available in the literature from a number of systems in which paging is simulated with hardware techniques [63] or the interpretation of instructions [11, 28]. The author has also found that when conceptually simple scheduling and allocation techniques are actually coded for a specific system, their operation can be significantly different from that which was expected (see, for example, Appendix B, Section V).

In order to give the simulation and mathematical models in this study a solid foundation, a data collection facility was put into the MTS time-sharing system on the IBM System/360 Model 67 at the University of Michigan Computing Center. This operating system, which is described briefly in Appendix A and more completely in [51], was developed at the University as an interim system pending satisfactory performance of the IBM Time-Sharing System [31,35,47]. MTS, which provides a simpler virtual storage implementation than that of which the 360/67 is capable, is a highly successful system. The data collection facility (described in Appendix C) has proved to be a very useful tool for the MTS system programmers, as well as providing the data given in Appendix D. In addition to its help in locating system errors, it is being used to

- measure operating system overhead

- sample the CPU idle time distribution

- monitor its own interference with other tasks

- collect "repeatable" data for experiments with billing algorithms

- find a specific hardware problem (in the high-resolution timer) and verify that a software modification corrected it

- provide an alternative to manufacturer's estimates of hardware performance data

- help identify inefficient sections in the coding of a large compiler

- test a new data structuring scheme for the GPSS/360 simulator

- discover the extent to which sharing routines in MTS would be useful, by determining the number of times that copies of a system routine are being used concurrently simulate the system with different loads by running it with artificially constructed programs

Programs written by the author to analyze the MTS data required a detailed study of the structure of the supervisor. This resulted in considerable insight into the differences between the actual system and various models which had been proposed. In the sense that the results fill a serious gap in the published data about virtual memory systems, and are being directly used to improve the system concerned, the data collection facility may prove to be a very significant contribution of this research. In any case, it is shown to be an important tool in the study of these systems.

Digital simulation models are being used with increasing frequency in studying computing system behavior [2,29,52,61, 62]. When arrival rates for tasks to parts of the system are a function of the service rates in others, when service rates have atypical distribution functions, and when multi-level preemptive priority scheduling schemes are employed, these systems defy mathematical analysis. Even when a theoretical model of part of the system has been constructed, it is desirable to have another model at a greater level of detail or using more realistic distributions, as a check on the assumptions made to derive the simpler one.

The simulation studies in the literature which relate most closely to the present work are Scherr's simulation of CTSS [62] and Nielsen's model of the IBM TSS [52]. In spirit, the present model is like that of Scherr: data from a real system is used in a model which is much less complex than the real system, and it is used to explore alternatives for improving the system. However, the CTSS system is fundamentally different and simpler than the current virtual storage systems, and consequently the present model actually bears little resemblance to Scherr's. Nielsen [52] has produced a very detailed model of a specific combination of machine and software systems. The machine used there is the IBM 360/67, from which the MTS data was taken. However, his model was constructed at least in part to evaluate the specific IBM TSS operating system in its then unreleased, unrefined, and hurriedly constructed state of development. In contrast, the model in this paper is an order of magnitude simpler (the chief difference being that no record is kept of the identity of individual pages) and is used to investigate and compare supervisor techniques that are currently thought to be extremely effective, rather than those which were actually coded at an early date. Finally, the data describing tasks for the model is taken from MTS with the data collection facility, reduced to the level of detail of the model, processed to remove some of the effects of the MTS system itself, and then inserted directly into the simulation model.

V. SUMMARY

This study covers a number of techniques for attacking a single, basic problem. Chapter I provides a way of using stationary program reference probabilities to select the best of several proposals for segmenting a program. Chapter II extends the discussion of Markov models to paging systems, and shows how one can decide, in advance, whether the Markov model for that program results in improved storage allocation. It is shown how decisions can be made during execution to choose dynamically between several standard allocation procedures and a predictive one. Reference probabilities can be obtained by using a data collection facility such as the one constructed for the MTS system. Chapter III gives a model of the set of tasks concurrently sharing the CPU and a paging mechanism which yields parameters for a dynamic supervisor control policy of scheduling and storage allocation. Chapter IV reports the results of a simulation project to compare a number of simpler controls for storage allocation. Detailed descriptions are given in the appendices of the actual system used for experimentation, the facility used to monitor its activity, the data obtained with that facility, and the simulation model.

CHAPTER 1. STATIC ALLOCATION

"It is not at all clear that a Markov model is useful for modelling program behavior, for too many correlations are involved, arising principally from loop behavior of programs."

P. J. Denning

"Each venture
Is a new beginning, a raid on the inarticulate
With shabby equipment always deteriorating."

. S. Eliot

This chapter considers the use of a Markov chain model to aid in the selection of program segments. The implications of the Markov assumption are discussed, and the model is used to formulate an optimization problem for segment selection.

I. THE PROBLEM

Overlay segmentation is used on single-thread and multiprogrammed computing systems, both large and small, as a means of increasing the effective size of main memory without the use of special relocation hardware. Parts of a program are loaded from secondary storage as the execution proceeds, and others are discarded or returned to secondary storage.

In this chapter we will think of programs as being divided into a number of small <u>blocks</u> of data and/or instructions which could be chosen in a variety of ways. These blocks are to be grouped into sets we call <u>segments</u>. The problem is to group the blocks (divide the program) into segments so that system overhead is minimized. Once segments have been specified, the actual overlay process uses well-known techniques summarized, for example, by Pankhurst [53]. Facilities exist in programming languages [e.g. the

FORTRAN INCLUDE statement and COBOL segments] to allow the programmer to explicitly form or partially control the formation of segments. If the choice is left entirely to the system, segments are usually chosen to match some physical characteristic of secondary storage.

II. MODELS

The essential information required for segmenting is the set of program references: those points in a program where a branch occurs to an instruction at another point or data stored elsewhere is needed. A useful model which abstracts this information is a directed graph, each node of which represents a small block of data or instructions. The arcs of the graph represent the references between blocks: an arc may be used to denote a branch from one instruction to another, and a reference to data may be denoted by a two-way arc, to indicate that the information in the referenced block is required; yet the control remains with the referencing block. It is also useful to label each node with an estimate of the size of the corresponding block (amount of storage it requires).

The graphic model provides a convenient way of working with program references: algebraic operations with the connection matrix yield a number of useful descriptors of program behavior. The models of Marimont [48], Prossner [55], Krider [44], Karp [39,40], Schurmann [64], Salwicki [60], and Ramamoorthy [56,57] use these calculations to look for programming errors, rearrange the parts in storage, and count the number of loops in which a given node is contained. As an example of the use of this technique, one can identify the n maximal strongly connected components of the graph. If the corresponding parts of the program could be chosen as segments, then whenever the program was executed, at most n-1 overlays would have to be performed. The property of strong connectedness has little to do with program size, however, and a single instruction or an entire program may happen to be such a component. Segments must be small enough to meet constraints, yet large enough to eliminate the overhead in dealing with them. Thus we need a model which contains more information.

To obtain a more realistic model, we observe that both data references and branches in a program are conditional. For example, if the command

IF A = 5 OR B > 6, GO TO C

is mapped into machine instructions from left to right, a reference to the datum B is necessary only if $A \neq 5$, and a branch to C occurs only if at least one of the relations is true. By associating with each reference a probability that it will be required, we obtain a stochastic model of the program. If, in the previous example, the distributions of values of A and B are independent, and

$$\Pr[A = 5] = .3, \quad \Pr[B > 6] = .2$$

then

$$\Pr[B \text{ is referenced}] = 1 - .3 = .7$$

and

$$\Pr[C \text{ is referenced}] = .3 + .2 - .06 = .44.$$

In particular, if we assume that the probability of making a reference is always independent of the number of blocks of instructions so far executed, we have a (stationary) Markov process, or Markov chain. A state of the Markov chain corresponds to a node of the graph model, and the arcs (or rather entries in the connection matrix) are replaced by transition probabilities.

It is evident that computer programs need not have the Markov property: the likelihood of a reference is not independent in general of the history of execution. However, we may still find a Markov model useful for several reasons. First, we are interested in the expected behavior of the program over its total execution rather than at any given instant. Second, a number of individual references for which probability estimates have been obtained may be combined into a single block, or state of the model, tending to smooth the time-dependence errors. In general, the detail of this kind of model is adjustable to a level only as fine as necessary for the goal of the study. And finally, the results will be used to make qualitative rather quantitative decisions, so that a consistent bias may have less effect.

There are many ways to determine probability estimates for references in a computer program. Perhaps the most general way is to begin with probability distributions for appropriate ranges of values of input data, observe how the data is modified by the execution of the program, and obtain derived distributions for data which determine conditional references. Much simpler methods may be adequate and will have to suffice in practice, however. The data collection facility for MTS, described in Appendix C, has been used to obtain such estimates for a large program. The simulator described by Coffman and Varian [11] constructs a transition matrix as part of its output.

A typical computer program has one or more entry and exit points. We will assume that the entry point is chosen at random from a probability distribution, and that an exit to some "higher level" routine always occurs eventually. Thus each exit corresponds to an absorbing state of the Markov chain, and each block of instructions or data gives rise to a transient state (is used at most a finite number of times).

III. NOTATION

Let the set of transient states of the Markov chain be denoted by $\{1,2,\ldots,n\}$. The analog of the connection matrix of the graph is the $n \times n$ transition matrix $Q = [Q(i,j)]$, where $Q(i,j)$ is the probability that a transition occurs from state i to state j. If we let $V = \{v(1),v(2),\ldots,v(n)\}$ be the probability distribution for the choice of an initial state, then the i,j entry of the k-th power of Q is the probability that j is the k-th state to be reached in the process, given that it started in state i. Since Q contains only the transition probabilities for the transient states, the powers of Q tend to the zero matrix, and so

$$\sum_{k=0}^{\infty} Q^{k} = (I-Q)^{-1} = N,$$

where

$$Q^{0} = I$$

is the $n \times n$ identity matrix. If the process started in state i, then the i-th row of the fundamental matrix N is the vector of expected frequencies of the states: $N(i,j)$ is the mean number of times a reference to j occurs in the process.

The matrix N provides a way of measuring the relative frequency with which blocks (and references between them) occur in the process. It also gives the expected length of the process in terms of number of references. If the entry point to the program is known, only one row of N is needed.

Karp [39], and Ramamoorthy [56] considered an alternative form of the model which yields some of the same information: if one thinks of restarting the program at every exit by simply repeating the entry point experiment, then the exit states are included in the model, the process never terminates, and we have a regular Markov chain. The analog of the fundamental matrix for regular chains is a limiting state probability distribution independent of the starting state, which can be computed by solving the matrix equation

$$M = M\,Q',$$

where M is the unknown vector and Q' is the m x m transition matrix of the regular chain. Here we have

$$\sum_{i=1}^{m} M(i) = 1.$$

The absorbing chain model is related to the regular chain model in the following way: each absorbing state of the former (which was not represented in the matrix Q) appears in the latter as a state whose transition probabilities repeat the experiment for the choice of an intial state. These additional states appear in Q', and the probability of starting in one of them is zero. Let the new states be given indices n+1, n+2,...,m. Then the first n entries in the solution vector M are related to the last m-n and and the fundamental matrix N of the absorbing chain in the following way:

$$M(i) = D \sum_{k=1}^{n} \{v(k)N(k,i) / \sum_{j=1}^{n} N(k,j)\},$$

$$\text{where } D = 1 - \sum_{n+1}^{m} M(i).$$

The regular chain stationary distribution can be computed by recursive techniques [58] which are especially designed for this type of equation. The generating function approach can also be used [56]. Thus even though the regular chain model has more states than the absorbing model, the calculation of the solution is competitive with it. If the starting state is known, that fact can be used in the solution of the absorbing model: then it is only necessary to solve a matrix equation instead of obtaining an entire inverse matrix. We also note that the absorbing model contains more useful information about the program, for the N(i,j) are actual expected frequencies rather than relative weights. Thus we can determine an estimate for the mean length of the process. This would be useful, for example, to compare the speed with which two different algorithms are able to solve the same problem. In what follows we will work with the absorbing model and the state frequency vector M given by

$$M(i) = \sum_{k=1}^{n} v(k)\ N(k,i).$$

The size of each block of the program (storage required) is of concern here, hence we let s(j) denote the size of block j, giving a size vector S for the program. Though this fact will not be used in the sequel, we note that if t(j) is the average execution time for block j, then

$$\sum_{i=1}^{n} v(i)\ N(i,j)\ t(j)$$

is the total expected amount of execution time used by the j-th block during execution of the program. This is in fact a <u>semi-Markov</u> model of the program execution.

Segmenting a program corresponds to identifying a <u>cover</u>, or collection of subsets of the states of the model, the union of which is the entire set. If no state can appear in more than one subset, the cover is a <u>partition</u>. Let U = [u(1),...,u(h)] denote the sets of the cover U. The following zero-one matrices are useful for characterizing U:

P where P(i,j) = 1 if and only if state i is a member of set u(j).

L where L(i,j) = 1 if and only if for some index k, i and j are members of u(k).

P represents the sets u(j) directly. If U is a partition, then P can have at most n non-zero columns.

Hence it is convenient to allow empty sets and fix P as an $n \times n$ matrix. If U is a partition, then every row of P is a unit vector. The matrix L is a better representation of U from which to discover if a pair of states lie in the same set. L is symmetric with a unit diagonal. If U is a partition, L can also be given by

$$L(i,j) = \sum_{k=1}^{n} P(i,k)\ P(j,k).$$

We say that a cover U is <u>admissible</u> only if the subset of program blocks corresponding to every set in U satisfies a segment size limitation (the sum of the sizes of the blocks is not greater than a constant), expressed by

$$\sum_{\{u(k)\}} s(i) \le r(k),$$

where $R = [r(1),\ldots,r(n)]$, so that $r(k)$ constrains the size of segment k. We conclude our definitions with the assumption that there is a cost $C(i,j,U)$ associated with the transition from block i to block j whenever these two blocks are not in the same segment. $C(i,j,U)$ represents the time the CPU is idle during the exchange of segments.

IV. FORMULATION OF THE OPTIMIZATION PROBLEM

The Markov model yields the expected frequency $M(i)$ for block i during the execution of the entire program. After each execution of i there occurs a transition to j with probability $Q(i,j)$ and cost $C(i,j,U)$, where i and j are indices and U is present to indicate that the transition cost may depend on the sets of the partition.Thus the expected cost of this transition for the entire execution is

$$M(i)\ Q(i,j)\ C(i,j,U).$$

Let $u(k)$ be a set of U in which state i resides. Then the total expected cost of an exit from $u(k)$—or of all overlays which originate from the corresponding segment—is

$$\sum_{i=1}^{n} P(i,k) \sum_{j=1}^{n} M(i)\ Q(i,j)\ C(i,j,U)\ (1-P(j,k)),$$

where the multiplier $P(i,k)$ "counts" just the states in the set $u(k)$, and the factor $(1-P(j,k))$ eliminates the terms for which there is a copy of state j in $u(k)$ as well. The total cost of all intersegment transitions (the cost of this segmentation of the program) is the above expression summed over all the sets of U. It is the latter which must be minimized over all admissible covers, that is, over all Boolean matrices P satisfying the relation

$$S\,P \leq R,$$

where S is the size vector for the program blocks, and R is the size constraint vector for the segments. Generally, all segments are constrained to the same fixed size, and thus R can be written as a constant times the vector of all ones.

At this point, we note that the use of a segmentation with multiple copies of certain blocks (a cover which is not a partition) is not a good model if the original probabilities $Q(i,j)$ were obtained when only a single copy of the block was used. The appropriate model requires that the transition probabilities $Q(i,j)$ be replaced by sets $Q(i,j(1))$, $Q(i,j(2))$,... giving probabilities of transfer to each distinct copy of the state j. This in turn requires more information about the original program, and in essence dictates that the original transition matrix be constructed with each copy represented as a separate state. Hence it is sufficient to analyze the case for a partition, although in practice it is harder to obtain the appropriate partition when blocks of the program are to be shared between segments. Another simplification can be made by observing that in general the real cost of a transition from one segment to another, which requires time for the access and transmission of data from auxiliary storage, depends more on access time than on transmission time. Thus if the auxiliary storage medium is direct access in nature, the cost of a transition is relatively independent of the identity of the segment involved. At this level of detail, at any rate, the costs can reasonably be taken equal:

$$C(i,j,U) = C(i,j) = 1.$$

With the above assumptions, the objective function for the entire segmenting cost reduces to a difference of two sums, the left-hand one of which is constant with respect to the matrix P and the latter of which is

$$\sum_{i=1}^{n} M(i) \sum_{j=1}^{n} Q(i,j) \sum_{k=1}^{n} P(i,k)\ P(j,k).$$

If we now apply the definition of the matrix L, we obtain

$$F = \sum_{i=1}^{n} M(i) \sum_{j=1}^{n} Q(i,j)\ L(i,j).$$

The problem has thus been reduced to the form

$$\max_{L} F \quad \text{subject to} \quad S\ P \le R, \quad \text{and } P \text{ stochastic}$$

where P is a Boolean square matrix of order n in which each row is a unit vector, and L is easily obtained from P. The constraint that P be zero-one is a non-linear one, and the unknowns are the n^2 entries in L (or equivalently, P).

V. STABILITY

It is of interest to consider the stability of the cost function with respect to small changes in the transition matrix Q: if the chosen Q differs somewhat from the "best" Markov model of the program, how does the obtained value of the cost differ from that of the "best" model?

First, we note that Q may be any one of a large class of matrices. An exact method of determining the transition probabilities Q(i,j) has not been specified, but with almost any reasonable scheme,

> given any square, non-negative, sub-stochastic matrix Q with rational entries, there is a program with Q as its Markov transition matrix.

Using the definition of the weighting vector M = V N and the fact that L is symmetric, we can write the cost function as the sum of the entries in the vector

$$V\ (I-Q)^{-1}\ Q\ L,$$

where the initial state probability vector V forms a convex combination of entries in each column of the fundamental matrix N $(I-Q)^{-1}$, and the matrix L selects subsets of the entries in the convex combination. Neither V nor L is affected by changes in the matrix Q, and hence the stability of the cost function rests on the product

$$(I-Q)^{-1}\ Q.$$

More precisely, if we replace Q by a matrix $Q+E$, how do the entries in the foregoing product compare with those of

$$(I-Q-E)^{-1}\ (Q+E)$$

if we assume that $Q+E$ is also non-negative and sub-stochastic. We will see that the stability of the cost function depends not so much on E as on Q itself.

The eigenvalues of $(I-Q)^{-1}$ are of course the reciprocals of the eigenvalues of $I-Q$. If h is an eigenvalue of Q, then because Q is non-negative, $1-h$ is a nonzero eigenvalue of $I-Q$. Hence $1/(1-h)$ is an eigenvalue of $(I-Q)^{-1}$. Since the inverse of $I-Q$ exists, the eigenvalues of Q are all less than one. From the Perron-Frobenius theory [71] we get the following useful results:

- there is an eigenvalue of Q equal to the spectral radius $sr\{Q\}$,
- if E is non-negative and not zero, then $sr\{Q+E\} \geq sr\{Q\}$,
- if Q is irreducible, then $sr\{Q+E\}$ is strictly greater than $sr\{Q\}$.

Thus increasing the entries in Q tends to increase the spectral radius, and the more so as Q is closer to being an irreducible matrix. These matrix properties can be related to program models in the following way: increasing the entries in Q (making Q closer to a stochastic matrix) corresponds to modelling a program which runs for a longer time. Hence at least some of the entries in $(I-Q)^{-1}$, which counts the mean number of times in each state, are larger. As Q becomes more irreducible, a modelled program tends to have more loops and transfers which make it possible to reach more parts of the program from others. The effect on $(I-Q)^{-1}$ is to make the entries more comparable in magnitude (homogeneous). (In practice, however, a perturbation of Q would probably not change the location or number of the zero elements.)

In summary, the most important factor affecting the stability of $(I-Q)^{-1}$, and hence the stability of the product in the cost function, is the spectral radius of Q. For $sr\{(I-Q)^{-1}\} = (1-sr\{Q\})^{-1}$, and the latter increases rapidly as $sr\{Q\}$ approaches unity. Generally speaking, the closer $sr\{Q\}$ is to one, the more mathematical instability (sensitivity of the inverse to perturbations) and computational instability (errors in computing the inverse) are present. Two observations can thus be made:

(a) The cost function is not stable or unstable *per se*, but rather instability depends on the choice of Q, and hence the program model. It is only a problem if the spectral radius of Q is close to one.

(b) If $sr\{Q\}$ is near unity, the entries in $(I-Q)^{-1}$ tend to be rather large, and hence a perturbation of Q has less effect on the relative error than it has on the absolute error.

VI. SOLUTION PROCEDURES

The optimization problem stated in Section VI is of a particularly difficult general type to solve. It is non-linear (discrete), has many variables, and the property that other information about the corresponding actual situation does not readily contribute to a special iterative procedure for finding a global optimum. There seems to be no *a priori* way in which the optimal value of the objective function can be determined or reasonably bounded. Although suboptimal algorithms can be invented, such as directed search techniques based on the properties of the corresponding graph model, their utility is doubtful because there is no way to judge the values so obtained.

After a number of other unsuccessful attempts, the author has concluded that the most satisfactory way to find the optimum is probably a general technique for Boolean variables, such as that given by Ivanescu and Rudeanu [36,37]. This general method makes no use of the special kind of problem we have, but the fact that the variables are all Boolean allows some economy of method: the objective function can always be written as a polynomial with integer coefficients, and the fact that the constraints are non-negative in this case means that they can be incorporated into the objective function, leaving an unconstrained problem. From that point on the "pseudo-Boolean" procedure referenced above is an essentially enumerative one, although additional shortcuts might be taken if this particular application were assumed.

The solution procedure just considered is rather time-consuming and inelegant, especially as it comes at the end of a long path of assumptions about the structure of a program and the availability of accurate information describing it. From a purely practical point of view, it is not worth it to carry out this optimization procedure. However, the objective function itself is still useful, in that it provides a way of selecting the best segmentations (according to the model) from any given set of proposals. If the solution were more highly constrained in some particular case, then a simple enumeration of the alternatives which satisfied the constraints might be possible. Suppose that the matrices L^0 and L^1 give two different proposed segmentations which obey the segment constraints. In the notation of the previous section, the cost difference is given by the sum of the entries in the vector

$$CF = M\ Q\ (L^1 - L^0)$$

and the sign of the difference determines the better one.

VII. SUMMARY

In this chapter we have formulated a cost characterization to improve segmentation of computer programs. It is appropriate at this point to consider under what circumstances the results might be used. First, despite a history of proposals to that effect (e.g. Ramamoorthy [57]), it will not be feasible to put the necessary algorithms into language processors to gather and use stochastic information about the way the program is expected to behave during execution. It is also unlikely that much success can come of gathering and using such information dynamically during the execution of ordinary programs. The same amount of effort spent on heuristic techniques would probably achieve a better average improvement.

There is one specific application, however, for which this model retains some merit: large system programs. One index of the power of an operating system is the extent to which a user can rely on the facilities of the system and avoid writing his own programs. The trend thus seems to be toward operating systems with many heavily used subsystems, such as language processors, which tend to be composed of many subroutines, and are combined at execution time according to a rather constant pattern. Aside from the fact that data and instructions could be separated in order that the instructions may be shared, the subroutines themselves are often rather small.

Data usually consists of a number of scalars, fixed-size arrays, and perhaps dynamically changing data structures. This natural structure can be exploited in choosing blocks for the Markov model as the entire set of scalars, each fixed-size array, each subroutine, and perhaps a number of blocks for each growing array.

System programs not only enjoy heavy use, but they are available for study, and are normally subject to regular growth and maintenance. The time spent to collect the data for a Markov model and carry out the analysis can be justified if even a small average improvement is realized in performance. As a purely practical consideration, it is often possible in this case to enlist the aid of the individuals who wrote the programs to observe and understand their execution, temporarily add instructions to trace flow, etc. To take a concrete example, the MTS data collection facility has been used to trace the subroutine calls in a new compiler, and show the amount of time spent in each subroutine before the call to another. A semi-Markov transition matrix can easily be constructed from such data.

VIII. APPLICATION TO MULTIPROGRAMMING

When more than one program can reside in main storage at once, the cost structure is usually more complicated than the one outlined in this chapter. For the simplest multi-program storage allocation scheme, however, it remains the same. The so-called _partitioned_ system simply divides main storage into a fixed number of fixed-size units, and one task at a time is allowed to execute in each such partition. The CPU is allocated to a new task, if possible, whenever the currently executing task pauses for I/O or a new program segment, or terminates (to be replaced by another). Any use of multiple segments by a single task is restricted to the given partition, as if main storage were only that large, and regardless of the fact that parts of other partitions may have unused space. Since there is no interaction between tasks except in the use of the CPU, the length of the CPU queue is maximized (and hence the cost of swapping segments for the whole system is minimized) when each task is organized as if it were to be run optimally in a single-thread system with a main storage the size of its partition.

If the multiprogramming system can allocate a variable amount of storage space to tasks, allow their allocation to change dynamically, and keep a variable number of tasks in main storage, then the cost structure becomes much more complex. There are two primary reasons why this is so: (a) at the very least, a storage use time integral is required to express the cost to the system of a task's use of storage; (b) storage cost is not necessarily a linear function of size, for it in fact depends on how many other tasks can be readied for execution, and hence on system decisions. The possibility of queueing for storage is also introduced whenever a task switches segments.

A model of an individual task cannot hope to contribute to multitask queueing problems, but the Markov model can be reformulated for optimization with the cost based on a storage use time integral. The following modifications are necessary to the formulas of Section IV:

- a semi-Markov chain is required—allowing a distribution for the time spent in a given state,

- the size constraint can be removed or at least replaced by a less restrictive one,

- a second term is added to the cost function, consisting of an increment to the space-time integral at the intersegment transition. It is of the form s(i) t(i,j), where s(i) is the size of segment i and t(i,j) is the time spent in i before a transition to j.

In practice, then, the additional data consisting of block executon times is required for the model. A segment execution time can be obtained from the block execution times by the following procedure: make all states of the model except those in the segment absorbing states, and compute the mean time to absorption (until the segment is exited). The additional computation required to obtain these times, however, effectively eliminates the possibility of using a general optimization procedure, since they must be computed for each potential segment in each possible segmentation. Comparison of a handful of given segmentations remains a possibility, though. The next chapter considers another way to use the same kind of information.

CHAPTER 2. DYNAMIC ALLOCATION FOR A SINGLE PROGRAM

"Modelling paged and segmented memories is tricky business."
P. J. Denning

This chapter examines the behavior of a single task in a virtual storage system and shows how stochastic models can be applied to dynamic storage allocation decisions. Cost functions are formulated which allow a comparison between predictive methods and conventional techniques, as well as showing the different ways in which the costs occur.

I. TERMINOLOGY

On a large-scale, time-shared computer where a number of individuals dynamically interact with the system from remote terminals, the concept of "program" or "task" in the sense of the batch system is no longer useful. A person may require a number of programs at any given time, which are dynamically linked together at his request. Some of them may be shared with other people. One may initiate an independent "subtask" which is processed concurrently with the primary computation. Definitions more appropriate to these systems are proposed by Dennis and Van Horn [20]. For our purposes, however, it is sufficient to consider as a task each distinct request for service from the system. In terms of a remote terminal user, this is a single interaction: the computation initiated with his command and terminated when the system has obeyed the command and returns to him to accept another.

II. PAGE-TURNING ALGORITHMS

The virtual storage concept is described in the Introduction as a method of storage allocation which makes the use of secondary storage for real memory overflow transparent to the individual programmer. Although some of the functions of the operating system are also simplified with this approach, there tends to be a higher level of basic operating overhead—some in the hardware itself—which one hopes can be made up by more efficient operation than would ordinarily be possible under periods of heavy load. However, the system is also required to make more and difficult decisions about scheduling and storage allocation. Because storage allocation is more flexible, information about the behavior of individual tasks can be used to better advantage.

The real cost to a multiprogramming system of executing a task is in part determined by the amount of interference it causes in the attempt to keep at least one task elgible to use the CPU. However, as we stated in the previous chapter, measuring that disruption requires a model encompassing all tasks in the system, as well as the scheduling and storage allocation algorithms. Since the frequency of storage allocation events depends largely on these algorithms and the other tasks, it is reasonable to consider only the storage usage pattern of the given task when using a single-task model. To this end we deal with space-time integrals as measures of cost in this chapter: the product of real storage space and the amount of time that space is used.

Time-sharing systems in use before the inception of virtual storage usually employed a swapping algorithm (e.g. see [59]) for placing an entire task in real storage as it was about to use the CPU. No additional real storage was needed during execution, hence the only ways a task voluntarily ceased executing were to wait for I/O or terminate completely. There were several disadvantages to this technique:

- The overhead of swapping was great enough so that during waits for I/O, except from the very slow remote terminal itself, no other task could use the CPU (unless there were room in real storage for more than one complete task).

- To meet response requirements, each task was swapped out after a certain maximum amount of execution time (<u>time slice</u>) if it had not completed by then. During a time slice, however, part of the swapped-in task was never referenced, so that in the long run much time was spent moving unused parts of a task in and out of main storage.

With more recent virtual storage systems the basic memory allocation algorithm operates in units of one page, antithetic of swapping: a page is brought into real memory only when it has actually been referenced. Coupled with this _demand paging_ rule is a replacement algorithm, which specifies a page to be pushed out of real storage in order to make room for the one coming in. Although demand paging is usually used as the input rule, there are a number of replacement rules [5,11,17,21] which may be selected for the overall _page-turning_ algorithm. The chief disadvantage of demand paging is that a task requesting another page unproductively occupies a number of real storage pages during the time necessary to fetch the new page.

III. STORAGE USE COST FUNCTIONS

When we examine the costs of various storage allocation schemes, it is necessary to keep in mind the distinction between two time scales: that of "real time," in which the system operates and system costs are assessed, and "task time," or the execution time of an individual task. A basic assumption of this paper is that for all the storage allocation algorithms under consideration, the same total amount of task time is required by a task. That is, a task must execute for a duration independent of the places and number of times it is interrupted, and the duration of real time of those interruptions.

In the discussion that follows we will focus on a time span shorter than that required to completely process a task. Since the number of requests for service which are currently in various states of completion at any moment can be very large, a subset of these is usually selected for concurrent execution over a short interval of time. Only the selected tasks are allowed to acquire more main storage and compete for the use of the CPU. When a task must wait for I/O or another virtual memory page, it may or may not be allowed to remain in that selected set. If the task has not completed after using a certain amount of CPU time (a time slice) it is removed from the set. The number of selected tasks may vary from time to time, and its membership changes continually.

In the rest of this chapter we are interested in the interval of real time over which a task remains selected (completes a time slice). We will simplify the discussion by assuming that each time a task ends a CPU interval, it does so by page fault, and that it resumes execution as soon as the page has arrived in main memory. Once referenced, a page remains in real storage until the end of a time slice. The exact reasons for making these particular assumptions will be discussed in a later section.

We assign as the cost to the system of a single time slice the space-time integral

$$\int_0^T P(t)\,dt$$

where the task begins the time slice at real time 0, terminates at real time T, and at time t has acquired P(t) pages of main storage (has referenced P(t) pages of his programs). The function P(t) is of course a step function, and the interval [0,T] contains intervals when the task is not ready to use the CPU as well as intervals of task time.

Let us now consider the situation at a given point t in the interval [0,T]. To simplify the notation, let P(t) = j. For the moment we assume that several characteristics of the future behavior of the task are known:

(a) the next page request will occur at time t+s.

(b) the time interval from that request to the subsequent arrival of the page in main storage is of length w.

(c) the identity of the next page to be referenced is known.

Then the cost added to the space-time integral because of this page request can be described as follows:

- If no action is taken until the page demand occurs at t+s, a delay of w will ensue during which j+1 real pages are occupied by the task (space for the new page must be allocated before transfer begins) at a cost of w(j+1).

- If the page were requested in advance, timed to arrive just as it was required at t+s, the additional cost would be for only one page over the time interval [t+s-w, t+s], or w.

- If a wrong page were selected and advanced in the manner just described, the page fault would occur anyway at $t+s$, and hence both the above costs are incurred: $w(j+1) + w$. (If the space occupied by the wrong page is not immediately released there would be a further cost.)

In practice, of course, exact values of s, w, and the identity of the next page are unknown. For any particular system, the paging delay w can be approximated by a random variable with a given distribution function H. We will use the function developed in Appendix B for a specific drum I/O channel as an illustration later in this chapter. We now assume that similar information is given about the task: the interval s between page requests is a random variable with a kncwn distribution function D, and a probability distribution is given for the selection of a specific next page.

In his investigation of control policies for allocating storage, Smith [67] finds that the optimal policy is often non-stationary. The exact advantage of a dynamic policy over a stationary one in this context is that it allows the distributions for w, s, and the next page to be conditional. Here, the delay w depends on the number n of page requests currently being serviced, the interval s is a function of the number j of pages already acquired, and the choice of the next page can be Markovian: conditional on the identity of the page most recently requested. If a decision is made dynamically, the current values of the parameters n and j are assumed to be known, and although the probabilities for referencing each new page are conditional on the last reference, a fixed policy can be used for the choice of the next page (the one most likely to be referenced next). Thus in the ensuing discussion on dynamic policies we treat the parameters n, j, and q as if they were unconditional, where q is the probability of referencing the chosen next page.

We now turn to the question of minimizing the expected cost of obtaining a single next page, given the options discussed above and the cumulative distribution functions $H(w)$ and $D(s)$. At time t we wish to make a decision about when to initiate the next page request. For advancing a page at time $t+z$ we have a mean cost of

$$q[(s-z)+r(j+1)(w-(s-z))] + (1-q)(s-z+(j+1)w)$$

where r is the variable

$$r = 0 \quad \text{if} \quad w < s-z, \quad \text{and}$$
$$= 1 \qquad \text{otherwise.}$$

Taking expectations with respect to the random variables w and s, we obtain the average cost

$$EWS(z) = \underline{s}-z + (1-q)(j+1)\underline{w} + q(j+1)CN(w,s,z)$$

where $\underline{s}$ and $\underline{w}$ denote the mean values of s and w, and the function CN in the last term is given by the convolution of D and H:

$$CN(w,s,z) = \int_0^{\infty} \int_0^{z+w} (w-(s-z))\, dD(s)\, dH(w)$$

To obtain the value z^o of z which minimizes the expected cost $EWS(z)$, we set

$$d[EWS(z)]/dz = 0$$

and solve for z. This is considered for specific forms of the functions $H(w)$ and $D(s)$ in Section V.

Using z^o in a dynamic control policy, we can advance a page at time $t+z^o$, and obtain as a result an expected cost increment of

$$ACI = EWS(z^o)$$

If, on the other hand, we wait for the next page demand to occur, we expect a cost of

$$DCI = \underline{w}(j+1)$$

Thus in order to decide which policy to select at time t, we compute the sign of

$$DCI-ACI = q(j+1)\{\underline{w} - CN(w,s,z^o)\}$$

and either advance the specified page at $t+z^o$ or take no action.

Now that the form of the cost function has been established, we note that if we condition the distributions for w and s, as mentioned above, the notation should be extended in the following fashion:

$$D(s) \longrightarrow D(j,s)$$
$$\underline{s} \longrightarrow \underline{s}(j)$$

where j is the number of pages currently in main storage for the task,

$$H(w) \longrightarrow H(n,w)$$
$$\underline{w} \longrightarrow \underline{w}(n)$$

where n is the number of requests already enqueued at the paging mechanism,

$$q \longrightarrow q(h,k)$$

and h, k, are the indices of the last page requested and the page most likely to be referenced next, respectively. In order to implement a dynamic policy for system programs, the constituents of the formula for DCI-ACI must be computed in advance and included with the program for use during its execution. To follow a stationary policy of advance paging we require only the identity of the most likely successor page for each page of the task and the relevant entries of the matrix $z^o(n,j)$.

IV. THE EXPECTED TOTAL COST

In addition to examining the cost added by each decision in a dynamic policy, it is of interest to write down the total expected cost of a stationary policy: always try to advance the next page or always wait for the demand to occur. For the sake of comparison, we also include swapping as a stationary policy--bringing an entire task into main storage at the beginning of a time slice.

The minimum cost, which is unattainable in practice, is the integral over execution time of the actual storage used (each page is included in this integral from the instant of its first reference to the end of the time slice):

$$MC = \int_{\text{task time}} P(t)\, dt$$

We express the cost of each stationary policy in terms of the amount which is added to this basic cost. Let us suppose that a task has a total of M^1 pages, and that M^0 of them will actually be referenced during the given time slice of duration T^+.

We consider first the additional cost of a swapping policy: at the beginning of a time slice the entire set of M^1 pages is brought into main storage. Thereafter no page request can take place, so the space-time integral is taken only over the initial page-loading delay and the execution time interval. It is reasonable to assume that for a swapping scheme pages of a task would be arranged contiguously in auxiliary storage; and that since page requests for a single task come in bursts, separated by relatively long intervals of execution, queueing for the paging mechanism has an entirely different structure than the demand or advance paging situations. To simplify matters in this discussion we will assume a swap which requires no queueing. Thus a request for M^1 pages takes an average of half a drum rotation to position the drum at the first page, followed by the bulk transfer of M^1 successive pages. In the notation of Appendix B, this requires

$$(m + 2M^1)/2$$

time units, where m is the number of drum sectors and a time unit is the transfer time for one page. Hence the additional cost of a swapping policy is given by

$$CSWP = M^1T^+ - MC + M^1(m + 2M^1)/2$$

Lauer [45] has written down the expected additional cost of a demand paging stationary policy, namely

$$CDEM = \sum_{j=1}^{\underline{M}^0} j\ \underline{w}(n) = M^0(M^0+1)\underline{w}(n)/2$$

or the space-time integral over the M^0 paging delays, each of which takes a mean time $\underline{w}(n)$. If an "advance paging" policy is used over the entire time slice, the expected additional cost is

$$CADV = \sum_{j=1}^{\underline{M}^0} EWS(z^0(n,j))$$

where

$$
\begin{aligned}
EWS(z^o(n,j)) &= \{\underline{s}(j) - z^o(n,j)\} \\
&+ q(h,k)\,(j+1)\,CN(w(n),s(j),z^o(n,j)) \\
&+ \{(1 - q(h,k))\,(j+1)\,\underline{w}(n)\}
\end{aligned}
$$

Although a dynamic policy would, in general, be expected to yield a lower cost than any of the above stationary policies, these cost functions are useful because they illustrate the extreme cases where a dynamic policy would produce a stationary choice. The advance paging formula is also interesting because its terms provide a separation of the kinds of costs entering into the total:

- the first term in each summand $EWS(z^o(n,j)$ is the average cost of advancing a page up to the point at which the demand actually occurs,

- the second term measures the cost introduced by the fact that a demand may occur for the advanced page before it has arrived,

- and the last term shows the expected cost added because the wrong page may have been chosen for advancement.

These total cost functions will be examined in more detail for a specific example in the concluding sections of the chapter.

V. EVALUATION

We pause at this point in the investigation of the costs for allocation policies to re-examine the basic assumptions that were made in deriving the model used in this chapter. An understanding of that reasoning is necessary to interpret the concluding example. Three major conditions have been assumed:

1. Execution time is independent of the storage allocation policy. This is not exactly true for several reasons.

   (a) different policies cause different numbers of interruptions in service, each of which has an associated, fixed CPU overhead,

(b) different allocation policies may require different amounts of CPU time, e.g. to evaluate the formula for DCI-ACI given in Section III,

(c) since allocation policies in general affect task scheduling, a specific scheme could include a variation in the length of a time slice.

The first two effects on execution time are felt to be an order of magnitude smaller than those considered in the model of this chapter. By ignoring the third effect, we are essentially assuming that if time-slice length varies, it does so uniformly for the policies under consideration.

2. A page is kept in main storage from the time it is first referenced until the end of its time slice.

Main memory allocation in existing systems generally uses a page replacement rule which allows a page to be returned to auxiliary storage before the end of its task's time slice. This means that several page faults can occur in practice for the same page. This phenomenon tends to moderate the decrease in the rate of page faults as the number of pages referenced increases. We ignore return to auxiliary memory here because the rate at which this happens depends on the competition of tasks for main storage, and we are dealing with a single-task model. However, this type of analysis applies to the replacement situation if the distribution function D(s) for CPU time between page faults is appropriately adjusted, and the storage use function P(t) is allowed to decrease as well as increase.

3. Each CPU interval is assumed to be terminated by a page fault.

The space-time integral for cost functions in this chapter omits two kinds of time intervals in which a task is unable to use the CPU:

(a) waiting for the completion of a non-paging I/O request

Some kinds of I/O (such as terminal wait) force the end of a time slice, and hence are not of concern here. Others (such as disk I/O) do not, however, and the corresponding increments to the cost function were ignored in the previous analysis. Since these delays occur with the same frequency and duration for all of the storage allocation policies, they introduce a variation in the cost only insofar as different amounts of main storage are held when such waits occur. The difference in the added cost between demand and advance paging, for example, can occur only if an I/O wait falls in the interval between the request for a page and its subsequent demand; and the difference is only the cost for one page held for the duration of the wait. While such differences will be almost negligible, the increase in cost under a complete swapping policy is considerably larger, since the entire program occupies main storage during I/O waits.

(b) waiting to use the CPU

Under conditions of heavy load, a task may accumulate a large space-time integral while in the CPU queue. No cost is associated with this time on the grounds that it is a factor in increasing, rather than decreasing, CPU utilization.

VI. A SPECIFIC EXAMPLE

In this section we compute the parameter $z^o(n,j)$ used in the decision formula when $H(n,w)$ is the distribution function below, which is established in Appendix B for a particular organization of a drum I/O channel.

$$H(n,w) = \sum_{k=0}^{n} G(n,k)\ F(k,w)$$

where the $F(k,t)$ are piecewise linear functions

$F(k,t) = \mathrm{Prob}[c \le t \mid k] =$

(a) 0 if $t < km + 1$,
(b) 1 if $t > (k+1)m + 1$,
(c) $(t-km-1)/m$ otherwise.

The mean value $\underline{w}(n)$ is shown in Theorem B-2 of that appendix to be of the form

$$(m + 2n + 2)/2$$

where m is the number of drum sectors and n is the number of prior page requests enqueued at all the sectors.

In order to simplify the expression for EWS(z), the expected cost of advancing a page at time t+z, we must also assume a specific form for the distribution D(j,s) (which is the length of time s until the next page request occurs, given that j pages have already been referenced during the current time slice). We assume that this request occurs at random, with mean time 1/r(j)--that D(j,s) is the negative exponential distribution with parameter r(j):

$$D(j,s) = 0 \qquad \text{if } s < 0,$$

$$= (1-e^{-r(j)s}) \qquad \text{otherwise.}$$

Given the forms of H(n,w) and D(j,s), we turn our attention to obtaining an expression for

$$CN(w,s,z) = \int_0^{\infty} \int_0^{z+w} (w-(s-z)) \, dD(s) \, dH(w)$$

in the last term of the function EWS(z). The lower limit of the outer integral is not unconditionally zero, however, because z may be negative: the occurrence of a negative value for z means that the page request is to have been made before the decision point. In practice, the decision point will probably be the time at which the task is given the CPU. We thus allow a negative z and interpret it as the initiation of a page request before the task is dispatched. The cost up to that time will be assessed for the requested page, and of course a page demand occurs before the decision point with probability zero.

Applying the definition of D, we have

$$CN(w,s,z) = \int_{\max[0,-z]}^{\infty} \{(z+w)D(z+w) - \int_{0}^{z+w} s\, dD(s)\}\, dH(w)$$

$$= \int_{\max[0,-z]}^{\infty} \{C^1 + C^2 w + C^3 e^{-r(j)w} + C^4 w e^{-r(j)w}\}\, dH(w)$$

where the multipliers constant with respect to w have the forms

$$C^1 = (z+r(j)^{-2}), \qquad C^3 = -e^{-r(j)z}(z+zr(j)^{-1}+r(j)^{-2})$$

$$C^2 = 1, \qquad C^4 = -e^{-r(j)z}(1+r(j)^{-1}).$$

In order to use the definition of H, we let i be chosen so that

$$im+1 < -z \leq (i+1)m+1$$

and make the understanding that if z is non-negative (i.e. i is negative) we ignore all terms in the formulas to come which would have a negative index. Then for each term f(w) in the integral above, we write

$$\int_{\max[0,-z]}^{\infty} f(w)\,dH(w) = \{G(n,i)/m \int_{\max[1,-z]}^{(i+1)m+1} f(w)\,dw$$

$$+ \sum_{k=i+1}^{n} G(n,k)/m \int_{km+1}^{(k+1)m+1} f(w)\,dw\}$$

Next, we obtain the minimal cost value z^o of z by setting the derivative of EWS(z) equal to zero:

$$EWS'(z) = q(h,k)(j+1)CN'(w(n),s(j),z(n,j)) - 1 = 0$$

The general form of this equation is

$$A^1 z e^{-r(j)z} + A^2 e^{-r(j)z} + A^3 z + A^4 = 0$$

where the coefficients contain the G(n,k), the number of drum sectors m, and the page request rate r(j). The solution to the equation EWS'(z) = 0 can be evaluated for appropriate values of the parameters n and j. It provides the value $z^o(n,j)$ which minimizes the expected cost for advance paging.

Further study of the MTS data will provide better information about the frequency of page requests and their dependence on other factors. A numerical approximation of EWS(z) can be obtained even if the distributions are less tractable analytically. Thus the drum delay distribution could also be replaced by the appropriate distribution for another paging device, and modified to include anomalies observed in actual operation.

VII. NUMERICAL RESULTS

We present here a brief numerical investigation of the total expected cost functions CSWAP, CDEM, and CADV for stationary storage allocation policies. The general forms of these functions were given in Section IV, and we use the specific distribution functions discussed in the previous section.

A detailed study of the relationship of the paging rates $r(j)$ to the number j of pages so far referenced is beyond the scope of this work. Data for exploring this relationship is available, however, in the normal output of the MTS data collection facility. It will suffice for our comparison of cost functions to assume a simple theoretical relationship between j and $r(j)$. The general form of the function $P(t)$, which gives the number of pages so far referenced, is like that of a negative exponential: the rate at which new pages are referenced decreases steadily as more of them become resident in main storage. This phenomenon is described by Fine, *et al.* [28], Denning [17], and Coffman and Varian [11].

We recall that $r(j)$ is the rate of arrival of the (random) j-th page request. By postulating an exponential form for $P(t)$, we specify the dependence of $r(j)$ on j: let us write

$$P(t(j)) = Q\,(1-e^{-Rt(j)})$$

for the value of P at the point $t(j)$ [$j=1,2,\ldots,M^0$] where the j-th page request occurs. From the form of P, we see that

$$t(j) = \ln(1-j/Q)/-R = \ln(Q/(Q-j))/R$$

The mean time before the j-th page request is thus described by

$$t(j)-t(j-1) = \underline{s}(j) = \ln((Q-j)/(Q-j-1))/R$$

or in terms of the arrival rate we have

$$r(j) = R/\ln((Q-j)/(Q-j-1))$$

where Q and R are parameters used to adjust the height and the slope of $P(t)$.

It was shown in the last section that for a specific drum distribution function $H(n,w)$ and a negative exponential page request arrival distribution $D(j,s)$, the optimal time to advance a page is $t+z^0(n,j)$, where t is the decision point and $z^0(n,j)$ is the solution of an equation of the form

$$f(z) = A^1ze^{-r(j)z} + A^2e^{-r(j)z} + A^3z + A^4 = 0$$

where the coefficients depend on n and j, as well as the parameters Q and R used to derive r(j). This equation can be solved for $z^o(n,j)$ with an ordinary Newton-Raphson iteration:

$$z(k+1) = z(k) - f(z(k))/f'(z(k))$$

for appropriate values of the parameters. However, as we can see from a few examples of EWS(z) in Figure 2-1, the slope of this function throughout the neighborhood of the minimal value is very small. Thus an iterative procedure like the Newton-Raphson method converges slowly and is prone to wide oscillations. In view of some experience with these difficulties and the existence of good bounds for the location of the minimal value, a simple search technique was used to approximate the optimal values in the calculations for the figures discussed in the remainder of this section.

Figure 2-2 shows a range of values of the total additional cost functions CDEM (demand paging) and CADV (advance paging) for increasing amounts of congestion in the drum I/O channel. For the given choice of page demand rate (which depends on the parameters Q and R) and all probabilities of correct next page selection (PR) taken to be .5, a stationary demand paging policy costs more than the stationary advance paging policy. The swap policy cost CSWP, shown here for only the case of no drum queueing at all, would presumably lie above the CDEM curve for any method of organizing the pages of a swapped task in the drum sectors. Thus the horizontal line showing this cost in Figure 2-2 is misleading. The notation "M0=10, M1=15" means that 10 of a total of 15 pages in the task are actually referenced in the hypothetical time slice. For a comparison of its relative magnitude, the minimum cost (MC) is also shown in the figure. The total cost of each policy is the plotted value _plus_ the minimum cost.

Figure 2-3 provides an indication of the sensitivity of the (stationary) advance paging policy to the quality of information available about page references. For three different values of the paging load parameter N, the function CADV is seen to decrease sharply as the probability of choosing the correct next page tends toward unity. The corresponding values for CDEM (which do not depend on this information) appear as horizontal lines. For the given values of the other parameters, we see that an advance paging policy is less expensive than a demand paging policy as long as the probabilities exceed one-half. With the typical probability of a wrong guess exceeding one-half, the cost of the advanced pages and the expectation of further demands combine to make the advance policy more expensive.

Another item of interest is the sensitivity of the cost functions to variations in the percentage of a task's pages actually referenced during a given time slice. This relationship is illustrated in Figure 2-4. Since the cost of each page demand is rather high, the principal advantage of the demand paging policy over swapping rests on the fact that the fraction of referenced pages is not usually close to one. For the example of Figure 2-4, CDEM is less than CSWP until this fraction exceeds about .65, and the corresponding advance policy retains the edge over swapping until nearly 80% of the pages are referenced. Again, we emphasize that the swapping policy illustrated assumes no drum congestion, whereas the other cost functions are given for a realistic load. Anyone seriously investigating a swapping policy could obtain a "referenced pages fraction" from the MTS data collection facility and compare the true costs for a specified drum storage allocation procedure with swapped tasks. Because of the way the page demand function was represented for these calculations, the length of the time slice increases as more pages are referenced. Hence even the "very stationary" swapping policy shows an increase in cost values in Figure 2-4. The minimum cost MC is also shown in this figure. We see again that this actual execution time-space integral is quite small in relation to the additional costs of the storage allocation scheme. Note that only when almost 100% of a task's pages are referenced is the minimal cost as large as the additional swapping cost.

Finally, we portray in Figure 2-5 the relationship of the costs to changes in the parameter R of the rate at which a task requests pages. Because the length of a time slice is very sensitive to this parameter, we display values for the cost functions normalized by the time slice length. After the other figures, the curves in the present figure offer no surprises. The demand cost is most sensitive to variations in this request rate. With two-thirds or more of the pages being referenced (M0=10, M1=15), the no-congestion swap policy is cheaper than the other policies over most of the range of values shown for R. Although advance paging "resembles" demand paging in structure, it is somewhat less sensitive to changes in this parameter. It is again a distinct improvement over the given range, at least for the given page choice probabilities.

In summary, our numerical calculations show that in at least the cases examined, a stationary advance paging policy provides a smaller average storage use than demand paging, and offers the most improvement over the latter policy in those cases when demand paging is at its worst, i.e. when pages are referenced at high rates and/or most of the pages of a task are used during each time-slice. Since the overall pattern of page request rates was assumed to be quite "bad" for the purposes of these calculations, we believe the numerical results indeed provide a lower bound for the performance improvement expected with a dynamic policy which selects the strategy for each page request that has the smaller expected cost.

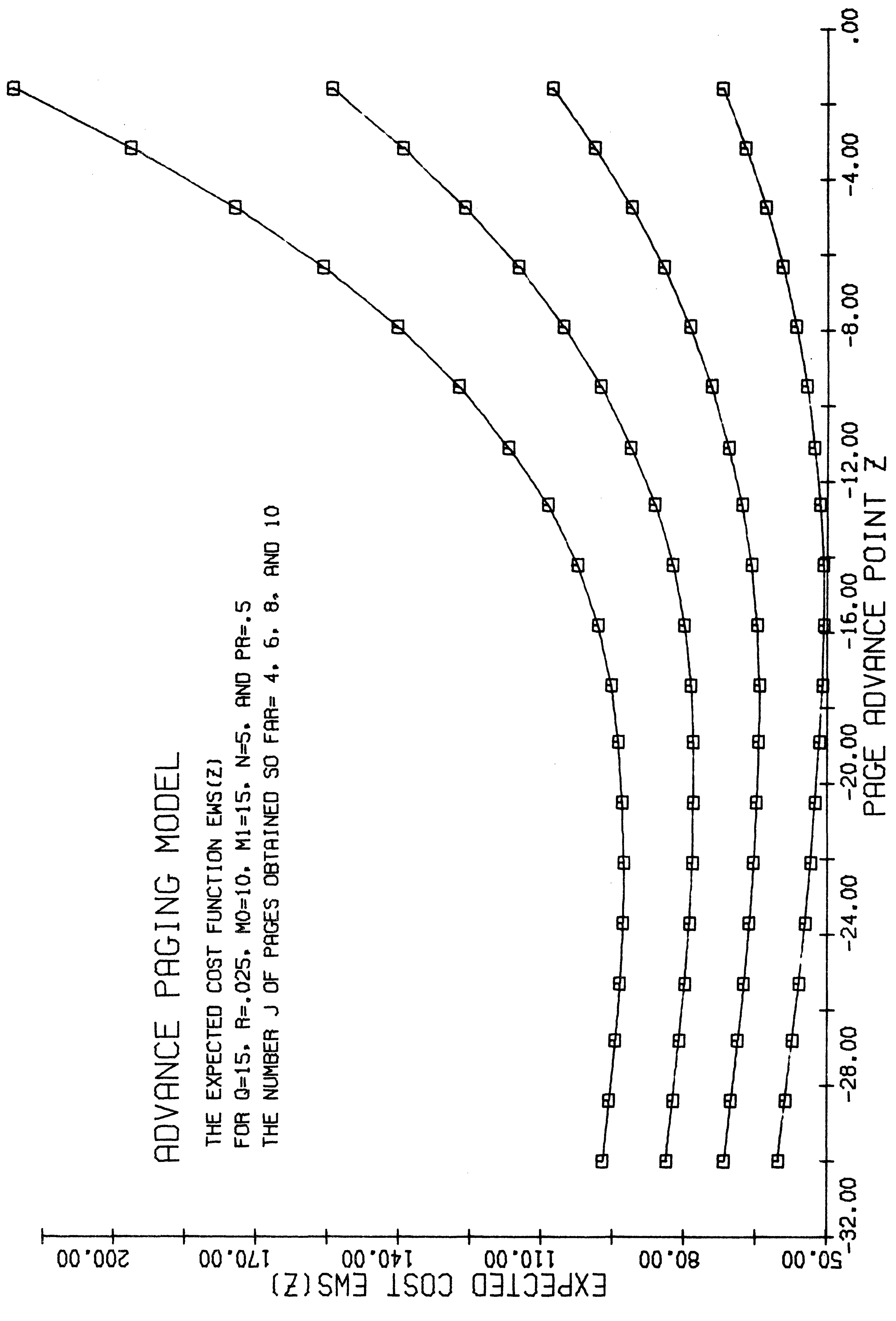

Figure 2-1. Examples of the Expected Cost Function

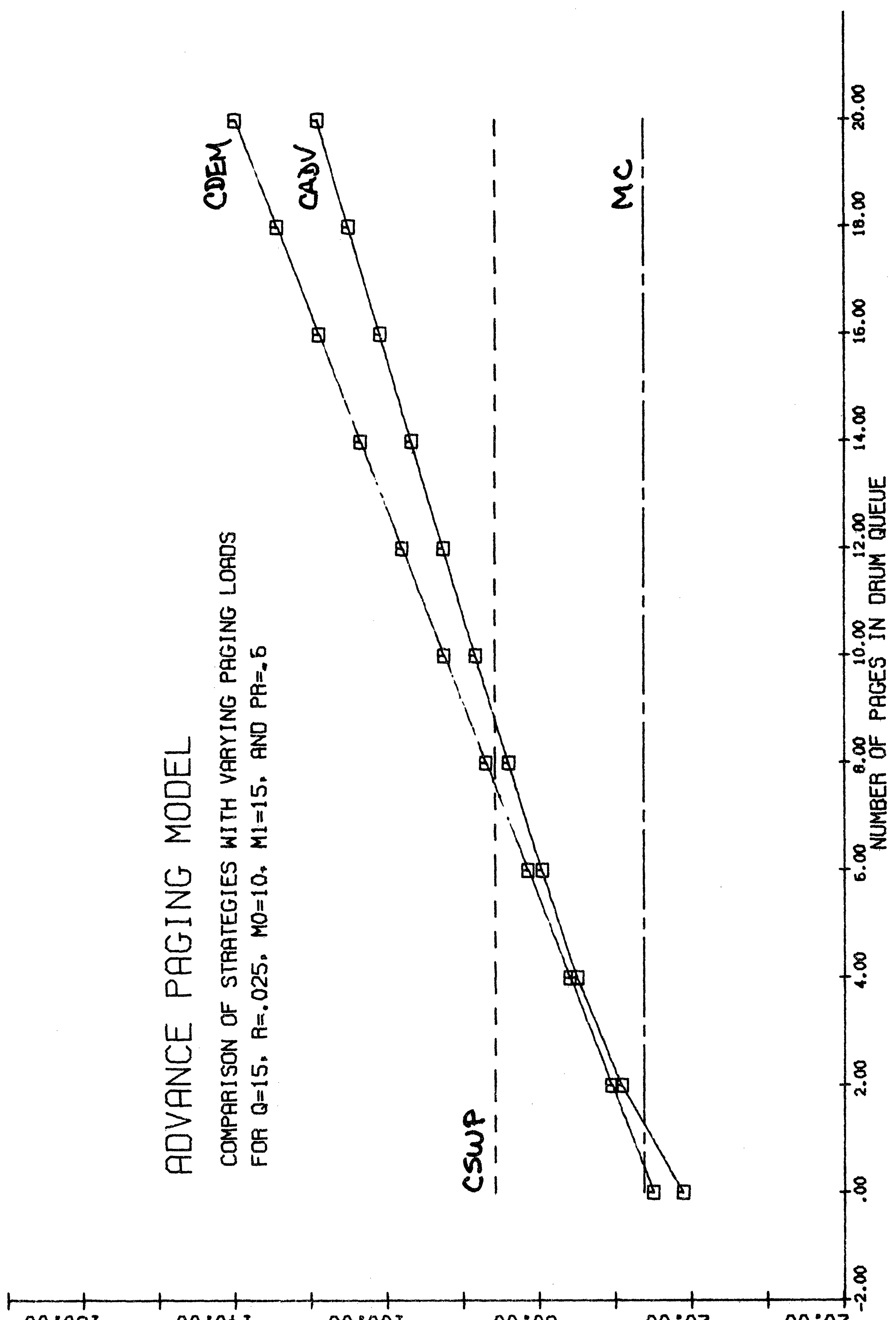

Figure 2-2. Effect of Drum Congestion on Cost

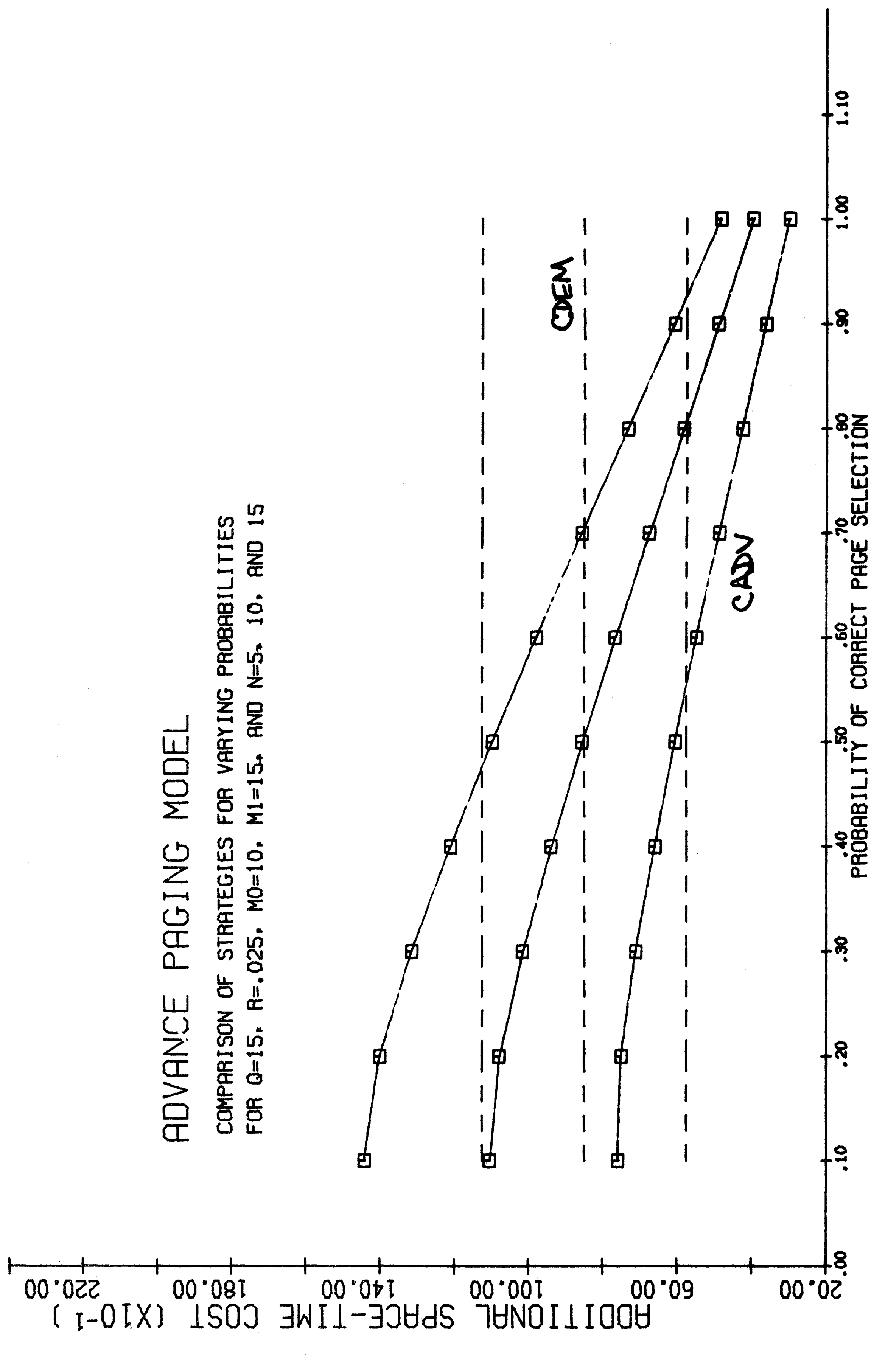

Figure 2-3. Effect of Reference Probability on Cost

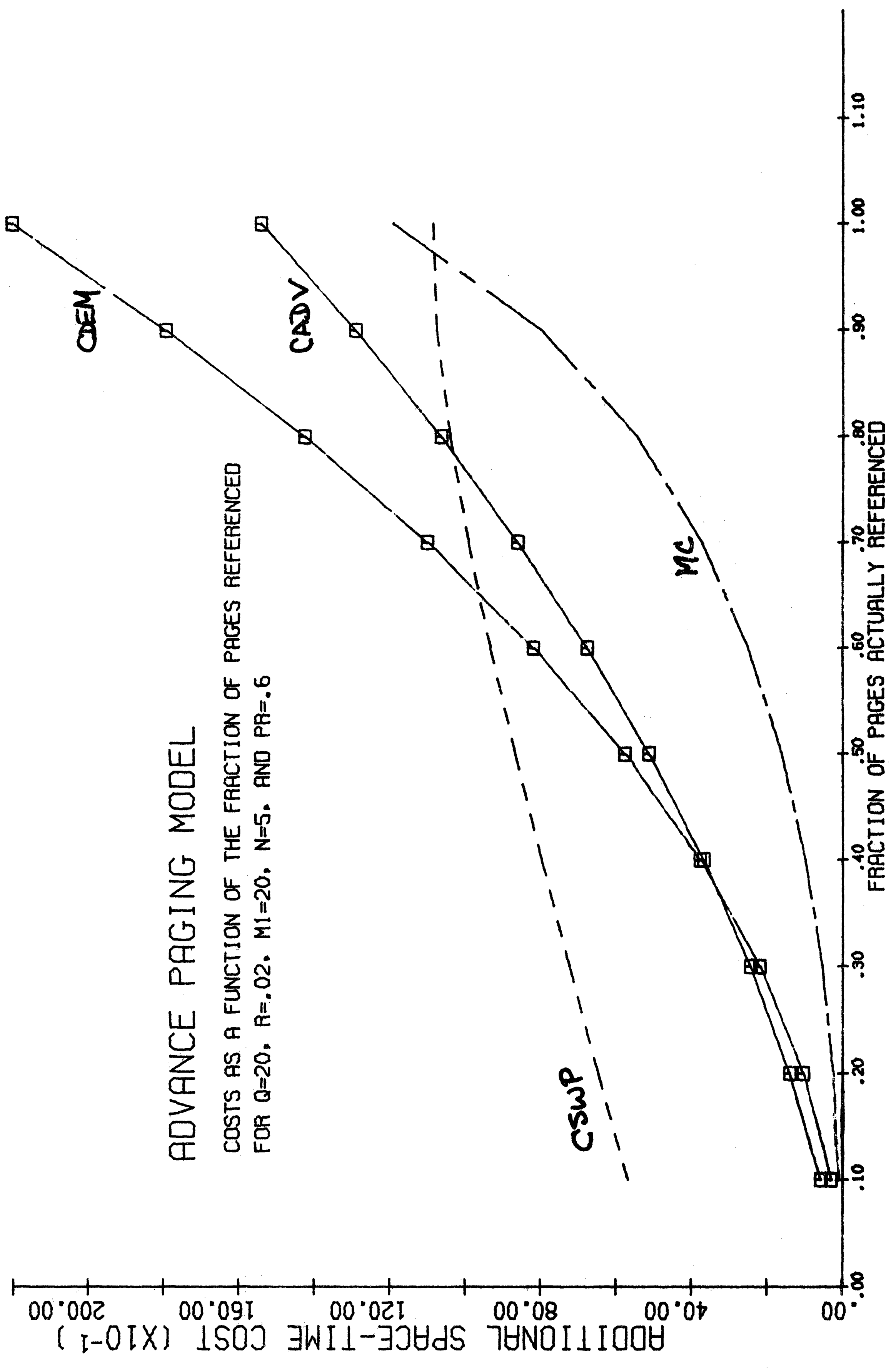

Figure 2-4. Effect of Number of Pages on Cost

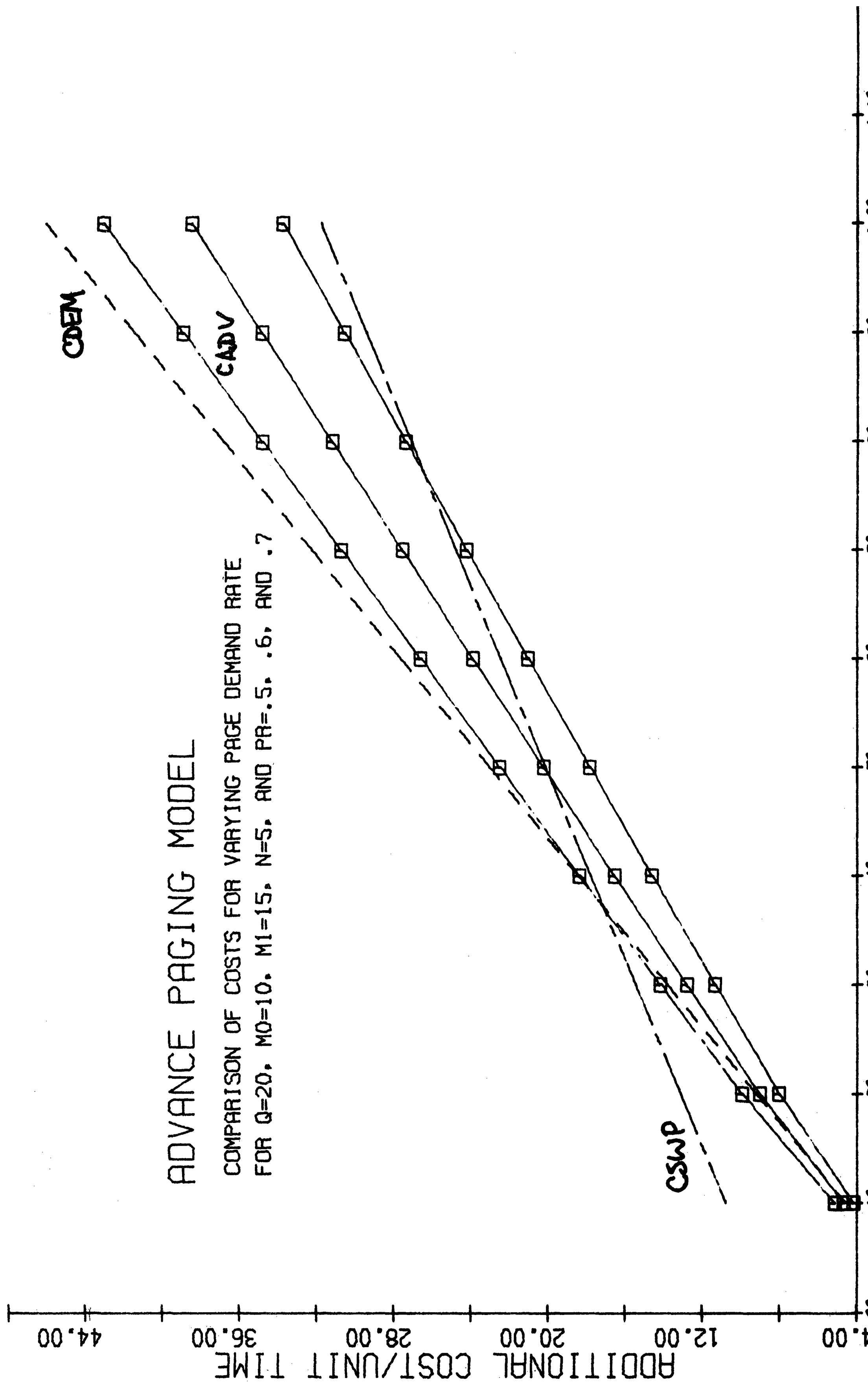

Figure 2-5. Effect of Page Request Rate on Cost

CHAPTER 3. DYNAMIC ALLOCATION FOR MANY PROGRAMS

"It is sometimes felt that when phenomena include men, it is tremendously more difficult to theorize successfully..."

J. D. Williams

"Although this may seem a paradox, all science is dominated by the idea of approximation."

Bertrand Russell

In this chapter we present an analytical model for a set of programs which alternate between paging operations and the use of a (single) CPU. Execution times between page requests are assumed to be exponentially distributed, and the duration of paging operations is given by an Erlangian distribution (see Appendix B). The model given here provides an estimate of the general distribution of queue lengths in front of the CPU, and hence the expected CPU busy fraction and the output of the system in CPU intervals per unit time. It can be extended to handle multiple processors and/or paging mechanisms.

I. MOTIVATION

We wish to examine the effects on productivity found in a virtual storage computer system when different numbers of tasks are present to compete for the use of the CPU. If the system chooses to multiprogram between too few tasks over a certain period of time, there is not enough demand for execution generated by these tasks between paging and I/O operations to keep the CPU appreciably busy. If, on the other hand, too many tasks are in competition at any one time, they each have a smaller share of real storage, and hence tend to request pages at a higher rate. This effect, when added to the already higher paging rate due to more tasks present, may increase the congestion in the paging mechanism to the point where the CPU is again idle a large share of the time (and more CPU time is required for the overhead of operating the paging mechanism). It is also of interest to estimate the effect on performance of adding a second CPU or paging drum. This model provides performance estimates for each such situation (in which all the parameters are fixed). By evaluating these results for a number of combinations of the parameter values, it is possible to get a rough idea of the "feasible subspace" of values of the parameters which can be controlled, and of the sensitivity of the system to variations in specific directions.

II. MODEL DESCRIPTION

We suppose that the system over the period of study is closed, containing a fixed number, U, of tasks. Each task is assumed to need only the information contained in its virtual memory: the only input/output is via the paging mechanism. Thus a task cycles through four states,

- queued at the CPU,
- in execution,
- queued at the paging drum,
- transferring a page.

The execution intervals of different tasks are required to be independent, exponentially distributed random variables, all with the same mean value S. The queueing time plus service time of the paging drum processor is assumed to be distributed according to an Erlangian distribution, which is shown in Appendix B to be a good approximation to the actual distribution for a particular drum I/O channel.

The movement of tasks through the model is pictured in Figure 3-1. Each of the k exponential stages of the drum transfer distribution appears in the model as a "queue with ample servers" [15], that is, the service of all "customers" at that stage proceeds concurrently according to the same exponential rate R, with the output process of that stage being exponential with rate uR, conditional on the number u of customers at the given stage. Since any stage of the drum model accepts each customer for service as soon as he arrives, the only genuine queue in this model forms in front of the CPU, and it is the distribution of its queue length that we wish to study.

The results in this chapter are an adaptation of the work of E. Koenigsberg [43], who stated and solved the equilibrium equations for a cycle of single-server stages.

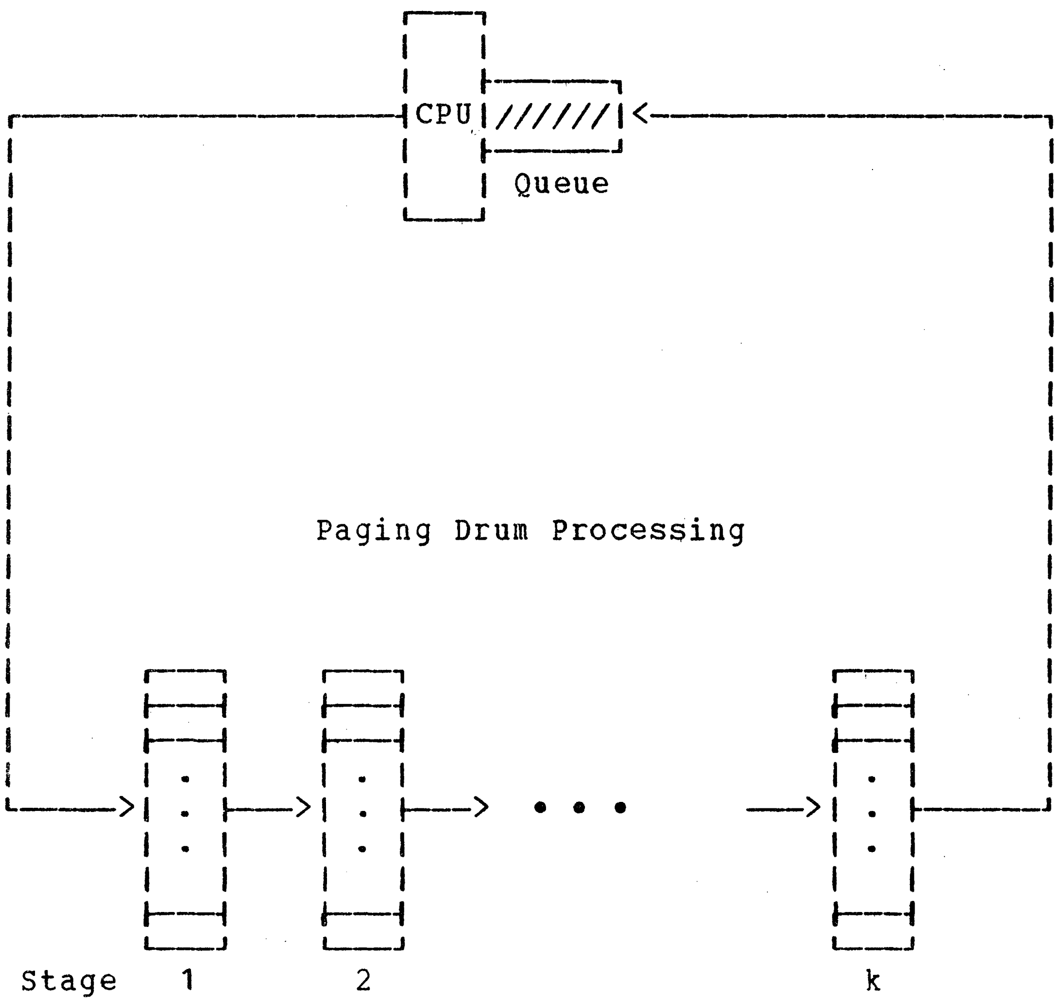

Figure 3-1. The Queueing Model

III. ANALYSIS

We number the CPU as the k+1st service point in the cycle. The equilibrium probability that there are u(i) customers at stage i, {i=1,2,...,k+1}, will be denoted by p[u(1),u(2),...,u(k+1)]. Let the output rate at stage i, which depends on the number of customers at that stage, be denoted by $v[i,u(i)]$. Then the equilibrium probability equations are

$$\sum_{i=1}^{k+1} v[i,u(i)]\ p[u(1),u(2),\ldots,u(k+1)] =$$

$$\sum_{i=1}^{k+1} v[i,u(i)+1]\ p[u(1),\ldots,u(i)+1,u(i+1)-1,\ldots,u(k+1)]$$

subject to the following conditions:

(a) $\sum_{i=1}^{k+1} u(i) = U$

(b) the i-th term does not appear in the equation if either $u(i) = 0$ or $u(i+1)-1 < 0$.

(c) the k+1st stage is linked back to the first, in the sense that the last term on the right is actually

$$v[k+1,u(k+1)+1]\ p[u(1)-1,u(2),\ldots,u(k-1),u(k+1)+1].$$

(d) the output rates $v[i,u]$ have the form

$v[k+1,u] = S$ for all u, and
$v[i,u] = uR$ for all u and $i < k+1$.

The solution of the equilibrium equations is given by

$$p[u(1),\ldots,u(k+1)]$$

$$= p[0,\ldots,0,U]\ v[k+1,u(k+1)]^{U-u(k+1)} \Big/ \prod_{i=1}^{k} v[i,u(i)]^{u(i)}$$

where $p[0,\ldots,0,U]$ is given by the expression

$$1/\Big[\overline{\sum_{\{k+1,U\}}} v[k+1,u(k+1)]^{U-u(k+1)} \Big/ \prod_{i=1}^{k} v[i,u(i)]^{u(i)}\Big]$$

and the notation $\{k+1,U\}$ indicates that the summation is over all sets of $k+1$ non-negative integers $u(i)$ which sum to U. It is also to be understood that for all cases where $v[i,u(i)] = 0$, the term containing it in the denominator has value 1. The probability that exactly J customers are at the CPU stage is thus (applying the definitions of the $v[i,u]$)

$$P[J] = \sum_{\{k,U-J\}} p[u(1),\ldots,u(k),J]$$

$$= (S/R)^{U-J} \sum_{\{k,U-J\}} \left[1/ \prod_{i=1}^{k} u(i)^{u(i)} \right] p[0,\ldots,0,U]$$

$$= a(U-J)\, X^{U-J} \Big/ \left[\sum_{\{k+1,U\}} (S/R)^{U-u(k+1)} \Big/ \prod_{i=1}^{k} u(i)^{u(i)} \right]$$

$$= a(U-J)\, X^{U-J} \Big/ \left[\sum_{i=0}^{U} (S/R)^{U-i} \sum_{\{k,U-i\}} 1/ \prod_{j=1}^{k} u(j)^{u(j)} \right]$$

$$= a(U-J)\, X^{U-J} \Big/ \sum_{i=0}^{U} a(i)\, X^{i}$$

where we let $X = (1/R)$ and include the powers of S in the coefficients $a(i)$. Of particular interest is $P[0]$, the probability that the CPU is idle. The mean number at the CPU stage is of course

$$MN = \sum_{i=1}^{U} i\, P[i]$$

Since the CPU is working on the average $1-P[0]$ of the time at the rate S, its output in number of execution intervals per unit time is given by

$$OP = (1 - P[0])\, S$$

We digress briefly at this point to indicate the way this model may be extended to handle multiple components. Appendix B, Section IV discusses the change in form of the drum delay distribution to accomodate multiple paging mechanisms. Such a modification alters the value of R and possibly the number of stages in the Erlang approximation of the drum delay distribution, but the cyclic queue model need not otherwise be changed. The structure of the model of this chapter is altered, however, when we consider multiple processors.

Suppose that the single CPU in this model is replaced by a (symmetric) pair of processors, sharing a common CPU queue. The effect of this change is to double the output rate of the CPU stage whenever it has at least two customers. This can be accomplished by making the replacement

$$v[k+1,u(k+1)] \longrightarrow 2v[k+1,u(k+1)]$$

in each equilibrium equation for which $u(k+1) \geq 2$. In the notation of this section, we replace S by 2S in the formula for P[J] whenever $J \geq 2$. Noting that the powers of S occur in the coefficients a(i), let us write $a^2(i)$ to denote the coefficients defined above with 2S substituted for S. We can then express the probability distribution for CPU queue length in the duplex case as follows:

$$P^2[1] = Sa^2(U-J)X^{U-1} / [\sum_{i=0}^{U} 2a^2(i)X^i - a(U)]$$

and for all $J \neq 1$,

$$P^2[J] = a^2(U-J)X^{U-J} / [\sum_{i=0}^{U} 2a^2(i)X^i - a(U)]$$

The output of the duplex system is

$$OP^2 = (1 - P[0] - P[1])2S + P[1]S$$

$$= (1 - P[0] - P[1]/2)2S$$

CPU intervals per unit time.

IV. APPLICATION

At this point we recall that the formulas obtained in the preceding section were based on the assumption that the exponential rates R and S of the service stages were fixed throughout the analysis. While S can certainly be taken as constant, the value for R is in fact dependent on the amount of congestion in the drum I/O channel, or in terms of this model, on U-MN. Thus the value of the parameter R is determined, as it were, by the solution. This suggests an iterative process for determining the value of R for which one will find the values of P[0] most meaningful:

(a) choose an initial value R(0) for R,

(b) compute the number of tasks in page wait (not at the CPU stage) at equilibrium,

(c) use the equilibrium number of tasks in page wait for R(h), together with the Erlangian model of the drum completion time distribution, to establish a new value for R, say R(h+1),

(d) repeat steps (b), (c), and (d) until two succeeding values differ by less than a prescribed amount.

It is shown in the next section that the sequence of iterates for R converges. At this point the system of queues has reached equilibrium for the rate of drum service which most closely represents the true rate for the equilibrium load.

The iteration described above is computationally feasible, despite the complicated expressions involved, because the number of terms in each summation is actually quite small: the number k of exponential drum stages is typically only 2, 3, or 4, and the interesting numbers of tasks (U) are usually less than ten. In addition, the coefficients can be computed in advance of the iteration process itself.

In order that the results of this chapter be directly usable in a supervisor scheduling algorithm to improve CPU utilization, a further optimization problem is encountered: one would like to know the number U of active tasks which maximizes CPU usage when all the other parameters have been fixed. In other words,

$$\min_{U} P[0]$$

established above as the number not at the CPU stage, less one for the task whose page is currently being transferred:

$$n = U - MN - 1$$

The mean of the drum distribution is given by

$$m = (M + 2n + 2)/2 = U + M/2 - MN$$

where M is the number of drum sectors (see Appendix B). Thus in the presence of $n-1$ other transfer requests, we expect a page to pass through the k drum stages (each of which services a single request at average rate R) in the total time m:

$$m = k/R$$

In what follows we will add an iteration index h to each of the values computed from the solution to the equilibrium equations using the iterate $R(h)$; for example MN becomes $MN(h)$, and for $P[J]$ we write $P[J,h]$. We will assume that the number of model service stages remains fixed during the iteration, and thus write

$$m(h+1) = k/R(h+1) = U + M/2 - MN(h)$$

The sequence $R(h)$, $[h=0,1,\ldots]$ converges if and only if the sequence $m(h)$, $[h=0,1,\ldots]$ does; for, as we shall see, the values of $m(h)$ lie in a closed interval not containing zero.

Theorem 3-1.

The sequence of values of $m(h)$, $[h=0,1,\ldots]$, converges provided $m(0)$ satisfies the relations

$$M/2 \le m(0) \le U + M/2$$

Proof: We show first that all iterates lie in the indicated interval provided the first one does, and in that interval they form a monotone sequence. Recalling that $MN(h)$ is a mean value for the number of tasks at the CPU stage, we see that $0 \le MN(h) \le U$. This shows, from

$$m(h+1) = U + M/2 - MN(h)$$

that m(h+1) lies in the indicated range. We now show that each difference

$$m(h+1) - m(h)$$

of successive members of the sequence preserves the sign of the previous difference:

$$m(h+2)-m(h+1) = MN(h+1) - MN(h+2)$$

$$= \sum_{i=0}^{U} ia(i)X(h)^{i} \Big/ \sum_{j=0}^{U} a(j)X(h)^{j}$$

$$- \sum_{i=0}^{U} ia(i)X(h+1)^{i} \Big/ \sum_{j=0}^{U} a(j)X(h+1)^{j}$$

= NUM/DEN, where DEN > 0 and

$$NUM = \left[\sum_{i=0}^{U} ia(i)X(h)^{i} \right] \left[\sum_{j=0}^{U} a(j)X(h+1)^{j} \right]$$

$$- \left[\sum_{i=0}^{U} ia(i)X(h+1)^{i} \right] \left[\sum_{j=0}^{U} a(j)X(h)^{j} \right]$$

$$= \sum_{i=0}^{U} \sum_{j=0}^{U} ia(i)a(j)X(h)^{i} X(h+1)^{j}$$

$$- \sum_{i=0}^{U} \sum_{j=0}^{U} ja(i)a(j)X(h)^{i} X(h+1)^{j}$$

$$= \sum_{i=0}^{U} \sum_{j=0}^{U} (i-j)a(i)a(j)X(h)^{i} X(h+1)^{j}$$

and now if we let $k = i-j$ if $i-j > 0$, and $k = j-i$ otherwise, we obtain

$$= \sum_{k=1}^{U} k \left\{ \sum_{k \le i \le U} a(i)a(i-k)X(h)^{i} X(h+1)^{i+k} - \sum_{0 \le i \le U-k} a(i)a(i+k)X(h)^{i} X(h+1)^{i-k} \right\}$$

$$= \sum_{k=1}^{U} k \left\{ \sum_{k \le i \le U} a(i)a(i-k)X(h)^{i} X(h+1)^{i+k} - \sum_{k \le i \le U} a(i-k)a(i)X(h)^{i+k} X(h+1)^{i} \right\}$$

$$= \sum_{k=1}^{U} k \left\{ \sum_{k \le i \le U} a(i)a(i-k)X(h)^{i} X(h+1)^{i} [X(h+1)^{k} - X(h)^{k}] \right\}$$

where all but the quantity

$$[X(h+1)^{k} - X(h)^{k}]$$

is positive and the latter retains the sign of the difference of the single powers. Recalling the definition of $X(h)$, we see that

$$m(h+1) > m(h) \implies 1/R(h+1) > 1/R(h) \implies$$
$$X(h+1) > X(h) \implies m(h+2) > m(h+1)$$

and similar reasoning prevails when the signs are reversed. We have seen that both the m- and R-sequences are monotonic on closed intervals, and hence they converge.

V. NUMERICAL RESULTS

The computational procedure given in Section IV has been used to prepare Figures 3-2 to 3-4, which illustrate some of the characteristics of the model results when they are computed with parameter values obtained from the MTS data. Figure 3-2 demonstrates the effect of changing the mean CPU service time on the overall CPU utilization. With four or more tasks in progress (using a 9-sector drum), there is only a small decrease in idle time after mean service time increases above about 8 msec. (Each of the figures in this section assumes a drum transfer rate of one page every 4 msec.) Figure 3-3 shows the relationship between number of tasks and CPU utilization for a few fixed values of CPU service time, in this case for a drum with four sectors. We see that for all but the smallest values of CPU service time, increasing the number of tasks beyond four or five has little effect on productivity.

To compare the 9- and 4-sector drums, we see in Figure 3-2 that if CPU service averages 6 msec. with four tasks, the CPU is kept busy less than 80% of the time. From Figure 3-3 we read a CPU utilization of over 90% for the 4-sector drum. Several of the lines plotted in Figure 3-3 exhibit abrupt changes in slope between a few pairs of the plotted points. These occur at places where the optimal number of drum service stages changes from one small integer to another. (These numbers are given in Figure B-3, Appendix B). Since the calculations for the figures always use the best number of stages, the graphs are essentially composites based on a number of different models.

Figure 3-4, which gives curves of equal CPU utilization relating the factors of CPU service time and number of tasks, shows that increasing the number of tasks beyond the optimum causes unproductive fighting among them for main storage: as the number of tasks increases, a small decrease in mean CPU time between paging requests (due to the fact that each task may use a smaller average amount of main storage), can cause a shift from one of the isometric curves to the next (they are very close together). Each such shift represents a decrease in productivity of 20%.

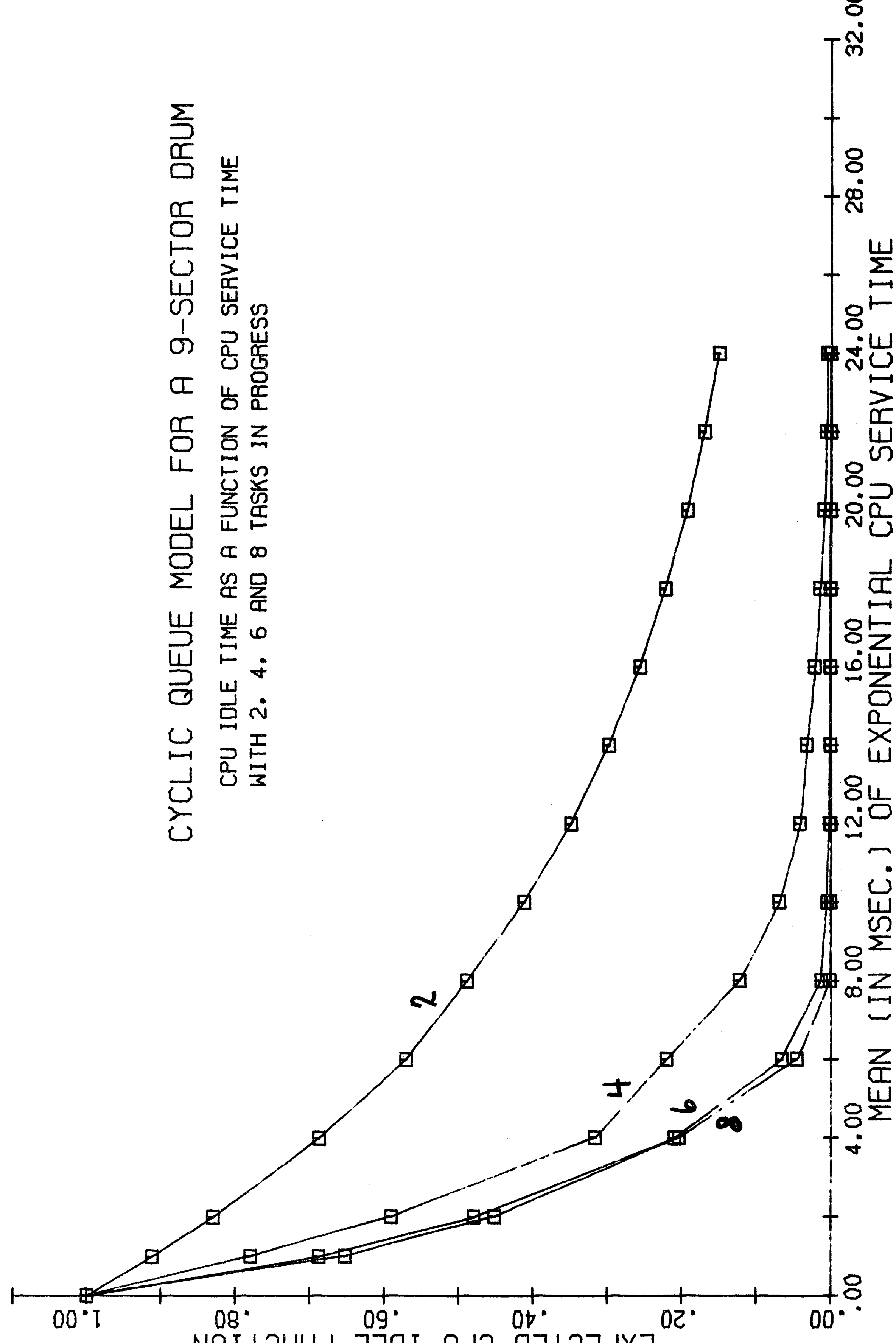

Figure 3-2. Effect of CPU Service Time on Idle Time

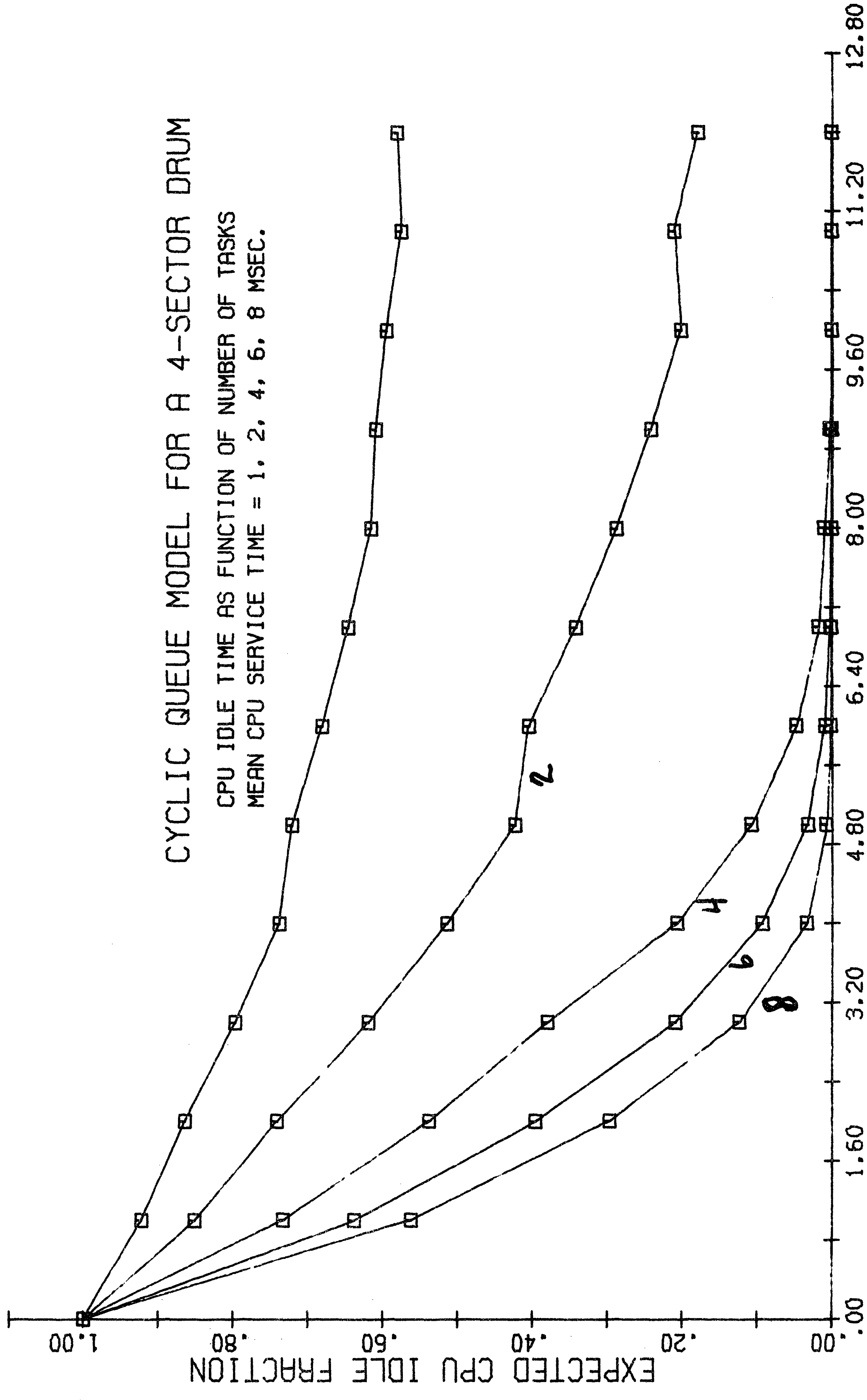

Figure 3-3. Effect of No. of Tasks on Idle Time

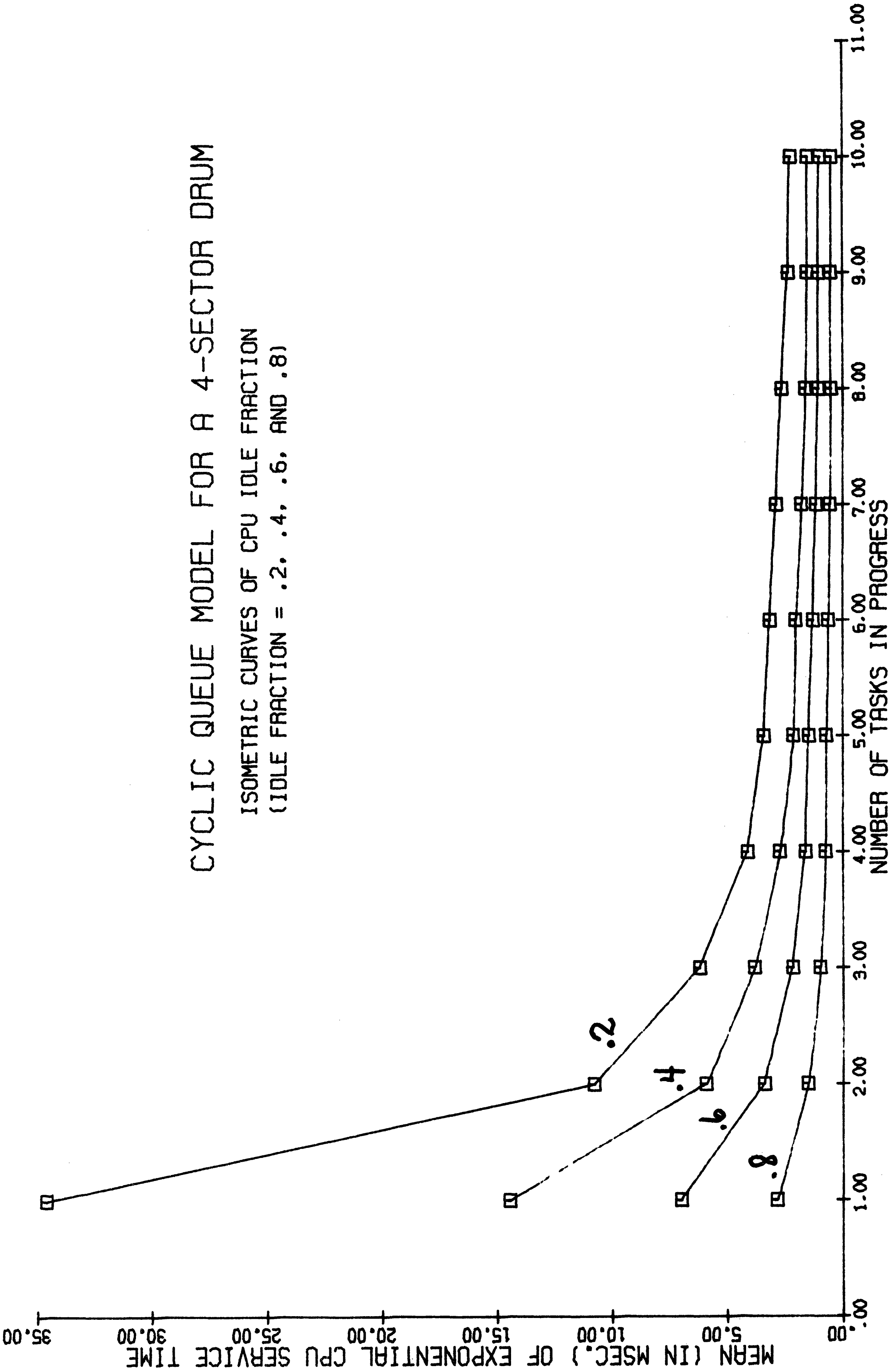

Figure 3-4. CPU Rate vs. No. of Tasks

CHAPTER 4. SIMULATION OF STORAGE ALLOCATION

"Unquestionably, for the moment, numbers are king."

TIME Magazine

In this chapter we report the results of a digital simulation of a virtual storage computing system. Elements of the model include an exact representation of the paging drum described in Appendix B, the flow of pages between main and auxiliary storage, the control of tasks during time slices, and data abstracted directly from the MTS system.

I. THE MODEL

The simulation model is described in detail in Appendix E, where a complete listing of the GPSS/360 program also appears. Briefly, its characteristics are these: a record of tasks is kept during a single time slice, beginning with the scheduling of the first page request. Once requested, each page is brought into main storage to remain until the end of its task's time slice. Since no page can be brought into full storage, there is the possibility of queueing for space. The time unit in the model is the page transfer time from the drum to main storage. Phenomena occurring in shorter time intervals, such as CPU/channel conflicts for storage reference cycles, are ignored. Processing time for scheduling and interrupt posting is also not modelled.

We assume an overloaded operation: there is always a task ready to enter the system. No classification of tasks or representation of activities is made beyond the level of a single interaction: the time from the end of a terminal I/O wait until the start of another. The end of a time slice may also occur if a task requests more than a specified amount of processing time, in which case the task is enqueued for another CPU service interval. If a page is requested which makes the main storage total for that task greater than a certain maximum, the task is terminated as if it had taken an I/O wait for the terminal. At any given instant a task is

- executing or enqueued for execution,

- waiting for a page to be brought from auxiliary storage,

- or waiting for the completion of a synchronous (e.g. disk) I/O operation

The simulation model represents two types of events: the flow of tasks between the aforementioned states and the induced flow of pages to and from main storage.

II. LIMITATIONS

There are a number of limitations inherent in the model just described. The decision not to maintain the identity of individual pages in the model was made in order to reduce the complexity of the model, and because it was felt that this aspect of task representation requires more information about program behavior for less increased accuracy than any other. (The data collection facility can provide such information, however.) Given that the identity of pages is not retained, one is essentially restricted to modelling tasks when they can actively compete for the use of the CPU. Another restriction arising from this assumption is that it is difficult to model a replacement rule: once a page enters core it is hard to distinguish a page-out and subsequent reference to that page from a first reference to a new page.

While the use of actual data to describe the individual task activities in general enhances the accuracy of the model a great deal, it also fixes that data so that some variations in behavior cannot be explored. For example, the page request rate is a function of the competition for main storage in the real system at the time the data was taken, and it cannot easily be varied in the model as scheduling policies are used which enforce greater or less demand for simulated storage. Another characteristic which depends on the actual load is the possibility or "reclaiming" a page in main memory which has remained there from a previous time slice or can be recovered while an attempt is being made to write it out.

The one item of data most desirable, and yet hard to obtain by software measurement techniques, is the pattern of actual inter-page references. Knowledge of this data is necessary to allow model adjustments to accurately represent program behavior in a limited amount of main storage under varying loads. On a very limited scale, the MTS facility can be used to obtain that information by artificially inducing background loads of specified proportions. Actual references can be observed by temporarily altering the supervisor to operate dynamic relocation with empty page tables, so that every off-page reference results in a page fault. This latter technique significantly alters program execution time, however, and thus distorts other information about tasks—such as I/O delays—which are sensitive to changes in time scale.

III. EXPERIMENTS

In order to validate the theoretical results it was initially felt that the simulation model should be designed to be as much like the real system as the chosen level of detail allowed. In fact, this was the principal motivation for the data collection facility and the resulting direct link between the analysis programs and the simulation model. The ultimate effect of pursuing that objective, however, was to make the model inappropriate for an investigation of the allocation techniques suggested in Chapters II and III. Two problems emerged:

(a) Because the simulated system closely resembles the real system, a host of data characteristics and supervisor algorithms enter the picture which are not covered by our analysis. Variations in performance due to decisions about such matters as CPU scheduling, page posting, size of core storage, and length of time slice can be significant enough to overshadow performance improvement techniques such as advance paging.

(b) The use of MTS data precludes an investigation of the cyclic queue model: adjusting the number of tasks to an optimal value is of no use unless an accompanying change in the page request rate is observed.

These observations about the application of theoretical techniques to a simulation model apply as well to the actual systems: until a number of basic adjustments have been made to a system and it operates in a reasonably efficient manner, there is no point in implementing a sophisticated memory allocation scheme. The hardware configuration must be roughly matched to the system load, with supervisor algorithms providing a better match, and the programs for driving such components as disks and drums must operate well enough so that no artificial imbalance is created. Most large-scale systems require a great many adjustments of this nature, and the problems are thus of considerable practical, if not theoretical, interest. The simulation program in this thesis is accordingly used to investigate some of these simpler controls over storage allocation.

Several algorithms used with the drum I/O channel are examined to see the extent to which they contribute to the efficiency of that component: a read-before-write policy for the sector queues is compared with the first-in, first-out discipline, and the effect of delayed posting of page transfer completions is compared with immediate posting.

Two aspects of the organization of the paging mechanism itself are considered: a 9-sector drum is compared with the same drum formatted for 4 sectors (which gives a capacity of 100 fewer pages at some increase in performance), and the use of a more expensive large-core storage device (with the same transfer rate but no latency) is simulated. The physical size of main storage is another parameter varied for the simulation runs.

One characteristic of the workload is a variable system parameter: the percentage of pages which is left unchanged through a period of residence in main storage. MTS experience suggests that 20-25% of pages read from auxiliary storage need not be written out because they are unchanged. Finally, in an attempt to match the CPU demands as well as possible, several values are tried for the length of a time slice, although this parameter is not varied dynamically during the simulated execution of programs.

Several inadequacies of the simulation model were revealed only during the final runs, whose results are tabulated in Appendix F. In all but the last run an intended variation in the percentage of unchanged pages did not work, with the result that in each case all pages were written out instead of the planned 70-80%. It was also apparent after the runs were over that the initialization periods before data was taken were too long. While this is not normally harmful, it had the effect in this instance of spacing the corresponding segments of different runs less regularly over the common task streams, thus reducing the value of that "parametric" feature.

An attempt to re-interpret the MTS data as characteristic of a heavier load backfired, with the result that the set of simulation runs reported in Appendix F is difficult to compare directly with MTS experience, and shows less than real differences in performance for the two data sets considered. The MTS paging load during the data collection period was usually way below capacity, primarily because the system was new and developing at the time at a rate which kept ahead of the increasing workload. Since a much higher paging load was certain to develop, and it was only then that improved storage allocation techniques would have any worth, the data was recast for use in the model in a manner described in the next paragraph. When a command was typed by an MTS user to initiate a new interaction, his task often already occupied several pages of main storage. The pages of his task in core had not been forced out since his last interaction because there were not enough page requests from other tasks.

In order to use the MTS data and yet simulate a heavier paging load, the number of pages already occupied by an MTS task at the start of an interaction was taken in the GPSS model to be the number of pages which had to be read in before the interaction (=task in the model) could begin. The unintended result of this action was to create an unusually heavy paging load in any MTS data used in the model. If the paging load was already heavy, this assumption had little effect. In anything but a heavy load, the situation modelled was more like swapping than demand paging, because many more pages were brought in initially for a task than were subsequently referenced during execution.

IV. DISCUSSION OF SIMULATION RESULTS

We now examine in more detail the results of applying the MTS data to variations in the simulation model mentioned in the previous section. As we see from the figures in Appendix F, the MTS # 1 data tape[1] produces tasks which under our interpretation require 75-80% of their pages to be read into main storage as a prerequisite to execution. These pages had remained in core in MTS from the previous

----

[1]The data used for these simulation runs was also used for the distributions of MTS task characteristics given in Appendix D.

interaction, and some of them would not have been referenced in the new interaction. Thus the paging load is unusual for this data in two respects:

(a) the number of page requests per task is generally greater than the actual number, and

(b) many of the requests come in large bursts at the beginning of each model task (interaction).

The corresponding "front end load" of page requests with MTS # 0 data is considerably smaller—it ranges from 28-60% and is almost always less than half. (One page is always requested before execution, and since the average number of pages per task is small (about 5-8) and skewed toward the smaller values, this case is a much better representation of the actual MTS situation). The total number of page requests is also significantly less for this data set because some of the pages forced out of core by a heavier load were not needed again.

Each of the figures in Appendix F is organized into two main sections: the upper half shows the performance obtained when the specific task stream was used with the given model. The performance index most useful for comparisons is the fraction of CPU utilization, since it is not sensitive to variations in the makeup of individual tasks. The other item of particular interest here is the performance of the drum or large core storage (LCS) unit used for paging.

In order to make comparisons between different model policies with this data, its workload characteristics are shown for each run. Although the same two task streams are used for all runs, the initialization intervals and different run segments (whose lengths are based on simulated time) use different numbers of tasks from the tape. Over the relatively short real time intervals which the data represents, there are some significant variations in the workload over the various run segments. The mean length of the requested CPU intervals is an direct statistic of the MTS data itself, but most of the other workload items given are normalized by run time, and hence are related to the way the model worked as well.

In all but Figure F-11, which will be discussed later, the number of pages read and written during the run segments were essentially equal. Hence only the latter figure is given. The item entitled "percent initial pages" is to be understood as follows: of the number of pages read, the given percentage were required before execution could begin. Appendix E explains the policy for allocating storage in the GPSS model: a new task enters the system whenever use of core storage drops below a certain threshold. A queue is maintained when core is full, but in order to avoid a "devil's embrace" in which all tasks in the system are in the core queue, the queue has finite storage of a length chosen in conjunction with the maximum pages allowed a task. When the queue is full an arriving task is terminated prematurely.

Data not given in the figures shows that core utilization under the aforementioned policy was always in the range 92-98%, with little variation in the segments of any run. The number of of tasks encountering a full core queue was similarly rather stable, with this number directly proportional to the rate at which tasks were processed by the model. The number of tasks exceeding the maximum page count was negligible. Since any prematurely terminated task had been simulated in part, results from what had been carried out are included in both the performance and workload data.

Comparing Figures F-1 and F-2, which show the same Basic[1] model with each of the two task streams, we see first that the workload for the second case consists of CPU intervals slightly half as long, fewer but longer I/O operations, and more evenly distributed page demands than we have in the first case. With the heavier load, the system seems to be getting more choked up as time goes on, rather than maintaining an equilibrium. However, the actual system was observed to be operating in the same fashion when the data was taken: over the sampling period the load increased significantly. A primary cause for the poorer performance seems to be the increase in duration of I/O operations, for drum congestion is much less than with the first task data.

Similar comments apply to the comparison of Figures F-3 and F-4, which show the Advanced model with the same two task streams. A far greater number of tasks is processed in each of these runs, however, which makes the average data appear more stable. The changes made in going from the Basic to the Advanced model are responsible for the much smaller degradation under increased load. These improvements greatly affect the ability of the system to cope with additional page traffic. In contrast to the difference

----

[1]The variations in simulation run parameters are discussed in Appendix F and briefly in a later paragraph in this section.

between the first two figures, we see here an increase in paging mechanism utilization with increased load, and a smaller decrease in CPU utilization. In all of the first four figures we see that a significant portion of core pages are enqueued for transfer at any time, with the peak queue length exceeding two-thirds of the core capacity.

The Basic system is outperformed 20-25% in CPU utilization for one data set and 30-35% for the second by its Advanced counterpart. This is no surprise, and in fact the difference might have been greater considering the magnitude of the changes in the system:

- The increase in core capacity from 80 to 144 pages, which approximates the increase in available storage in going from a System/360 MTS system with two core boxes to one with three,

- The use of LCS instead of a drum for paging, which increases the effective use of core storage,

- The assumption of immediate page posting, which cannot actually be carried out on the existing hardware without measurably increasing CPU overhead.

In addition, a longer time-slice was used to reduce the number of interruptions in CPU service, and a change in the drum queue discipline improved core utilization, although these effects produce smaller increases in performance than the ones enumerated above.

In Figure F-5 we see the results of a single one of the improvements in the Basic system with the lighter of the two workloads. Adding only the extra core box to the Basic model improves CPU utilization almost to the level of the Advanced system. Though this improvement would undoubtedly be less for the data with heavier paging load, we see here that the CPU has become much more efficient than it was with the Basic system. The larger core allows more tasks to be active at one time, so that performance is less affected by longer paging delays.

Figure F-6 shows the result of posting page transfer completions for the 9-sector drum immediately after each transmission rather than once per revolution. Though again performance "goes downhill" as the simulation progresses, it holds up better under this increasing load than the Basic system. This effect is much less significant than that of increasing core size, and less than was expected. Figure F-7 shows that using LCS is an improvement of yet smaller magnitude. One would expect the removal of the delay after transmission (before page posting) would create about the same improvement as removing the latency which occurs before a page is reached (by the use of LCS). While a variation in the workload may have caused some of the apparent difference between these two factors, the following characteristic of the drum is also significant here: because the drum has a number of sector queues which provide service independently of one another, its response to a burst of requests moderates the queueing delays, whereas in the case of LCS all the page requests pile into the same queue. As the drum load builds up, pages are delivered at a rate approaching that of LCS, and the some of the advantage of the more expensive device is lost.

Figure F-8 illustrates the use of a preemptive priority scheduling scheme of the type used in MTS (it is described fully in Appendix A). Each time a task completes an I/O operation it preempts the task currently using the CPU. In almost every respect, the results of this test are the same as those of the Basic system. The mean CPU service time is shorter owing to the interruptions, and the paging activity is slightly increased. The overall CPU utilization is slightly smaller. This policy exists in MTS in part to optimize the SPOOLing operations which run in the background and generate a large number of interrupts for short CPU service requests. The results of this run suggest that it does nothing to help overall CPU utilization for conversational tasks, and indeed seems to degrade performance slightly by increasing the paging load.

In Figure F-9 we give the results of a "negative" test: a poorer method of operating the paging drum is examined for the lighter paging load task stream to judge the sensitivity of the mechanism to such changes. When write requests are allowed to compete with read requests by being assigned at random to drum sectors and given equal priority in the queue discipline with read requests, the result is an expected decrease in performance, though it is no worse than expected.

A number of changes in the Basic model are lumped together in the run described in Figure F-10: the drum queues are ordered by giving priority to tasks which already have a large number of core pages (in an effort to complete these tasks in less elapsed time). The option of a preemptive CPU queue discipline is also included here. Finally, the time slice is increased from 16 to 24 time units, or for MTS about 64 to 96 msec., in order to reduce paging. Ccmparing the results with the corresponding Basic run, we see that they are inconclusive: there is an improvement in the first run segment which is at least partially cancelled in the second. Differences are probably at or below the level of variations in the task data. We saw earlier that the preemptive CPU queue option had a negative effect on CPU utilization. From MTS experience and simulation test runs, it seems that the change in time slice would produce a very small improvement in performance. Thus it also seems that the change in drum queue discipline has made little if any improvement.

As a final case, we consider the use of a 4-sector drum with the results given in Figure F-11, which reduces latency over the 9-sector drum by a factor of more than two, and also the effect of writing 20% less pages than are read (assuming that they are left unchanged since the last read operation). In this case we see a substantial improvement in CPU utilization, and a less significant increase in the use of the drum. Both of these changes improve performance. In view of our earlier experience with latency and the heavy paging load in the present case, we suspect that more of the credit for performance improvement belongs to the fact that 10% more drum operations are available for read operations than to the decrease in individual paging delays.

CHAPTER 5. CONCLUSIONS

"The purpose of computing is insight, not numbers."

R. W. Hamming

This study has concentrated on increasing the efficiency of operating systems by improving storage allocation techniques. The performance of modern computing systems is sensitive to a number of distinct but related parameters: the hardware configuration, the operating system algorithms which are used to support the machine, the structure and constraints under which tasks must be presented to the system, and the characteristics of the demands for service all have an important influence on the efficiency with which a system processes tasks. As systems grow larger and faster, doing more multiplexing and parallel processing, information about their performance and workload is harder to obtain and interpret, and theoretical models of system and task behavior are harder to formulate. Heuristic techniques for improving performance are also less successful because systems are poorly understood.

The central purpose of this research has been to identify ways in which the storage allocation aspect of these problems can be understood. A prerequisite to this knowledge is a method of learning exactly what goes on in a system that is of importance in affecting storage allocation events. This problem was solved for the purposes of the current research by designing and implementing a system for data collection and analysis for a new time-sharing system, and demonstrating that the facility could obtain the necessary data with sufficient accuracy. This tool has also proved useful for studying a number of other previously unmeasured phenomena, and for use in evaluating experiments performed with the real system.

A second problem in the investigation of storage allocation techniques for large-scale systems is that refinements in the scheduler and other central operating system components have little impact if the system is poorly configured for its workload or is supported by inadequate programs for management of the important auxiliary storage devices, such as disks and drums. Thus a number of equally difficult problems must be solved before the system can be "tuned up" to its potential. This problem was approached by using a simulation program to estimate the impact of some of these basic configuration and device management problems. Data from the system itself was invaluable in the design stage for the simulation model, but even more so as a source of input data for simulation runs. Not only does the use of an abstraction of real data preserve a number of subtle correlations and interrelationships among the different events and characteristics of system operation, but the simulation results can be compared more directly with the performance of the real system under the same conditions. The same data can be used repeatedly to test different simulated systems under a constant load.

One of the most intriguing possibilities for improving system storage allocation is the study of the way individual tasks can be modified in structure and arrangement in storage in order to meet this goal. In particular, one would like to automate the process of producing large programs which use storage efficiently and act in harmony with the operating system algorithms and policies. The first two chapters of this thesis indicate ways in which stochastic descriptions of the behavior of tasks can be used to minimize the system cost of processing them. Both the static and dynamic techniques can be automated by using a data collection facility and appropriate analysis programs to reduce the data to a concise stochastic model of task behavior.

Models of individual tasks are insufficient, however, to encompass the entire storage allocation problem for multiprogrammed systems which are designed to capitalize on variations in individual task behavior. In these systems there are additional problems involved with scheduling events whose contribution to system cost can only be measured in terms of the immediate environment or events which follow it in time. Thus the third chapter of this work deals with a simple model which represents the multiprogramming situation. Here, as with the other models, we try to represent the actual storage allocation structure as accurately as possible. The intent is not to decide whether a particular allocation scheme or hardware configuration is better than others, but to provide a tool which can be used to compare strategies in a specific situation. The multiprogram model is also amenable to automation, in the sense that a range of parameter values appropriate for the anticipated load can be stored in the system and used to help control the flow of tasks.

A word should also be said about the relation of this work to questions of response time and other descriptions of satisfactory system performance. It is frequently true that system efficiency and the maintenance of good service run counter to one another. We justify our disregard for this other aspect of system performance on two grounds: first, that a certain level of efficiency is required before these questions come into conflict. A very inefficient system serves no one well. Second, most questions of adequate response allow the system a certain amount of leeway in its control of tasks. Typically, a system must provide several levels of service, but no further constraints are given as long as these levels are maintained. Whenever such leeway occurs, the system should capitalize on the freedom in order to increase its own efficiency.

In summary, the results of this work demonstrate the advantages of using a "vertically organized" set of models for studying operating system and individual program performance: from data about an actual system we establish a set of models at different levels of abstraction which are related with the use of common data, and whose results may be implemented to improve the real system used as a basis.

Appendix A: UMMPS and MTS

This appendix describes the time-shared operating system from which operating statistics and job program data were gathered for use elsewhere in this paper.

I. UMMPS

UMMPS (University of Michigan Multi-Programming System) is a multiprogramming operating system for the IBM System/360 series computers. UMMPS executes jobs, which are initiated and controlled from the operator's console typewriter. Each job runs in problem state and uses supervisor calls for all its input and output operations.

A job program is the basic set of instructions which are executed when an UMMPS job is run. Job programs are core-resident along with the UMMPS supervisor and subroutines. A reentrant job program can be executed at the same time by more than one job. When a job program is written a set of device types and a set of memory buffers of various sizes are specified. Corresponding actual devices and memory space are allocated for any job initiated with that job program, and these are retained until the termination of the job. By means of supervisor calls, jobs may obtain and release additional devices and storage space during their execution. A single device (e.g. a card reader, communications terminal, or a disk module) is available for at most one job at any given instant.

The more recent (since November, 1967) versions of UMMPS use the dynamic relocation hardware peculiar to the System/360 Model 67 in order to provide a virtual memory buffer space of 256 pages (one page = 4096 bytes) for each job. The supervisor manages real core memory with a demand paging algorithm, using an IBM 2301 Drum for secondary storage. The drum format is described in Appendix B, as well as the queueing structure used to prepare program pages for reading and writing. The policies for using the queues are as follows:

(a) A channel program is constructed to read pages from as many of the drum sectors as possible. The remaining sectors, if any, are then used for servicing the first write requests. More than one of these channel programs may be constructed at one time and chained together, providing a complete schedule of drum operations for the next several revolutions of the drum.

(b) Unless a page in real core enjoys a special "temporarily resident" status, it is placed on the queue for page-out as soon as it enters core. This queue is reordered according to the usage of the pages enqueued: once every several hundred times that the supervisor is entered, the hardware "storage reference bits" are checked for each page in the queue, and all pages referenced since the last check are moved to the end of the queue.

(c) When the paging drum processor requests a set of pages from the supervisor to be written on the drum, those pages are made available from the head of the page-out queue only in case core is sufficiently full.

(d) The supervisor may also refuse to supply the drum processor with an empty page for a page-in request, if there are almost no available empty core pages.

The organization of the job scheduler information is diagrammed in Figure A-1. The main CPU queue is a list of jobs which are scanned in top to bottom order, and are given the CPU if they are ready to use it. Whenever a job initiates an I/O operation which makes it not ready for execution and guarantees that an interrupt will occur when it again becomes ready, its entry is removed from the main CPU queue. UMMPS also allows a job to remain ineligible for the CPU until a byte in main storage changes value. A job waiting for such an event remains in the CPU queue, and the byte is tested only when that job is the next one to be dispatched. If the byte has not yet changed, the job following it in the queue is given the CPU. (A job in execution can also voluntarily place itself at the bottom of the CPU queue.)

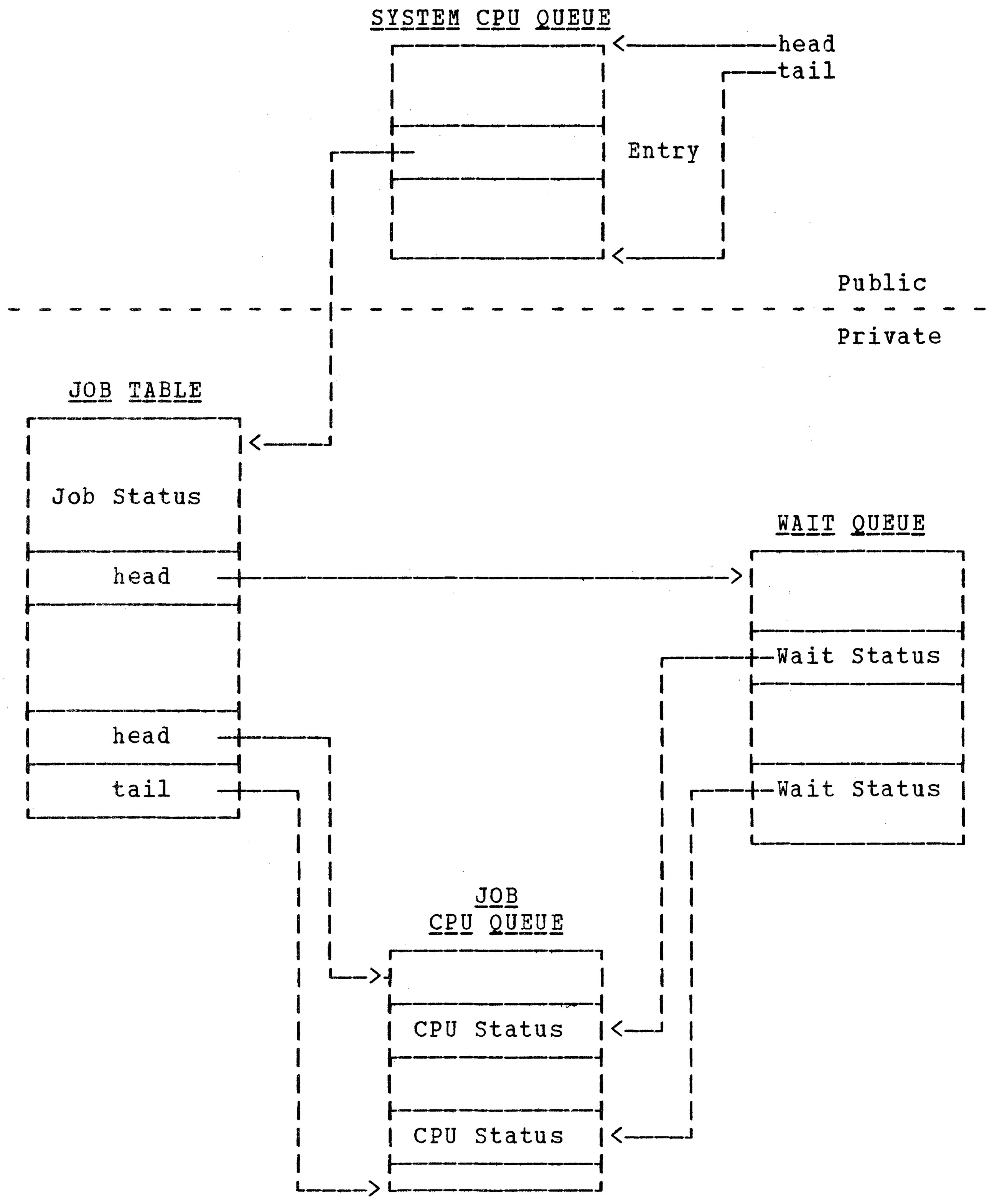

Figure A-1. The UMMPS Job Scheduler Queues.

When an interrupt occurs for a job which does not have an entry on the main CPU queue, it immediately preempts the job currently allocated the CPU (i.e. an entry is placed on the very top of the queue and the job is dispatched, while the preempted job remains next in the queue). Each job runs until it waits for some event or until a timer interrupt signals the end of a time slice.

After several months of demand paging operations, the UMMPS job scheduler was modified to improve its performance under heavy paging loads. The new algorithm, suggested in part by the investigation in Chapter 3 of this thesis, allows only a handful of jobs to attempt to acquire a large number of main storage pages at any one time. Another job requesting a real core page which would give it more than a certain threshold is made to wait until one of the "privileged" jobs has finished its time slice or begun an I/O wait. The threshold, which is initially about one quarter of the total main storage space, is lowered each time a job becomes privileged, and raised when a privileged job leaves that status.

The number of privileged jobs is a system parameter. A job entering privileged status at a time when k more jobs may yet do so is given a time slice which is k+2 times as large as the fixed value for unprivileged tasks. As an additional attempt to keep the jobs from fighting among each other for storage space, all main storage pages belonging to a job in page-wait are counted as having been referenced whenever the page-out queue is reordered.

Each job has a personal CPU queue which is used to keep track of multiple levels of execution. The top entry refers to the sub-task (if any) currently in contention for the CPU. A lower-level entry represents a sub-task which has been interrupted but may later be resumed. For example, some I/O interrupts cause a new entry on the queue for their processing. A job using the MTS job program with a remote terminal device may be given an attention interrupt by the MTS user (or operator), and then restarted after other commands, such as those to display and alter storage locations, have been given.

A wait queue is maintained for each job. It contains an entry for each personal CPU queue entry which represents a sub-task currently waiting. Thus a job may be waiting at several of its lower levels and executing at the top level. An interrupt signalling the end of such a wait can be properly recorded by removing a wait queue entry for the appropriate CPU queue level.

II. MTS

MTS (Michigan Terminal System) is a reentrant job program in UMMPS. It provides the capability of loading, executing and controlling programs from remote terminals and through a batch stream. In the rest of this paper, the term MTS job denotes an UMMPS job using the MTS job program. Since an enabled communications line may be used serially by more than one person, the term MTS task is used to refer to an MTS job during its use by a single individual, i.e. from the execution of a $SIGNON command to the corresponding $SIGNOFF. Together with UMMPS, MTS provides a simple but powerful time-shared computer system, whose salient features are these:

(a) Command language. Several dozen commands are available to cause the running and monitoring of programs, the manipulation of line files, and other communication with the system.

(b) Line files. A system of information organized in units of lines (1 to 256 characters) and files (0 to many thousands of lines) is provided for the storage of programs and data. A file may be public or private, and a private file may be permanent or temporary. These files reside on direct-access storage devices.

(c) Logical devices. When an MTS user specifies the origin or disposition of data, he may give, interchangeably, the name of a file location or a physical device. A logical device name or number is then attached to it. It may refer, for example, to a system (public) file, a new temporary private file, a card punch, or the operator's console.

(d) Dynamic loader. A program to dynamically load programs is an integral part of MTS. It may be invoked by both commands and subroutine calls.

(e) Libraries. External symbols which have been referred to by a set of loaded programs, but not defined, may be resolved by reference to a private or system library, which is a file containing object programs in a special format. Facilities exist in MTS and the Loader to pass over a library and selectively load only needed subroutines (and the subroutines that they need, etc.).

(f) Language processors. The MTS system makes available the IBM System/360 F-Level Assembler, the IBM FORTRAN IV G-Level Compiler, WATFOR (University of Waterloo FORTRAN "Load and Go" Compiler), PIL (University of Pittsburgh Interpretive Language), SNOBOL4 (Bell Laboratories string manipulation language), and UMIST (University of Michigan Interpretive String Translator), a string processor based on the TRAC text-processing language. These programs reside in system files, and are executed in the same way as one's own programs produced by their execution. Other powerful system features, such as the IOH/360 input-output conversion subroutines, macro libraries, plotting routines, etc., reside in the library and other system files.

Appendix B: The Drum I/O Channel

This appendix describes a type of input-output channel with an attached magnetic drum storage unit. An organization is specified for using this "drum I/O channel" as an extension of main storage in a paging system. For a particular set of parameter values, this section describes the IBM 2301 Drum with IBM 2820 Storage Control Unit. The organization described is in fact used with this drum in the IBM Time-Sharing System/360 (TSS) and Michigan Terminal System (MTS) operating systems.

I. Equipment and Data Format

We assume that both input and output data transfers must be processed sequentially through a single channel, with each transfer consisting of one _page_ of information. For this purpose the drum is formatted laterally into a number, k , of _tracks_, with each track divided on the circumference of the drum into m _sectors_ of equal size. The part of a track lying within one sector has a capacity of one page of information. Thus each of the m times k pages stored on the drum has a unique identifying address consisting of its (track, sector) pair. As a new sector of the drum reaches the read/write heads, the choice of a track for the next transfer request is made electronically (with essentially no delay), so that any of k pages may be selected regardless of the track used for the previous read or write.

We note, parenthetically, that on the IBM 2301 Drum four physical tracks may be read in parallel; hence they form together one "track" in the sense of the preceding paragraph. Furthermore, with the given page size of 4096 bytes, one "track" is divided into four and one-half sectors. Thus 9 pages are stored on a set of two adjacent "tracks" (eight physical tracks), and two drum revolutions are necessary in order to access the entire set. Used in this manner, the 2301 drum has 100 _logical tracks_, providing a total capacity of 900 pages (m=9, k=100).

II. Data Access Organization

A page read or write request for the drum I/O channel is placed on one of m sector queues. Since a read request involves a page which already exists on the drum, it is constrained to a specific sector. A write request, on the other hand, may be assigned to any sector (releasing a previous drum address, if any). A channel program for data transfer during one revolution past the m sectors is constructed by removing the first entry in each sector queue. The paging drum processor constructs each channel program and links it to the "command chain" of channel programs already enqueued for execution. Thus the queues are serviced cyclically as the appropriate sectors reach the read/write heads.

III. Analysis

We adopt a time scale in which one unit of time is the transfer time for one page of information, or almost equivalently, the rotation time of the drum divided by the number of sectors (for the IBM 2301 Drum this time is about 3.9 milliseconds [35]). We now compute the cumulative distribution function for the drum service completion time (queueing time plus unit service time), assuming that

(a) page transfer requests are uniformly distributed over the m drum sectors, and

(b) the sector queues are serviced with a first-in, first-out discipline.

Lemma B-1.

The completion time distribution function F(k,t) given that the sector associated with the request is known and k requests precede the arriving one on that sector queue is

$$F(k,t) = \mathrm{Prob}[c \le t \mid k] =$$

$$\begin{aligned} &(a) \quad 0 \quad \text{if } t < km + 1, \\ &(b) \quad 1 \quad \text{if } t > (k+1)m + 1, \\ &(c) \quad (t-km-1)/m \quad \text{otherwise.} \end{aligned}$$

Proof: The minimum time required to service the given $k+1$ requests is k drum revolutions plus one time unit, or $km+1$ time units. The arrival of the last request is arbitrary with respect to the rotational position of the drum, and the maximum time for servicing $k+1$ requests is one additional drum revolution, or $(k+1)m+1$ time units. The completion time of the $(k+1)$st request is uniformly distributed between the minimum and maximum values.

Lemma B-2.

The probability $G(n,k)$ that a given sector queue contains exactly k entries, given that n transfer requests are enqueued in the entire system, is

$$G(n,k) = C(n,k)\,(m-1)^{n-k}/m^{n}$$

where $C(n,k)$ denotes the binomial coefficient.

Proof: The k sector queue entries can be chosen from n in $C(n,k)$ different ways, and the remaining $n-k$ requests can be found on the other $m-1$ queues in

$$(m-1)^{n-k}$$

ways. The total number of distinct configurations is of course

$$m^{n}$$

Since the given sector queue must contain k entries, for some value of k between 0 and n, we have also that

$$\sum_{k=0}^{n} G(n,k) = 1$$

Theorem B-1.

$$G(n,k) = C(n,k)\,(m-1)^{n-k}/m^{n}$$

where $C(n,k)$ denotes the binomial coefficient.

Proof: The k sector queue entries can be chosen from n in $C(n,k)$ different ways, and the remaining $n-k$ requests can be found on the other $m-1$ queues in

$$(m-1)^{n-k}$$

ways. The total number of distinct configurations is of course

$$m^{n}$$

Since the given sector queue must contain k entries, for some value of k between 0 and n, we have also that

$$\sum_{k=0}^{n} G(n,k) = 1$$

Theorem B-1.

The unconditional distribution function $H(n,t)$ for the completion time, given n prior requests to the drum I/O channel, is

$$H(n,t) = \text{Prob}[c \le t] = \sum_{k=0}^{n} G(n,k)\ F(k,t)$$

Proof: An arriving request is assigned at random to a sector queue in which k requests are already enqueued with probability $G(n,k)$, and n requests are already waiting in the whole system. Its completion time in this case is F(k,t), where k may take on any value between 0 and n.

The drum I/O channel completion time distribution function is plotted in Figure B-1 for small values of n. Because its algebraic form is simple, the first and second moments can be obtained in closed form:

Theorem B-2.

The mean value M(n) of H(n,t) is given by

$$(m+2n+2)/2$$

Proof: By definition, M(n) is the total integral of t with respect to the differential of H(n,t). We apply the definition of H(n,t), differentiate and integrate under the summation operation, and split the resulting integrals each into three sections, according to the form of the functions F(k,t). The identity in the proof of Lemma B-2 is then applied to each of the final terms.

$$M(n) = \int_0^\infty t \, dH(n,t)$$

$$= \int_0^\infty t \, d \sum_{k=0}^{n} G(n,k) F(k,t)$$

$$= \sum_{k=0}^{n} G(n,k) \int_0^{(n+1)m+1} t \, dF(k,t)$$

$$= \sum_{k=0}^{n} G(n,k) \left[\int_0^{km+1} 0 \, dt + \int_{km+1}^{(k+1)m+1} t/m \, dt + \int_{(k+1)m+1}^{\infty} 0 \, dt \right]$$

$$= \sum_{k=0}^{n} G(n,k)((2k+1)m + 2)/2$$

$$= m \sum_{k=0}^{n} k\, G(n,k) + m/2 + 1$$

$$= n \sum_{k=0}^{n-1} G(n-1,k) + m/2 + 1$$

$$= (m + 2n + 2)/2$$

The same method of proof establishes the formula below for the second moment of the distribution, with which the standard deviation and coefficient of variation can easily be calculated.

Corollary B-1.

The second moment of the drum delay distribution is

$$m^2/3 + m + 2mn + n^2 + n + 1$$

Corollary B-2.

The standard deviation of the same distribution is the square root of

$$(m^2 + 12(m-1)n)/12$$

The graphs of Figure B-1 show that the drum delay distribution has the general slope of a negative exponential distribution. However, Figure B-2, which compares a few of the same curves with exponentials of the same mean values, shows that the exponential itself is not a particularly good fit.

Since the standard deviation of $H(n,t)$ is less than its mean, a much better approximating function of the exponential type is a gamma distribution [26] with integral parameter, sometimes called an Erlangian distribution. The

Erlangian distribution is selected so that it has the same mean value as H(n,t), and the standard deviations agree as closely as possible.

Figure B-3 gives the basic parameters of some of the functions H(n,t) together with the parameter q of the best Erlangian function, whose definition is

$$E(q,x) = \int_0^x (q/M(n))^q \, (t^{q-1}/(q-1)!) \, e^{-qt/M(n)} \, dt$$

Finally, several of the drum delay distributions are plotted with their corresponding Erlangian approximations in Figure B-4. It is interesting to note that for small values of n, the best Erlangian fit has a small, relatively constant parameter.

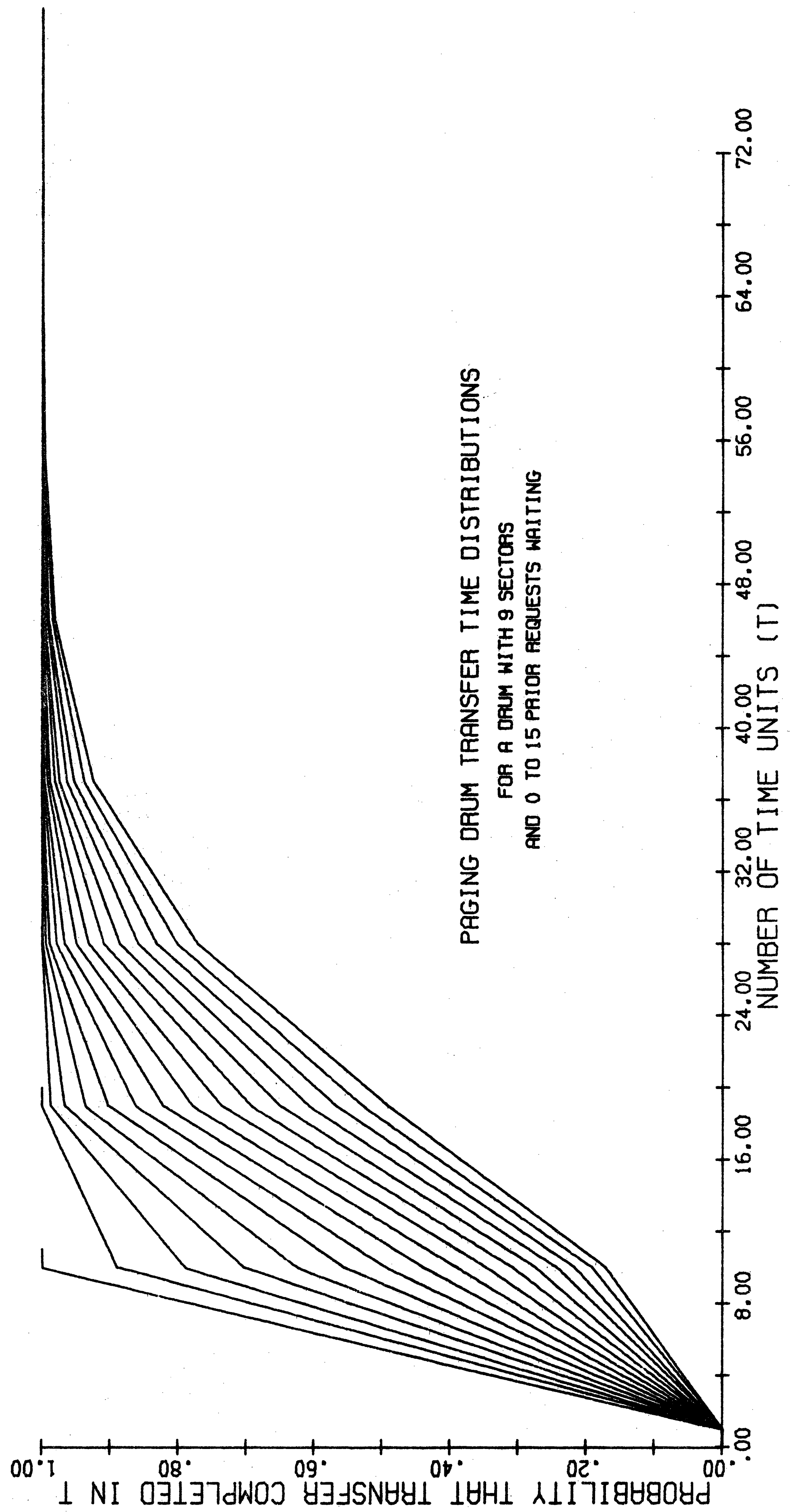

Figure B-1. The Functions H(n,t)

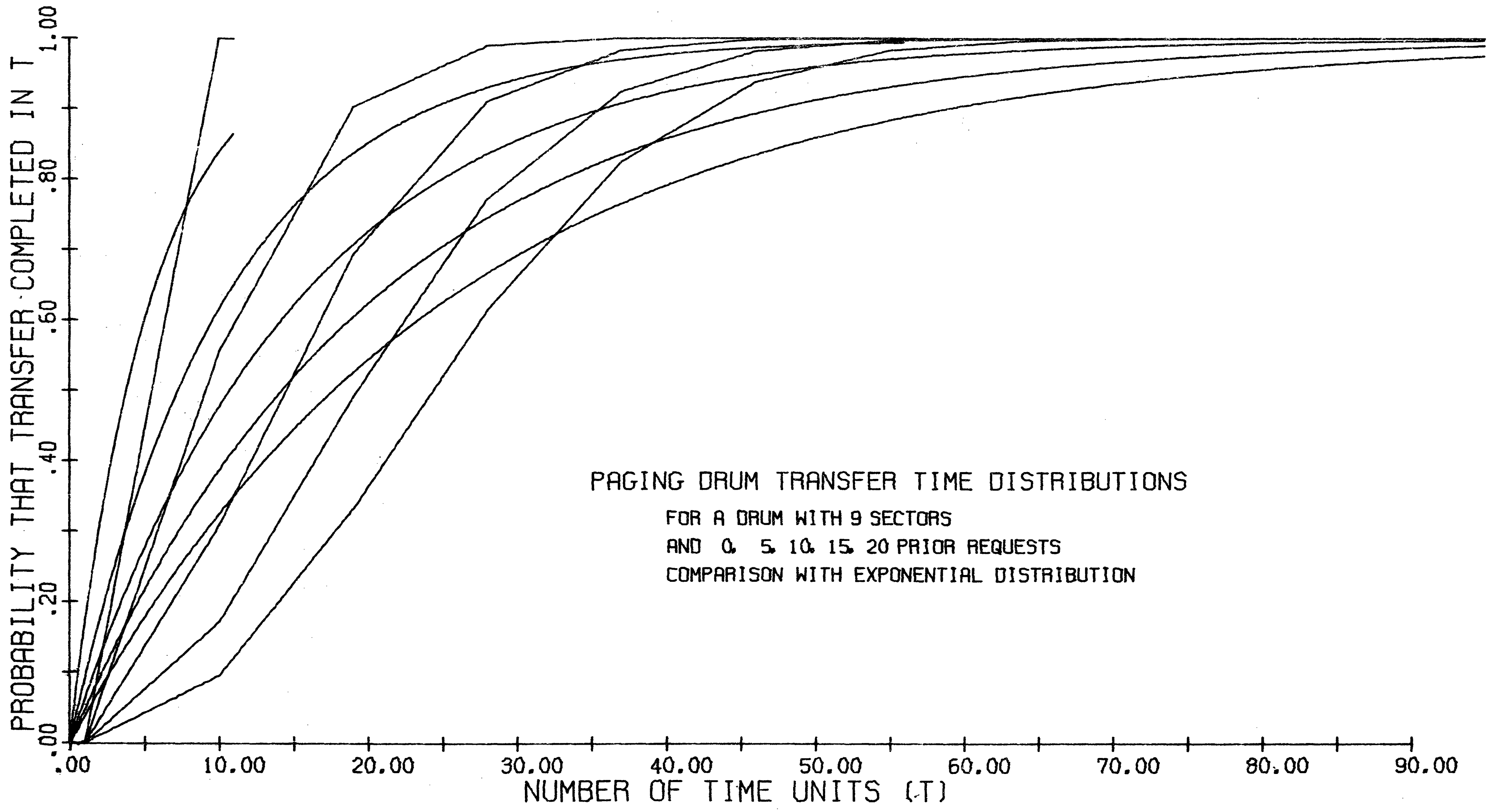

Figure B-2. The Negative Exponential Fit

Number of Sectors	Congestion index n	Mean Value	Standard Deviation	Coeff. of Variation	Erlang parameter
m=9	0	5.5	2.598	47.23	4
	1	6.5	3.841	59.09	3
	2	7.5	4.770	63.60	2
	3	8.5	5.545	65.23	2
	4	9.5	6.225	65.52	2
	5	10.5	6.837	65.12	2
	6	11.5	7.399	64.34	2
	7	12.5	7.921	63.37	2
	8	13.5	8.411	62.31	3
	9	14.5	8.874	61.20	3
	10	15.5	9.314	60.09	3
	12	17.5	10.14	57.92	3
	15	20.5	11.26	54.92	3
	20	25.5	12.91	50.64	4
	50	55.5	20.17	36.34	8
m=4	0	3.0	1.155	38.49	7
	1	4.0	2.082	52.04	4
	2	5.0	2.708	54.16	3
	3	6.0	3.215	53.58	3
	4	7.0	3.651	52.16	4
	5	8.0	4.041	50.52	4
	6	9.0	4.397	48.86	4
	7	10.0	4.726	47.25	4
	8	11.0	5.033	45.76	5
	9	12.0	5.323	44.36	5
	10	13.0	5.598	43.06	5
	12	15.0	6.11	40.73	6
	15	18.0	6.807	37.82	7
	20	23.0	7.832	34.05	9
	50	53.0	12.30	23.21	19

Figure B-3. Characteristics of the Distribution H(n,t)

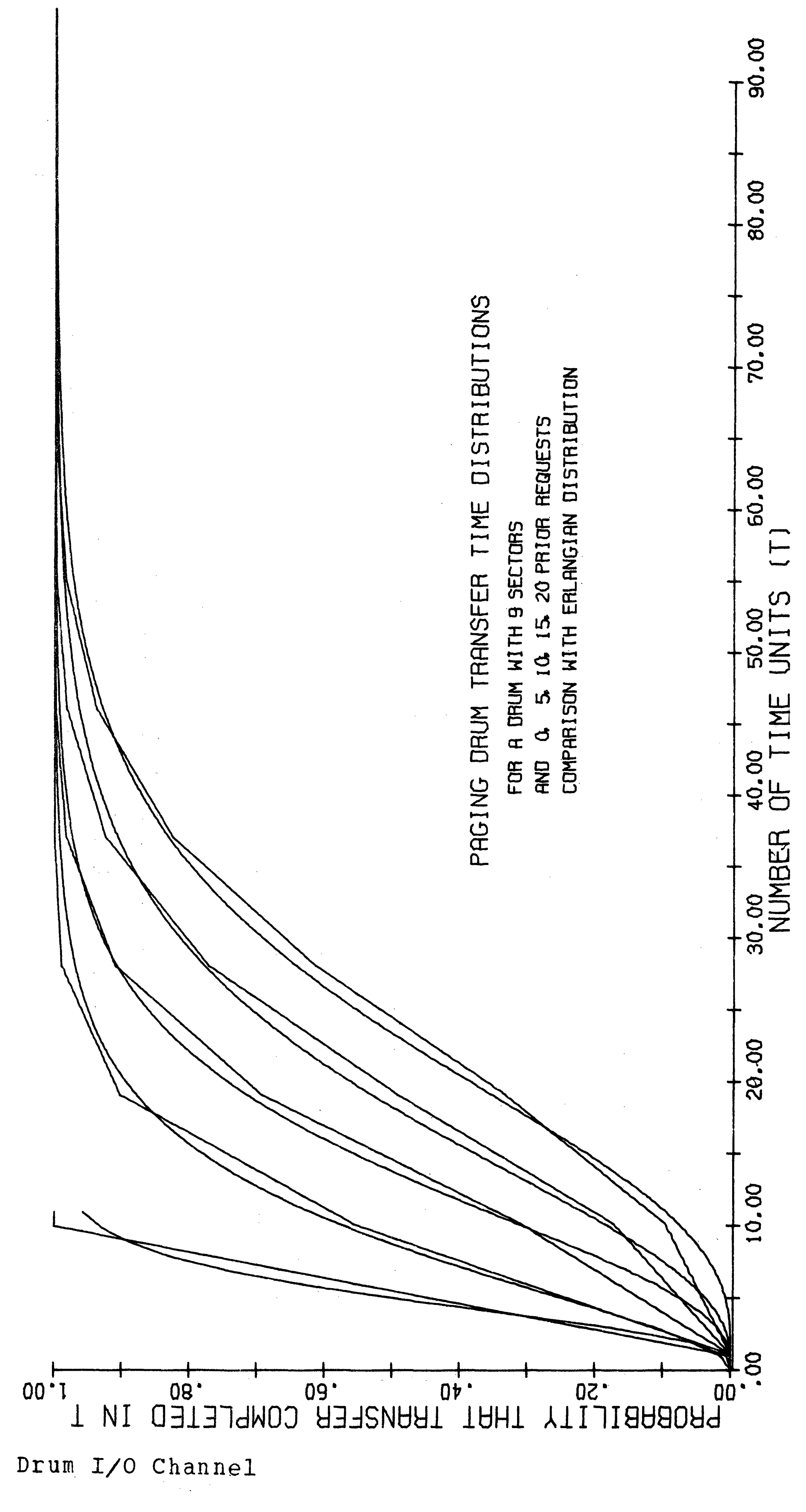

Figure B-4. The Erlangian Fit E(q,t)

IV. Multiple Drums

We note here that a simple modification of the function $H(n,t)$ is all that is needed to represent a configuration of several independent paging mechanisms. For example, with two drums on separate channels, we assume that requests for pages are equally distributed between the two, which leads to

Lemma B-3.

The probability $G^2(n,k)$ that a given sector queue contains exactly k entries, given that n transfer requests are enqueued in the entire system, is

$$G^2(n,k) = C(n,k)\,(2m-1)^{n-k}/(2m)^{n}$$

where $C(n,k)$ denotes the binomial coefficient.

Proof: The proof of Lemma B-2 applies when the page requests are uniformly distributed over $2m$ sector queues instead of m. ▪

Theorem B-3.

The unconditional distribution function $H^2(n,t)$ for the completion time, given n prior requests to the pair of drum I/O channels, is

$$H^2(n,t) = \mathrm{Prob}[c \le t] = \sum_{k=0}^{n} G^2(n,k)\; F(k,t)$$

Proof: Exactly the same as the proof of Theorem B-1.

V. Practical Considerations

There are several ways in which the management of the drum I/O channel by an operating system tends to alter its performance from the functional characterization given in the previous section:

(a) If a channel program is constructed and added to the command chain too far in advance of its execution, there may be sectors for which no read or write requests are available. Yet such a request arriving once the program is enchained cannot be serviced until a subsequent revolution. This phenomenon tends to make the drum I/O channel respond as if it were subject to a heavier load.

(b) The real core page involved in a transfer request cannot be used until the completed transfer is _posted_ (a table is updated to record the transfer). Programmed interrupts are usually inserted in the command chain to trigger page posting, but since a noticeable amount of CPU time is required to process such interrupts, one can afford to use them only to signal the completion of groups--say 5 to 10--of transfers. Thus the effective service time is increased by the fact that a typical page is unavailable for a time after the actual completion of its associated transfer.

(c) In practice a queue discipline other than first-in, first-out might be used with the drum sector queues. If, for example, there are priority classes of page transfer requests, then the drum delay distribution is applicable only to the highest priority class, where for the number n of prior requests waiting we use only the number of highest priority requests waiting. A reasonable priority scheme, for example, is to give all read requests priority over every write request in the sector queues. This is done in the MTS system.

Appendix C: The Data Collection Facility

This appendix describes additions that were made to UMMPS and MTS in order to record details about jobs executed in the system, and the way in which they were processed. The effect on the data of measuring it is considered, and shown to be so small as to be negligible. Also described are programs written for the analysis and reduction of the data obtained in this fashion.

I. Data collection

The data collection facility includes two job programs added to the UMMPS system (STAT, STATSW), a subroutine available for execution at the MTS device support routine interface, a supervisor call (SVC) instruction, and a number of additions and modifications to UMMPS and MTS to use the new programs: data is taken from those points in the system where relevant information is available about a job being run.

(a) STAT Job

An UMMPS job called STAT exists to dispose of data being collected by the system. STAT links and manages a chain of one-page buffers into which the supervisor places data. This job is dormant except when a buffer becomes full, whereupon STAT empties it onto tape and marks its availability again. One or two tape units may be used, with reel-switching when necessary.

Whether data is collected or not for a particular UMMPS job depends on the condition of a word of switches in the job table. Provision has been made for each type of data item to be collected or not for each job independently of the other items, but currently either all the items are collected or none. A data item is placed in the current STAT buffer by the supervisor or by executing the previously mentioned SVC, which obeys only in case the job table bit is set for that job. Parameters for the STAT job therefore not only include the names of tape units, but also job (and eventually item) identification numbers as well, specifying what data is to be collected. The STAT job sets the appropriate job table bits, and resets them when data collection is terminated.

The CPU idle condition is handled in UMMPS by pretending to execute a (non-existent) job whose number is zero. Specifying this job number for data collection will cause recording of all transitions to and from the CPU idle state.

(b) STATSW Job

If data collection is to be initiated or terminated for a job once the STAT job has already been activated, the STATSW job is used. This job simply turns specified job table bits on and off.

(c) SVC STATENT

The supervisor call for entering a data item in a STAT buffer accepts up to 24 bytes of data, and prefixes it with a standard 8-byte unit which contains

1. one byte of item identification and length
2. a two-byte job number
3. a five-byte timer value

Separate identification codes for 32 distinct items are provided. Most of these codes have been assigned meanings by their use in UMMPS and MTS for specific kinds of system data. The remaining codes may be used in job programs by system programmers as an aid in error analysis. Figure C-1 summarizes the definitions of the existing codes.

STANDARD DATA ITEMS COLLECTED WITH THE STAT JOB

Parts of the data items which are not described below are unused or contain meaningless data. Each standard item begins with a two-word prefix: the ID and length in byte 1 in the form ID*8+LEN-1, then the low order timer byte in byte 2. The job number occupies bytes 3-4 and the timer word is in bytes 5-8.

Note: The first two items are placed in the buffers by STAT itself and do not have the standard prefix described above.

NAME	ID/LEN	DESCRIPTION
Overflow*	0:1	The second half-word of this one word item contains a count of the number of items which were missed at the point of occurrence because the STAT job could not keep up.
Date*	1:3	Words two and three of this item contain the EBCD date obtained from the system and placed in the first buffer by the STAT job.
Adtotp	2:3	This item occurs when a new entry is added to the top of the CPU Q for this job. Byte 9 contains the index of the new CPU Q entry, and bytes 10-12 contain its address.
Popq	3:3	This item occurs whenever an entry is removed from the top of the CPU Q for this job. Byte 9 has the index and bytes 10-12 the address of the new top of Q entry, as above.
Wayt	4:4	A wayt item occurs when a job enters wait state at its top CPU Q level for any reason. Byte 9 contains the index of the next lower wayt Q entry, and byte 10 the index of the CPU Q entry corresponding to the new Wayt. Bytes 11-12 contain the hex value 00ff if the wait was not for I/O, otherwise they contain the device address. Bytes 13-16 contain the flag and address specifying the location of a wait byte.
Unwayt	5:3	Whenever a job stops waiting for any event at any CPU Q level, the index of the top remaining Wayt Q entry is given in byte 9, and the address in bytes 10-12.

Figure C-1. Standard System Data Items

NAME	ID/LEN	DESCRIPTION
Q	6:3	This type of item is recorded whenever the job given by the number in bytes 3-4 relinquishes the CPU to the job whose number is in bytes 11-12.
Statsw	7:3	The job number given in bytes 11-12 is that of a job whose status with respect to data recording has just changed. Recording has just begun if byte 9 is zero and has just ended if byte 9 is FF.
Paginstr	7:5	When a page-in operation is started the following is given: the real core page address in bytes 8-9, the virtual memory page address in bytes 10-11, the page control block status bits in byte 12, the storage key and other bits in byte 16, the PDP and address flags in byte 17, and the external (track, slot) address in bytes 18-19.
Pagindon	9:5	When a page-in operation is completed the same data is given as for 'Paginstr' above.
Pagoutst	10:5	When a page-out operation is initiated the same data is given as for 'Paginstr' above.
Pagoutdn	11:5	When a page-out operation is completed the same data is given as for 'Paginstr' above.
Pagreclm	12:5	If a page is reclaimed during page-out the very same data is given as for 'Paginstr' above.
Getvmpag	13:5	When a new virtual memory page is allocated the same data is given as for 'Paginstr' above.
Frevmpag	14:5	When a virtual memory page is released the very same data is given as for 'Paginstr' above.
Mark	23:?	This entry is reserved for the use of system programmers in that it is the only one guaranteed to be unassigned to some standard system function, and is 'watched for' by the *ANALYSIS program so that it appears with interval timing on the output format, and is appropriately marked on input format.
Vmpages	24:5	Whenever the number of virtual memory pages used by a job either increases or decreases, an entry appears to give the current value of the space-time integral in 300ths of a second times half-pages in bytes 9-12

Figure C-1. Standard System Data Items (cont'd)

NAME	ID/LEN	DESCRIPTION
		and the time of day when the value last changed in bytes 13-16, with the current (new) number of half pages in bytes 17-20. Note that this is a virtual, not real, storage usage integral.
Waitfor	25:2	A minimal entry is made whenever an MTS user signs off, leaving the job for someone else.
Unload	26:7	When this type of item appears a program has just been unloaded in MTS. Its name is given in bytes 9-24 and the storage index number corresponding to it is in byte 25.
Load	27:7	The information provided above for an 'Unload' is also given for every 'Load'.
Freespac	28:3	When core space is released by an MTS job the storage index number is given in byte 9 and the number of bytes released is given in bytes 10-12.
Getspace	29:3	The same information is given whenever core space is requested by an MTS job.
Dsrin	30:2	When a device support routine is entered the minimum two-word item is given for an input line, and for an output line the following: bytes 9-12 contain the file or device name. Byte 13 contains the current prefix character. The first byte of the FDUB (including a bit for input or output) is given in byte 14. Bytes 15-16 contain the length of the I/O message, and bytes 17-20 contain the first four characters. This information is currently collected only for I/O for devices (not files), and not for lines with a prefix character of . (indicating loading).
Dsrout	31:5	When a device support routine is exited, the minimum entry is given for an output line, and the item which is described above for output lines at Dsrin is given at 'Dsrout' for input lines.
Dsrout	31:7	If an input line begins with the characters $SIG then two additional words (8 characters) of the line are given in bytes 21-28.

Figure C-1. Standard System Data Items (cont'd)

II. Effects of Data Collection

A software method of sampling operating system data itself requires system resources, hence it is necessary to take a hard look at possible bias in the data introduced by the concurrent use of the CPU, I/O channel, and core storage for the data collection activity itself. Happily, the interference in this case can be shown to be negligible.

Data collection overhead occurs in five situations:

- CPU time for deciding if data is to be collected.
- Core storage for data buffers.
- CPU time to enter data in buffers.
- I/O channel time to write buffers to tape.
- CPU time for buffer management.

The first two of these items can be quickly dealt with. By counting mean instruction times, it has been determined that an average of less than 10 microseconds per supervisor entry is required to decide whether or not to collect one or more data items. This time is spent whenever the system is in operation. Core storage consisting of three pages is required for data buffers whenever data is being collected. In the pre-paging versions of the supervisor and MTS, this represented over three per cent of the total storage capacity for MTS jobs, since a maximum of 97 pages was available for non-resident programs. However, with the advent of a virtual memory space for each MTS job, the effect of reserving those three pages of real memory is only to increase the paging load on the system by a maximum of 3%.

Data is placed in the buffers in two essentially different ways:

1. by the supervisor use of a buffer entry subroutine during supervisor activity for some other purpose, e.g. for interrupt processing

2. by an SVC instruction issued by MTS or any job program, which causes an entry into the supervisor especially for data collection

In the former case the additional time required is only the execution time of the buffer entry subroutine. In a typical data collection period over 90% of the items are entered in this manner. The CPU time required by this routine was estimated by summing the average execution times for its instructions. Four cases are distinguished, with their corresponding execution times shown in Figure C-3.

- Items consisting only of the standard 8-byte ID, length, job number and timer value, all supplied by the subroutine itself.

- Items including one to 24 additional bytes of data peculiar to the item type and passed to the subroutine from the point of call.

- Items which cannot fit in the current buffer, and therefore require buffer switching in addition to the normal entry execution time.

- Items which cannot be written because no buffer is available. These are called overflow items. When the STAT job cannot keep up, a count is kept of the total number of items missed before the next new buffer becomes available, and the count is entered as the first item in the new buffer.

Figure C-2 shows a typical distribution of data among the various item types. The first count indicates the total number of items of each type which appear in that portion of the data which was examined. In practice, over 60% of the items are the three-word queue items, and most other items do contain additional data words. Since the buffers hold an average of 310 items each, the additional buffer-switching time occurs only infrequently. And finally, data is lost only when a job enters a loop while generating items, or tape writing errors occur at a peak period of activity. The average overhead induced by a supervisor-written item is about 65 microseconds.

When a data item is written via an SVC instruction, additional activities are performed which overshadow the data collection time itself: (a) machine status is saved at the supervisor entry, (b) the SVC is traced and the appropriate processing subroutine is called, (in this case the buffer entry subroutine), and (c) the job scheduler is activated to dispatch the appropriate job at the supervisor exit (usually the same job in this case). The total time required for a data collection SVC was measured by repeatedly executing the SVC and differencing the successive timer values appearing in the resulting data items. As an additional check the time required to execute a sequence of "null" SVC instructions was noted: a supervisor call with no processing subroutine. The results are displayed in Figure C-3.

ITEM FREQUENCY DATA

ITEM TYPE	TOTAL COUNT	NO. SELECTED
Overflow	0	0
Date	1	0
Addtotop	7372	4814
Popqueue	7243	4737
Wayt	56339	34438
Unwayt	56327	34433
Queue	328031	197544
Statsw	17	9
Paginstr	702	491
Pagindon	702	491
Pagoutst	783	523
Pagoutdn	779	519
Pagreclm	3621	2098
Getvmpag	4098	2346
Frevmpag	4087	2333
Mark	0	0
Vmpages	5220	2975
Waitfor	54	29
Unload	150	75
Load	152	77
Freespac	9690	5635
Getspace	9858	5685
Dsrin	4496	3188
Dsrout	4491	3185

Total number of input items =	504204
Total number of missing items =	0

Figure C-2. A Sample Distribution of Data Items

Timer units with the System/360 high-resolution timer are of thirteen and one forty-eighth microseconds duration. Thus only two to four different timer values were observed when timing SVC instructions. Since succeeding SVCs occurred at different instants with respect to the beginning of a timer unit, however, a more precise value could be obtained by dividing the total elapsed time by the number of

BUFFER ENTRY SUBROUTINE TIMES* (Sums of mean instruction times.)	PAGING without	with
• Called by supervisor		
short item	58.0	58.0
long item	65.5	65.5
buffer switch item	82.2	82.2
overflow item	34.4	34.4
• Called by SVC instruction		
short item	58.0	74.4
long item	65.5	81.9
buffer switch item	82.2	98.6
overflow item	34.4	50.7
SUPERVISOR CALL TIMES* (Obtained with the data collection facility.)		
• Data collection		
short item	243.3	323.9
longest item	253.5	339.9
no entry		265.8
• Null SVC		261.6
*All times are given in microseconds.		

Figure C-3. CPU Time for Data Collection

SVCs. The data in Figure C-3 is based on averages for 500 or more such successive observations. Data taken before and after paging was introduced differed significantly in accordance with the changes made in the system, but showed surprising agreement with the expectations of the system programmers, and consistency among the various observations.

Two more overhead values for data collection remain to be discussed: the CPU and I/O channel times required for the STAT job to write the data buffers onto tape. These values were easily determined by collecting the standard data for the STAT job itself. The average CPU time required per buffer written was also determined by observing the total time at the end of the run and counting the number of tape records written.

The STAT job execution times exhibited a variation of as much as about 20%, due to the fact that it is nearly all spent in the supervisor and is therefore sensitive to the system load. Slightly over one msec. is required to set up a full buffer for tape writing, and a similar interval is necessary after output to prepare the buffer for reuse. These execution intervals are separated by an I/O wait of about 51 msec. (or 49 msec. for one of the buffers, which is 5% shorter). The overhead per record of data written is about 2.3 msec. of CPU time and 51 msec. of channel time. The total amount of time required by the STAT job depends, of course, on the volume of data collected. Three to eight MTS terminal users working concurrently generate an average of about six records/minute. Each record contains between 120 and 508 items, depending on the buffer length and item sizes, but the value is usually very close to the average of 310 items/record.

Thus the typical overhead values for one minute of operation consist of

6 x 2.3	= 13.8	msec. CPU for STAT job.
65 x 280 x 6	= 109	msec. CPU for supervisor items.
330 x 30 x 6	= 59	msec. CPU for SVC items.
51 x 6	= 306	msec. channel time.

or not more than about 170 msec. (0.3%) of the total CPU time and 306 msec. (0.5%) of channel time.

In summary, then, we assert that both in terms of total time and individual time intervals, the use of system resources by the data collection facility is so small as to have a negligible effect on the other jobs which share them. This is particularly true since the total CPU <u>idle</u> time (also measured with the data collection facility), averaged between 50% and 90% during the months in which this data was collected, and the I/O channel is one of two multiplexor subchannels dedicated to the eight tape drives.

III. Data Analysis

Several programs have been written to analyze and reduce the data collected as described above. The following are the primary goals of the analysis programs:

(a) To print the data items which were collected, with the timer information and other standard data interpreted. An example of this format is given in Figure C-4.

(b) To reduce the standard job scheduler items to a sequence of WAIT, READY, and ACTIVE intervals for each job, sifting out only relevant additional information, such as the reason that an active job terminated execution. A typical page of the printed form of this option is shown in Figure C-5.

(c) To produce distribution tables and graphs which describe the storage execution and I/O wait characteristics of jobs, especially the MTS jobs run from remote terminals.

(d) To provide input data for a simulation of other possible operating system policies and designs, using known workload characteristics.

Functions a and b above are provided by an object program in the MTS file *ANALYSIS. Parameters to the program include a selection of one or both of the a and b formats, a selection of a subset of jobs, item types, and tape locations (records, files) of data to be analyzed. The latter options are necessary in view of the large volumes of data which can be collected by the facility in short periods of time.

The output of *ANALYSIS gives the information for multiple jobs interspersed in chronological order. Yet the reduction indicated in b above requires that the information actually be separated according to jobs during processing. The remaining analysis required is essentially that of interpreting information from the job scheduler tables and queues and isolating what the job actually requested from the way the requests were handled by the supervisor.

Thus we distinguish, for example, between the _active_ intervals when a job has the CPU and the longer "potentially active" intervals for which it would have used the CPU before a wait, if other jobs had not been dispatched.

The information produced by the analysis program can be further analyzed using several modes of output the program provides. Options are available for passing the data to subroutines. A binary tape output format is also provided from *ANALYSIS for input to other programs. The information provided in these formats is more comprehensive than that which is printed.

Functions c and d above are carried out by a second program (STP2) which can read the tape output from *ANALYSIS or be called as a subroutine. STP2 produces two output tapes, which are formatted for input to the GPSS/360 models described in Appendices D and E. The model explained in Appendix D consists solely of tabulation statements to collect and display MTS job characteristics.

The second simulation model, described in Appendix E, forms a major tool in this work for the investigation of operating system phenomena. The input data for this simulation takes the form of models of the actual jobs run in MTS, with most of the characteristics due to MTS itself removed.

```
NAME       JOB    MICROSEC  ITEM IN HEXADECIMAL...             1         1

QUEUE       19    12163580  32130013  00FF0E41  2608007D
PAGINSTR    19    12164700  44690013  00FF0E41  005F0102  4C010064  16050300
QUEUE        3    12165195  328F0003  00FF0E41  12320013
QUEUE       19    12216393  32EB0013  00FF0E50  01000003
PAGINDON    19    12216497  4CF30013  00FF0E50  00330102  2C010130  16070300
ADDTOTOP    19    12217031  121C0013  00FF0E51  4A007384
WAYT        19    12219986  23FF0013  00FF0E51  924A00FF  20102005
QUEUE       39    12220416  32200027  00FF0E52  12220013
QUEUE       19    12222929  32E10013  00FF0E52  2608007D
ADDTOTOP    19    12223958  12300013  00FF0E53  28007274
POPQUEUE    19    12224388  1A510013  00FF0E53  4A007384
UNWAYT      19    12224570  2A5F0013  00FF0E53  920075C4
POPQUEUE    19    12228020  1A680013  00FF0E54  390072FC
QUEUE      125    12228424  3287007D  00FF0E54  12220013
QUEUE       19    26295989  32CC0013  00FF1ED0  00000000
ADDTOTOP    19    26297057  121E0013  00FF1ED1  8300754C
POPQUEUE    19    26297916  1A600013  00FF1ED1  390072FC
UNWAYT      19    26298098  2A6E0013  00FF1ED1  00000000
PAGINSTR    19    26299179  44C10013  00FF1ED1  00290100  4C010198  16050430
QUEUE        3    26299739  32EC0003  00FF1ED1  12320013
QUEUE       19    26341458  32700013  00FF1EDE  01000003
PAGINDON    19    26341549  4C770013  00FF1EDE  00290100  2C010280  16070430
DSROUT      19    26345846  FEC10013  00FF1EDF  D3C1F1F3  00000010  5BE2C9C7
PAGINSTR    19    26348255  447A0013  00FF1EE0  006A0101  4C010280  16050532
QUEUE        3    26348802  32A40003  00FF1EE0  12320013
QUEUE       19    26392552  32C40013  00FF1EED  01000003
PAGINDON    19    26392656  4CCC0013  00FF1EED  00500101  2C010380  16070532
GETSPACE    19    26393528  EA0F0013  00FF1EEE  00000018
GETSPACE    19    26394153  EA3F0013  00FF1EEE  00000030
GETSPACE    19    26395013  EA810013  00FF1EEE  00000068
GETVMPAG    19    26395677  6CB40013  00FF1EEE  00000103  08000380  10000000
PAGRECLM    19    26395898  64C50013  00FF1EEE  00570103  0C000480  10020000
GETSPACE    19    26396562  EAF80013  00FF1EEE  00000E30
VMPAGES     19    26397174  C4270013  00FF1EEF  000D7EF8  00FF1EEF  00000008
GETSPACE    19    26397695  EA4F0013  00FF1EEF  000001D8
```

Figure C-4. A Sample of Annotated Data

```
TASK     JOB     STATE           MICROSEC    ITEM    WAIT   DEV    FDNAME   I/O   LENGTH   PFX   LINE

****** THE RECORDING DATE WAS 03-21-68   AND THE TIME WAS 15:28.27
  1      19      UNSURE          10657643       1     UNKN   0000              00      0
  1      19      ACTIVE              1614       2    VMPAGES=   0
  1      19      UNSURE             51197       3     UNKN   0000              00      0
  1      19      ACTIVE              4023       4    VMPAGES=   0
  1      19      WAIT                2513       5     ¬I/O   00FF              00      0
  1      19      ACTIVE              5494       6    VMPAGES=   0
  1      19      UNSURE          14067565       7     ¬I/O   00FF              00      0
  1      19      ACTIVE              3750       8    VMPAGES=   0
  1      19      PAGEWAIT           41718       9
  1      19      ACTIVE              7343      10    VMPAGES=   0
  1      19      PAGEWAIT           43750      11
  1      19      ACTIVE             11302      12    VMPAGES=   4   GET=   4968
  1      19      WAIT               58919      13     I/O    0217              00     16              $SIG
  1      19      ACTIVE              3789      14    VMPAGES=   4
  1      19      WAIT               24335      15     I/O    0217              00     16              $SIG
  1      19      ACTIVE              2356      16    VMPAGES=   4
  1      19      READY               2851      17
  1      19      ACTIVE              5455      18    VMPAGES=   4
  1      19      WAIT               31132      19     I/O    0217              00     16              $SIG
  1      19      ACTIVE              3033      20    VMPAGES=   4
  1      19      WAIT               11132      21     I/O    0217              00     16              $SIG
  1      19      ACTIVE              3502      22    VMPAGES=   4
  1      19      READY               2669      23
  1      19      ACTIVE              7486      24    VMPAGES=   5   GET=   2136
  1      19      READY               2291      25
  1      19      ACTIVE             10208      26    VMPAGES=   5
  1      19      READY               5260      27
  1      19      ACTIVE              3919      28    VMPAGES=   5
  1      19      WAIT               15260      29     I/O    0217              00     16              $SIG
  1      19      ACTIVE              1835      30    VMPAGES=   5
  1      19      READY               2356      31
  1      19      ACTIVE              5156      32    VMPAGES=   6   GET=   3632   FREE=    432
  1      19      READY               1731      33
  1      19      ACTIVE               221      34    VMPAGES=   6
```

Figure C-5. A Sample of Analyzed Data

Appendix D: The MTS Data

This section describes the data collected from the University of Michigan MTS time-shared operating system using the facility for data acquisition explained in Appendix C. The nature and limitations of the data are discussed and the data is displayed in tables and cumulative frequency distributions.

I. General Description

The data acquired for this study was taken during the period October 15, 1967, to March 31, 1968, in normal operating periods of the MTS system. Data collection periods ranged from 15 minutes to 7 hours duration, depending on the volume of data being generated and the nature of the jobs being observed. The periods were selected insofar as possible to mirror the typical prevailing demand on the system: unusual circumstances of light load (such as just after system startup or malfunction) and heavy load (such as the hours preceding a student problem due date) were avoided.

Essentially three types of jobs are run in the UMMPS-MTS system:

(a) Normal remote terminal, interactive use of MTS with Model 33/35 Teletypewriters, IBM 1050 and 2741 Communications Terminals. (During the data acquisition period the number of such tasks which could be supported concurrently by the system grew from about 4 to 40).

(b) Non-interactive use of MTS via batch mode, using an IBM 2540 card reader/punch and 1403 line printer as input/output devices, while providing full use of the command language and other system features.

(c) Peripheral support programs for an IBM 7090 batch-processing system (University of Michigan Executive System [70]) which produce input tapes from punched cards, and print and punch from UMES output tapes.

During normal daily operation of the MTS system, from 9 a.m. to midnight, the then current maximum number of communications lines for remote terminals was enabled, one or two batch streams were processed, and up to two additional line printers and reader/punches were used for peripheral support jobs. Generally, less than half of the remote terminals were active at one time, one batch stream was rather consistently busy, and an average of two to three of the SPOOLing[1] operations were in progress. Data are not included here for the SPOOLing jobs, since they exhibit a very regular behavior: each I/O wait for tape or unit record device requires 50-200 msec., and is separated from the next such event by 2 to 3 msec. of processing. These peripheral support programs create a maximum of about 15% of the CPU load, and normally only 5-10%.

Data given here for the use of the CPU and I/O devices are separated for batch and conversational tasks. The I/O delays are also shown separately for different kinds of devices. A great deal of information is implicit in the data being collected which is not shown here, and additional kinds of data can be collected to answer specific questions.

At least three distinct points of view could be taken to govern the organization and collection of computing system data:

- workload
- system
- user

For the purposes of this investigation we have regarded their relative importance in the given order. For example, in displaying CPU data, we are more interested in the CPU intervals requested by a task than by the service actually supplied by the system. Again, when considering I/O delays, we are concerned about the length of time the system must wait before it can resume processing a task rather than the time an individual must wait to receive his answer. Thus in our case some output delays appear to take almost no time because the lines are buffered and the computation can proceed while the line is still being typed.

----

[1]Simultaneous Peripheral Operations On-Line.

II. Specific Characteristics

Some facts about the MTS data chosen for display in this section are given in Figure D-1. The number of jobs is the approximate number of UMMPS jobs using the MTS job program (see Appendix A) that were observed with the data collection facility. The number of tasks is the approximate number of individuals who used these jobs during the collection interval. The total number of unit record devices, communication lines, disks, etc., that were referenced by these jobs is also given. The identification numbers 0-5 of these different sets of data will be used to label graphs in the succeeding figures.

Several general observations can be made about the environments in which the data was taken. MTS # 4 was obtained after only a few weeks of experience with paging in MTS. At that time the system had a communications line capacity of about 10: at most 10 conversational users could run in addition to the batch and SPOOLing operations. Furthermore, use averaged considerably less than the capacity. The data sets # 2 and # 3 include only data for the batch streams active at the time. Because they use unit record devices instead of remote terminals, batch jobs are processed much more quickly and create a heavier system load than conversational jobs. More I/O operations generally occur in batch, since people take advantage of its lower cost and speed for input and output functions.

The data # 1 was taken with 10-20 conversational users, which approximated an average load at that stage of MTS development. With less than about 16-20 users, the paging drum frequently had periods of inactivity: most command chains for I/O operations were half-empty, and there was often no channel program in progress at all. In order to observe tasks under a heavier system load, data set # 0 was taken while a special background job (called PAGE-IT) was running to increase drum activity. The PAGE-IT job acquires a large number of virtual storage pages and references them cyclically in rapid succession. When run with a moderate load of normal tasks, PAGE-IT keeps drum channel programs running more or less continuously and forces other tasks' pages out of main storage at a faster than normal rate.

Figure D-2 shows the distribution of actual CPU intervals obtained by tasks during the data collection periods. Any interruption in service to process another task ends the CPU

General Characteristics of the MTS Data

ID	MTS # 0	MTS # 1	MTS # 2*	MTS # 3*	MTS # 4
Date	3-20-68	3-18-28	3-18-28	1-15-68	11-29-67
Time	15:27	15:36	11:41	14:51	10:23
Duration	20 min.	76 min.	150 min.	79 min.	266 min.
# Items	542246	1355274	1123503	466223	705890
# Jobs	40	40	2	1	9
# Tasks	50	150	75	16	84
Devices	57	65	29	20	29

* Denotes batch job data.

Figure D-1. The Selected Data

intervals appearing in this distribution. One to four percent of these intervals lie in the neighborhood of about 300 microseconds, or about the minimum time required by UMMPS to service an interrupt which requires little or no processing. The smoothest curve, and the one with the smallest mean, is that of data set # 0. The reason for this is that the system was the most occupied with ordinary tasks at that time, and PAGE-IT was running to force additional page-wait interruptions for the normal tasks. Since # 0 was the latest of the given data to be taken, it also reflects some improvements made to the system which make it operate more efficiently under periods of heavy load.

Actual CPU intervals depend on the frequency of interrupts, hence it is not surprising that the data set (# 2) with the longest mean value was collected over a noon hour. Except for that case, the mean length of a CPU interval decreases steadily with time—average system load grew significantly over the period of study. Data set # 1, whose curve shows a large number of intervals of length about 4 msec., was taken during the testing of a new I/O routine when an error occurred, and over 28,000 I/O operations were executed in rapid succession, generating an equal number of short CPU intervals.

In Figure D-3 we have distributions of CPU intervals requested by individual tasks: here the interruptions in service to process other tasks are removed from the data, so that the given times represent CPU intervals terminated only by I/O and paging operations for the task using the CPU. The curves for # 3 and # 4 also accumulate time across paging delays for the given task—they show the distributions of CPU time requested between I/O operations. We note here that both of the latter curves exhibit a gap in observed values near the origin: there are a few nearly immediate I/O operations, but then almost none of duration 800 to over 2000 microseconds. The difference between the curves for # 3 and # 4 is probably due to the fact that the former data is for batch jobs, which generally do more I/O operations.

Turning to the curves for # 0 and # 2 in Figure D-3, which represent CPU intervals terminated by either page or I/O wait for the given task, we see again that a heavy paging load (# 0) significantly reduces the mean length of a CPU interval. There is too much variation in the origin and makeup of batch jobs to draw hard and fast conclusions from differences between the curves for # 1 and # 2, which represent conversational and batch jobs run on the same day. Batch runs include a number of distinctly different tasks: small student problems limited to batch mode, use of faster devices for listing and card reading jobs, and long computational tasks. The difference between the two curves is probably more influenced by the fact that the batch data was taken earlier in the day, when a lighter overall load contributed an average of fewer page-wait interruptions. The density functions for requested CPU intervals exhibit a number of local maxima in the range 0-6 msec., which are present in almost every case and more noticeable with less paging load. A careful analysis could probably associate these peaks with one or more frequently-used system functions.

Figure D-4 displays the distributions of page-wait delays experienced with the MTS paging drum processor. The organization of the queues and the format of the drum are detailed in Appendix B. The mean values of these distributions strictly increase with increasing system load, which is the reverse order from which the data sets are numbered. Although the given data all represent the time required to obtain a requested page, each curve is a composite of several different kinds of distributions. Once an unavailable page is referenced one of five actions may be taken:

(1) it is a new page for which space can be allocated immediately in core (1-5 msec.)

(2) it is a new page for which core space can be allocated only after another page has been pushed out (a written page may be posted at any time)

(3) an existing page must be read from the drum (at least 36 msec.)

(4) an existing page must be read from the drum but must wait for a write to provide core space

(5) the page may still exist in core even though a write operation is currently in progress, which may be cancelled to make it available immediately.

If a drum operation is required (cases 2 to 4 above) then a further distinction is possible: whether or not the drum is currently under the control of a channel program to which the new request(s) can be chained. If so the distribution of actual completion time is that discussed in Appendix B. If the drum is not currently transmitting an additional average delay of half a physical revolution is experienced, since a new channel program is always started at the drum index point. In the latter case, however, it is unlikely that more than a single revolution will be necessary to reach and transmit the desired page.

One final fact is of importance in understanding the distributions of Figure D-4: MTS page transfers were posted (at the time) exactly once at the end of each logical revolution of the drum. Thus every actual read or write operation is known to be completed only after some multiple of the logical revolution time (35 msec.), plus any delay in synchronizing with the construction of channel programs.

All the data in Figure D-4 is dominated by the characteristics of start-stop rather than continuous drum operation except # 0, where the drum was forced to run more or less continuously. In that case a great many read operations took three logical revolutions or more. Apparently most available core pages were "reclaimed" or used for newly created pages, so that read requests for drum-resident pages often had to wait for drum writes as well as queueing and read delays. In any case the performance of the drum under the conditions of # 0 leaves a great deal to be desired.

The overall distribution of I/O wait times given in Figure D-5 is poorly shown due to a gap in the plotted points in the range from 0.2 to 2.5 seconds. Most delays fall into three ranges according to the type of device:

(a) terminal and other I/O to buffered devices which appears to take almost no time at all

(b) disk and unit record I/O, most of which lies in the range from 35 to 70 msec.

(c) terminal output with buffer full, and unbuffered input, which usually takes over half a second.

The curve for MTS # 1 is missing from Figure D-5 because of the large number of identical I/O waits occurring in the test mentioned earlier. Batch and conversational jobs have distinct distributions—in the former case the longest normal I/O operation is a maximum disk seek, which requires about half a second.

The ready distributions of Figure D-6 show how the MTS system responds to requests for service. This data is rather consistent. The fact that # 0 has the shortest mean is again due primarily to the fact that the heavy paging load forces many more transitions between tasks entering page-wait.

The remaining figures in this section give disk and terminal characteristics. The disk observed during this period was the IBM 2314 Disk Storage Unit, which provides a bank of eight separate packs on a single control unit and channel. The disk is used in MTS for line file storage and utility files for compilations and assemblies. In Figure D-7 we see the actual lengths of disk I/O operations. The principal components are

- control unit wait (transmission of data or commands from another disk pack on the unit

- seek time to move the read/write heads to the proper cylinder (which is often not required)

- rotational delay to reach the front of the appropriate record on the disk track

- transfer time for actual reading or writing of the record

The distribution with the longest mean is that for the data taken under the heaviest load. Under these conditions we expect control unit wait to be a significant factor, since at the time up to six of the packs could be in competition for the use of the single control unit. Another very large factor is the frequency with which seeks are necessary: a single task will often require several records from the same cylinder in short succession, hence few seeks are necessary unless the load is such that a task can obtain only cne record at a time before another task causes a seek to a new cylinder.

The fact that the earliest data has the second longest mean value is due in part to the fact that less efficient file routines were in use at the time, which required more long search operations that tie up the control unit. Another cause is the fact that only four disk packs were available. Similar remarks can be made for Figure D-8, which shows the distributions of times that tasks had to wait in queue for the use of a specific disk pack. The times in Figure D-8 are somewhat inflated because the end of a wait for a pack is non-interrupting, and hence it is not discovered to be over until the task reaches the head of the CPU queue.

Finally, we display in Figure D-9 a few distributions of terminal I/O times. Two distributions each are given for specific Teletype lines from the three sets of conversational data. Although this data exhibits considerably more variation than the synchronous intervals, we can see the effects of the following characteristics of terminal I/O operation: over half the times are for output lines, which clearly predominate. Many of these lines can be placed immediately in the one-line output buffer, so that most output times are less than the time required to actually print a line at the remote device. Shorter times for the input lines of MTS # 0 data suggest that as system response time moves away from practically instantaneous, an individual can make use of the time to formulate his next command, which he then gives more quickly.

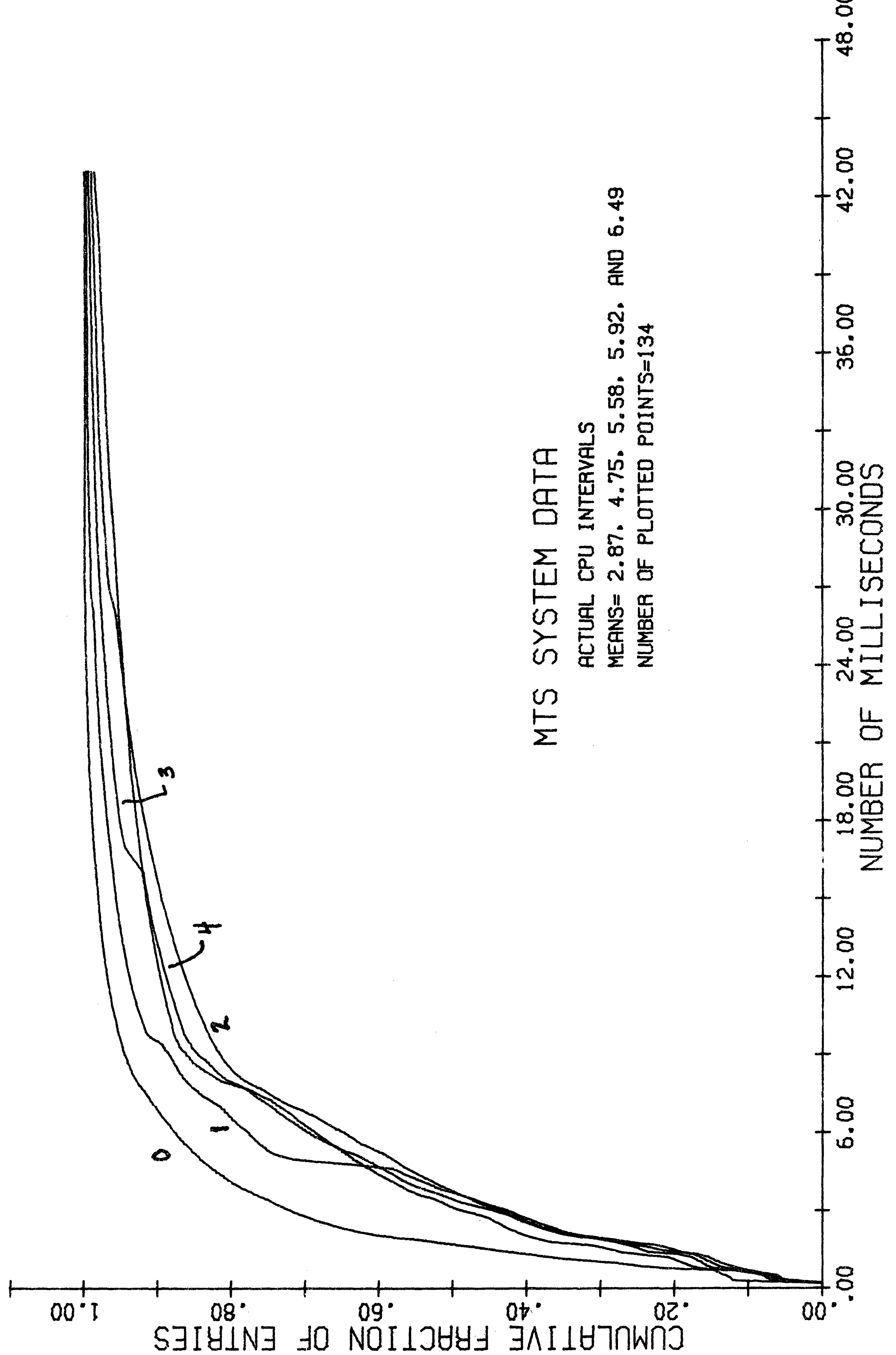

Figure D-2. MTS CPU Intervals

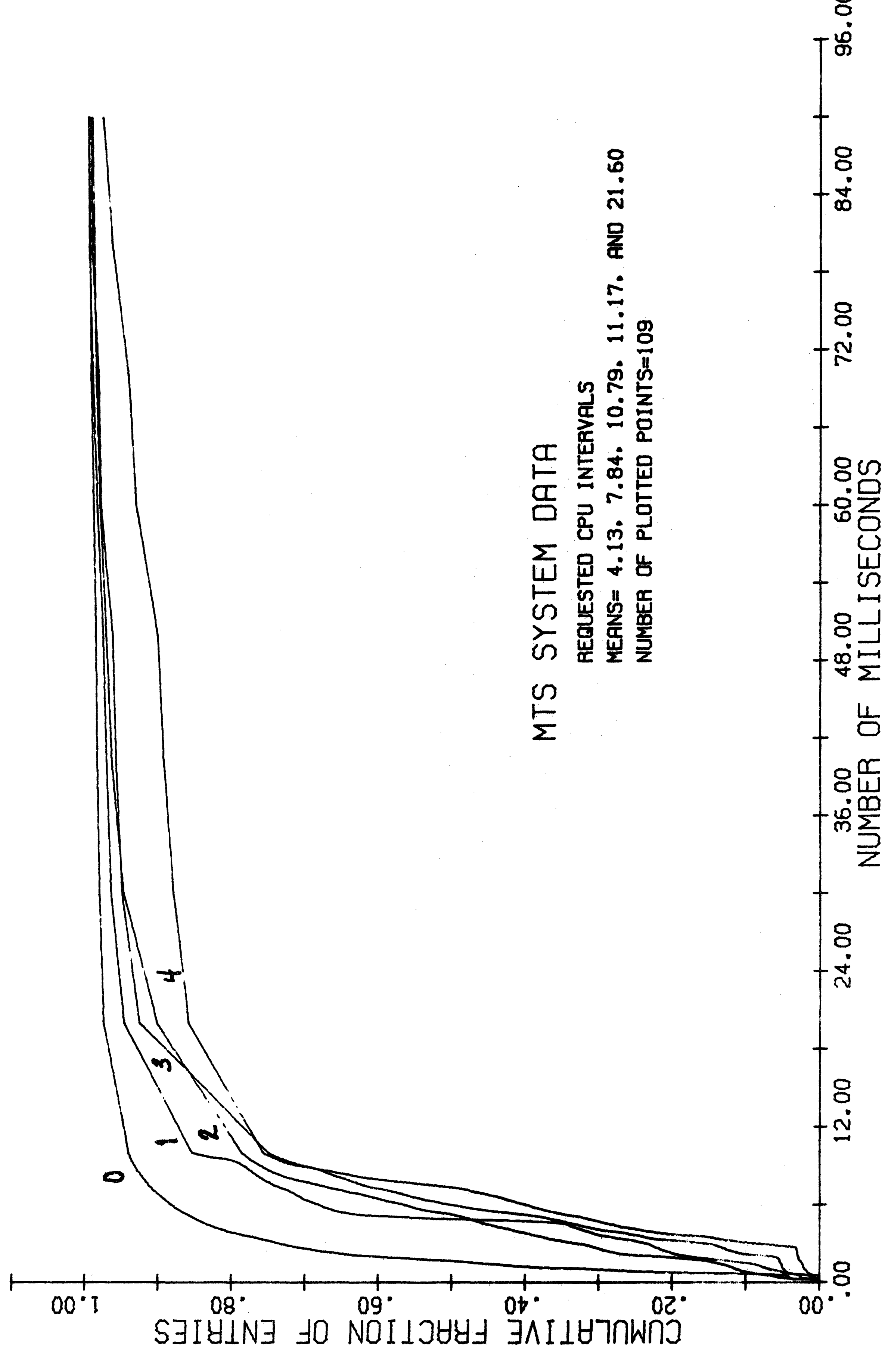

Figure D-3. Requested CPU Intervals

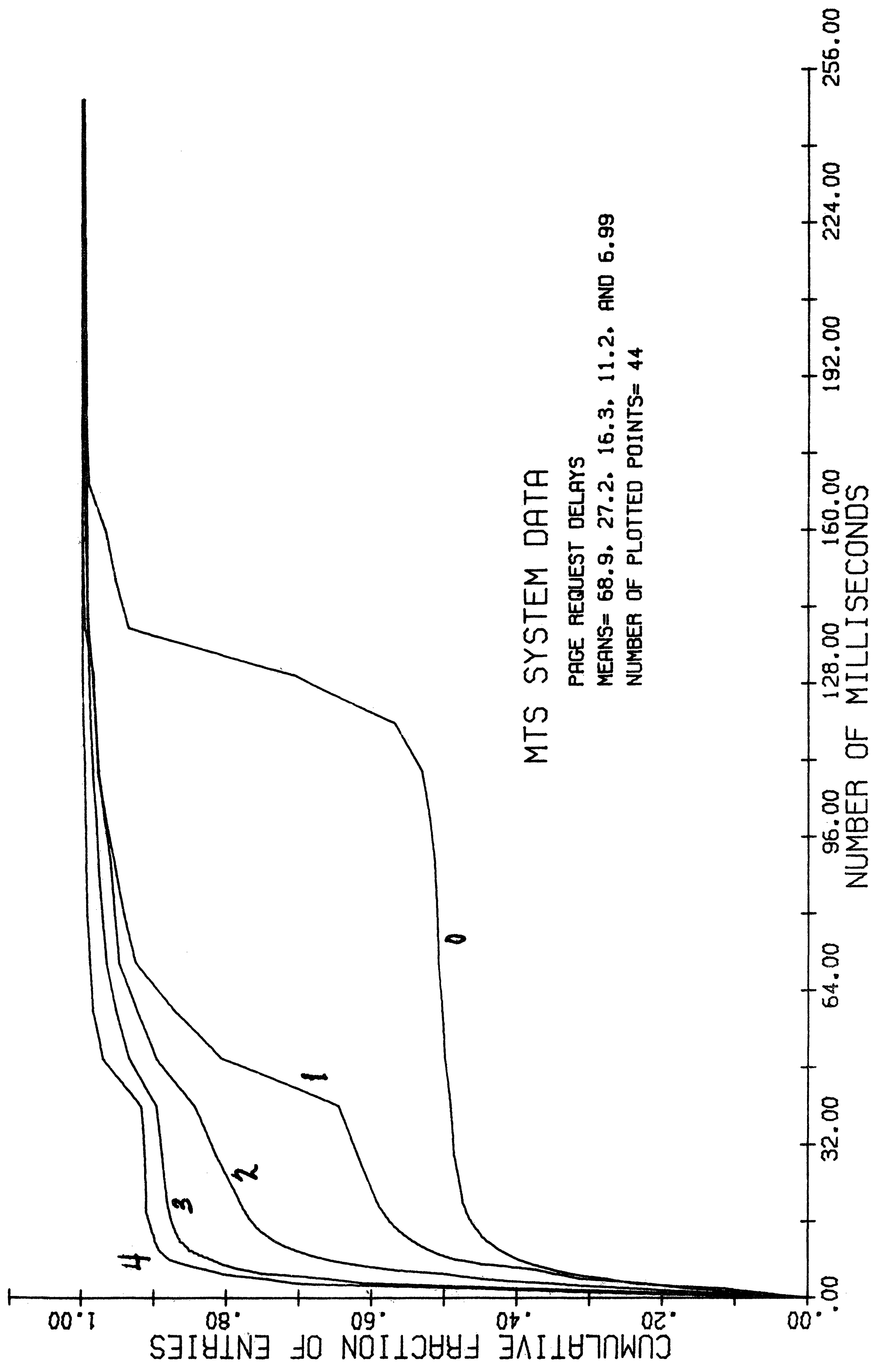

Figure D-4. Page Wait Delays

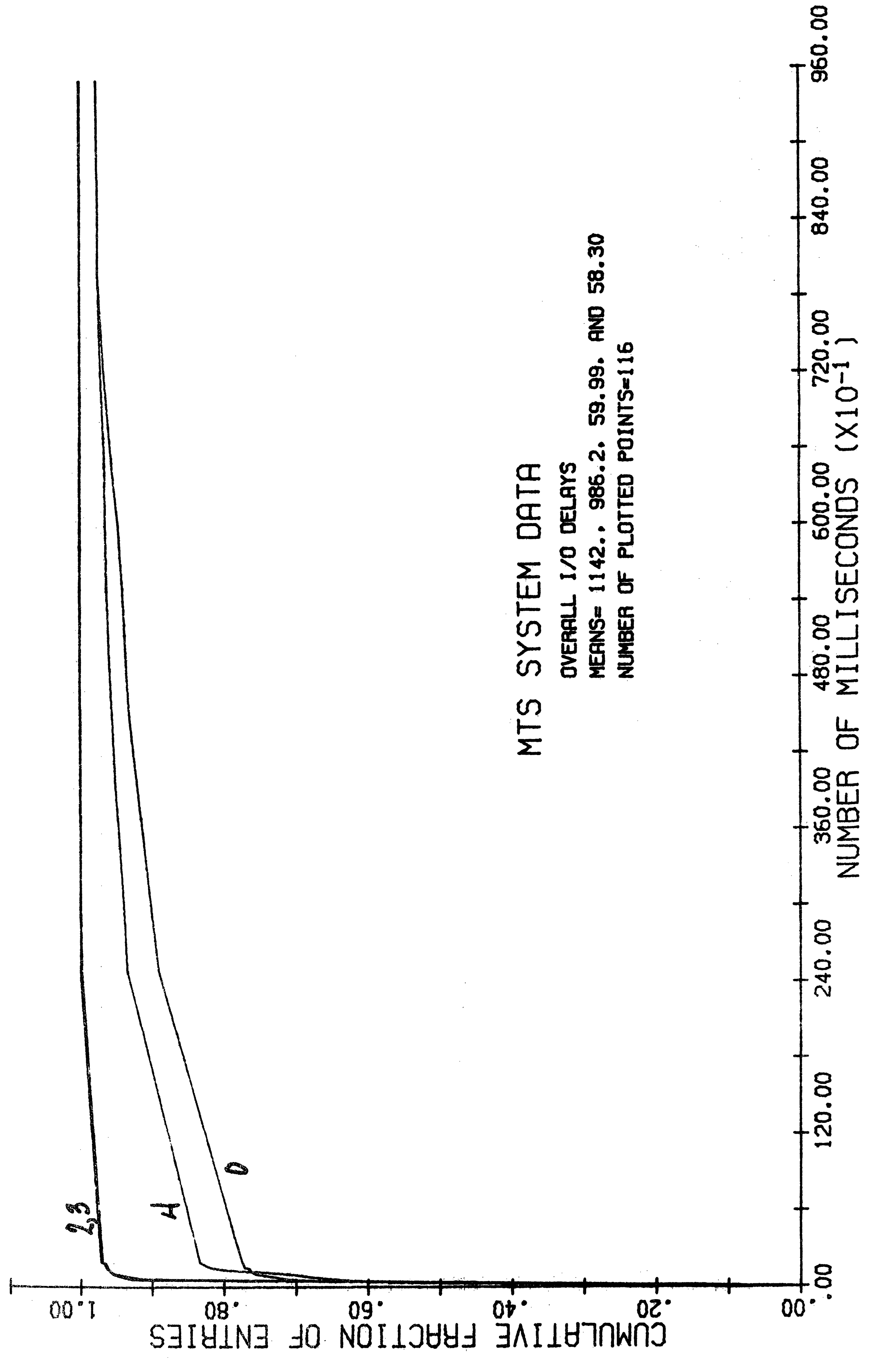

Figure D-5. Overall I/O Wait Distribution

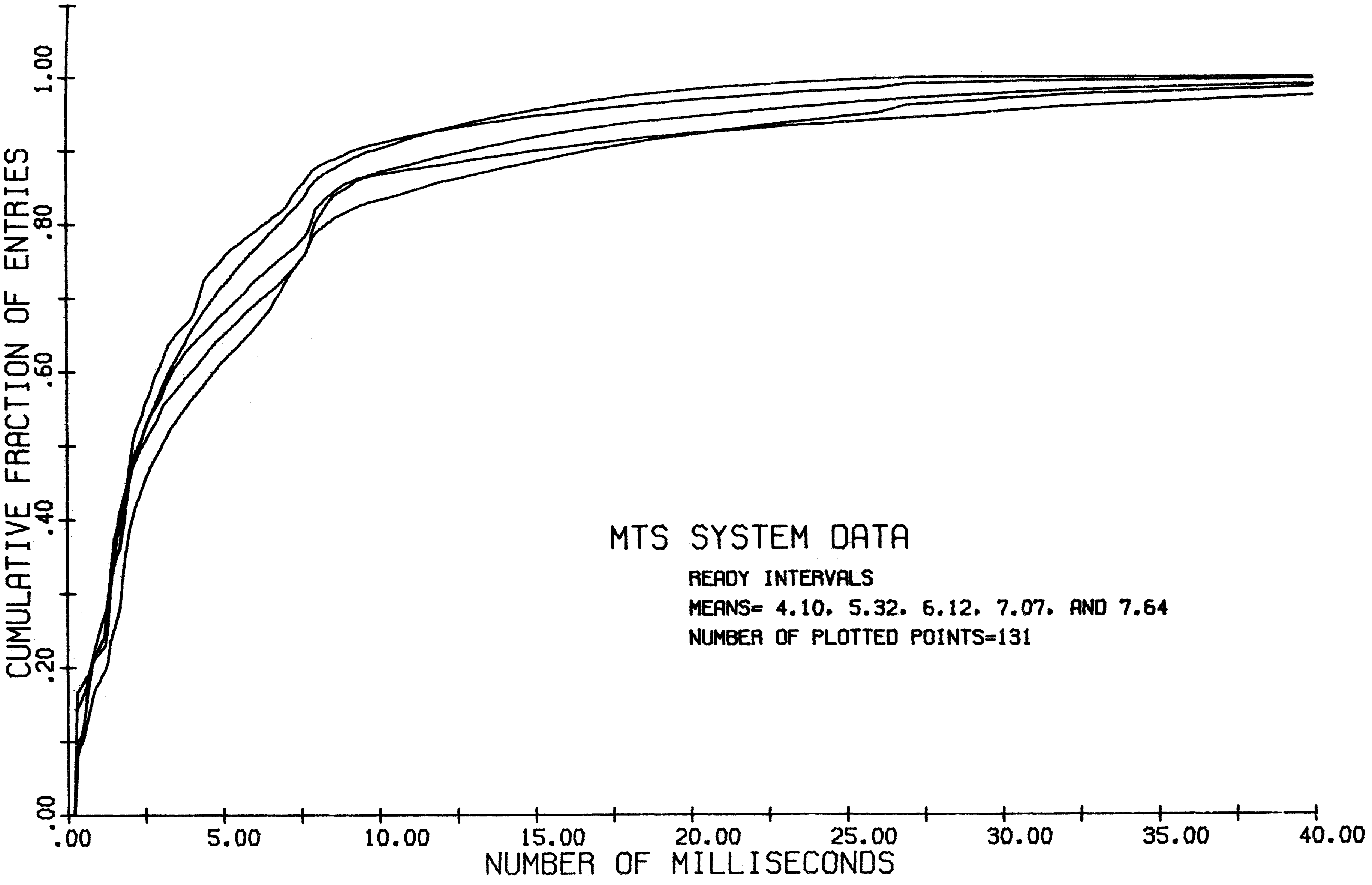

Figure D-6. MTS Ready Intervals

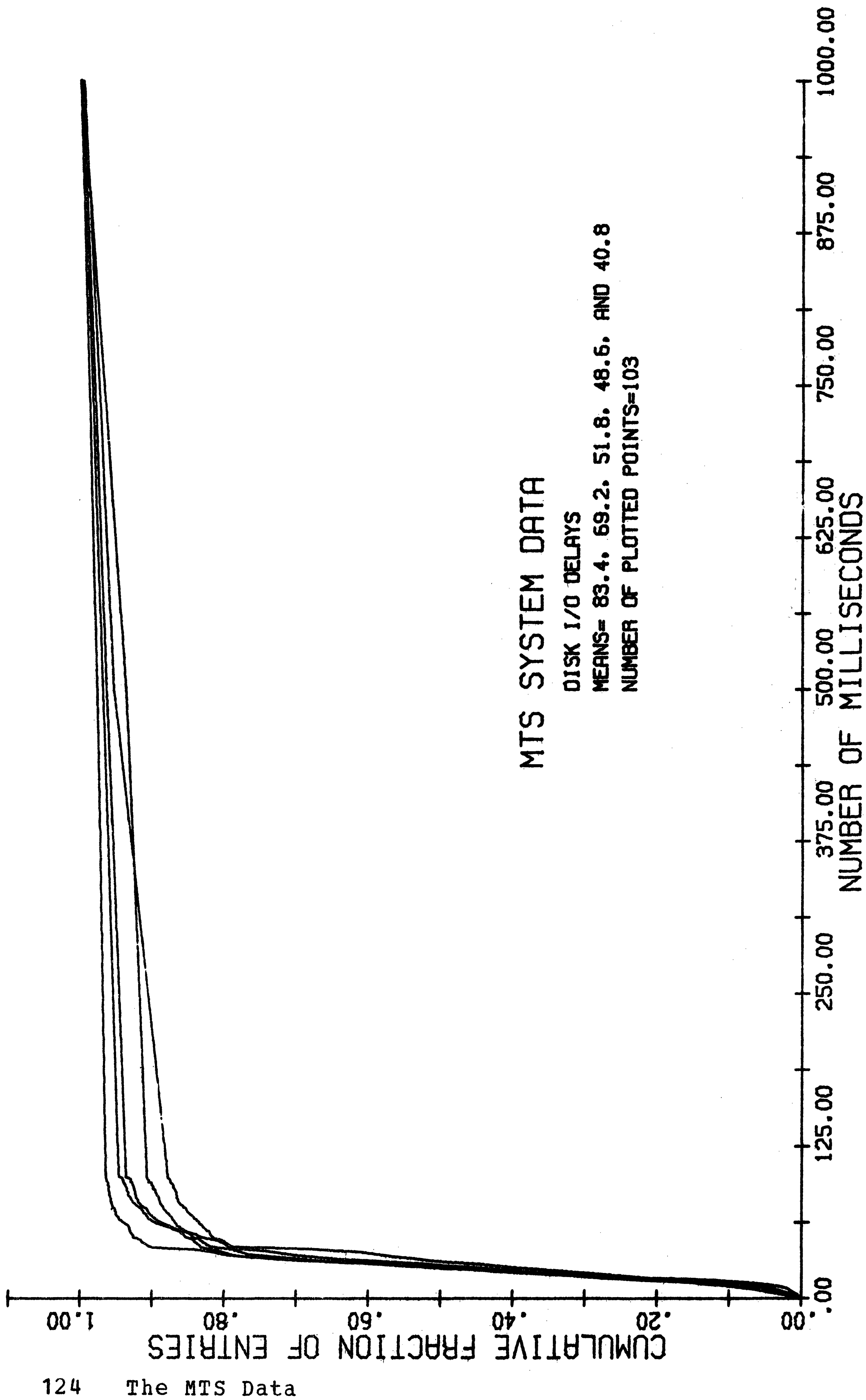

Figure D-7. Disk I/O Delays

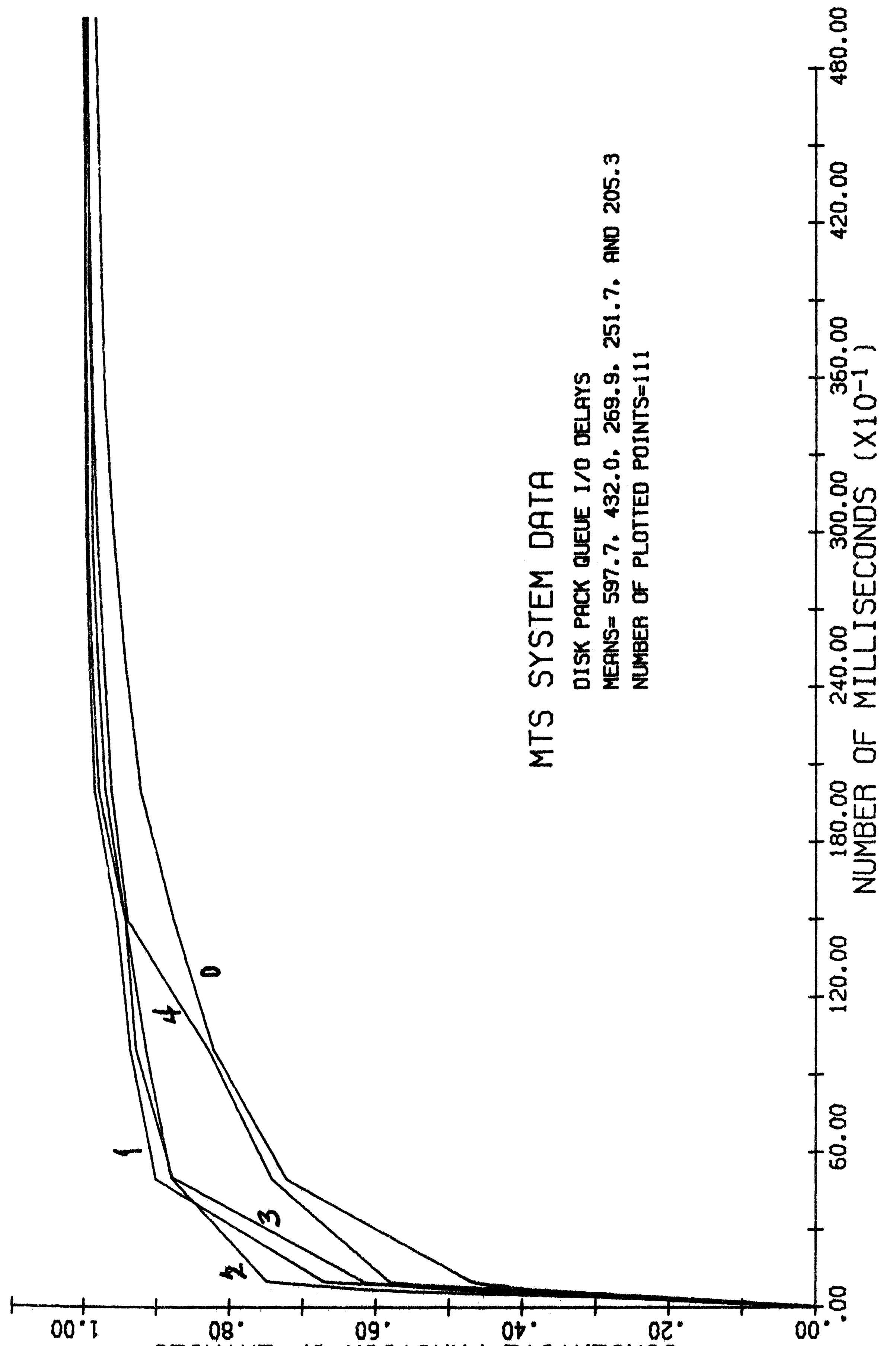

Figure D-8. Disk Pack Queueing Delays

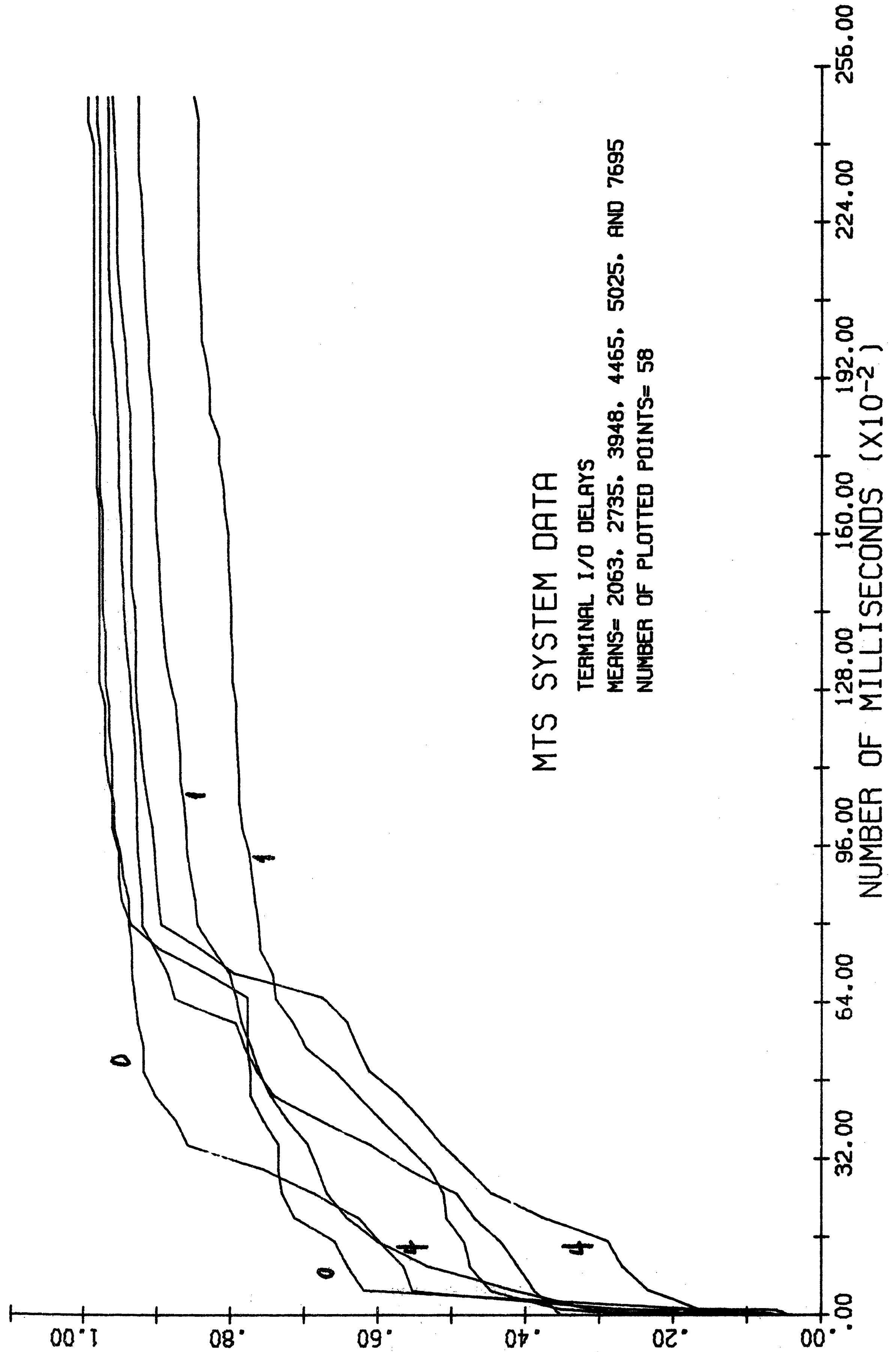

Figure D-9. Terminal I/O Waits

Appendix E: The GPSS/360 Simulation Model

This section discusses the details of implementing the simulation model of Chapter 5 in the GPSS/360 language.

I. GPSS/360

The General Purpose Simulation System/360 is a discrete, digital simulation program developed by the IBM Corporation [34] which runs in the standard IBM Operating System/360. Simulation model statements, after simple processing by an assembly program, are interpreted during execution by the GPSS/360 program.

GPSS/360 is the outgrowth of a series of general purpose system simulators for the IBM 7000-series computing machines, the most recent version of which is titled GPSS-III. GPSS/360 relaxes many GPSS-III restrictions, introduces new entities and block types, allows much more control over storage allocation, and provides a limited graphic output feature.

The GPSS language is particularly suited for modelling in such commercial applications as job shop scheduling and manufacturing network flow. However, it is also useful for computer system simulation at the fairly gross level of detail which is used here. Its primary disadvantages are that it is comparatively difficult to learn and relatively slow in execution. Its generality and flexibility were considered to outweigh those disadvantages for this particular application.

II. Organization of the Model

The simulation model described in this paper consists of about five essentially different parts:

- hardware descriptions
- operating system algorithms
- workload characteristics
- variable system parameters
- statistics gathering

The first of these is the simplest to implement in a simulation language, and requires a negligible fraction of the set of model statements. Characterizing the data in the system (in this case requests for computing services) and specifying the parameters by which the system is "tuned" to improve its performance require about the same amount of effort, and together account for only a fifth or so of the total description. By far the most design and analysis work and the resulting model statements are required to describe the algorithms used by the operating system for handling devices and service requests, and to decide what information about the model operation is relevant and measurable, and how it should be collected and presented. Each of these two aspects of this particular model required over a third of the total simulation effort.

The following sections will discuss each of the basic aspects of the model in terms of its content, overall structure, and the GPSS statements essential to its implementation.

III. Model Data: the workload description

Chapter 5 of this paper has described the motivation and methodology for collecting data from a specific system for use in this study. An approximate description of each MTS interaction is given by the parameters of a single transaction (XACT). The size of the executing program, the time intervals required for execution and synchronous I/O waits, and the frequency of requirements for additional virtual memory pages during execution are all described by XACT parameters.

The master XACT representing each MTS interaction is passed through the job scheduling part of the model. Transactions are also used elsewhere to stand for a single memory page as it passes through the drum I/O channel, an entry on the queue for use of the CPU, and to implement the data transfer during the drum rotations.

Virtual memory page XACTs required by a task are members of a GPSS assembly set, and a specified number of them must be ASSEMBLEd before a task is ready for execution. These XACTs are SPLIT from others as needed, and TERMINATEd when they are no longer required to trigger additional events by their flow through the model.

IV. Hardware Descriptions

The only parts of this model which directly represent physical devices occur in the choice of the time unit as the drum sector rotational delay, and in the model parameter which specifies the capacity of core storage. Hardware/software combinations such as the operation of the drum as an extension of main memory and the supervisor algorithm which drives it are also modelled.

As each page transfer request arrives at the drum processor, it is LINKed to one of the GPSS "user chains," which are used to represent drum sector queues. Once every time unit a special "clock" XACT passes through an UNLINK block which attempts to remove a queue entry from the sector currently at the read/write heads. Any XACT so UNLINKed is next ADVANCEd for one time unit to represent the transfer time, and then either sent to be ASSEMBLEd for CPU service (if a page-in request) or simply TERMINATEd to remove it from the model (if a page-out request).

V. Operating System Algorithms

Additional algorithms represented in the model for the operating system itself include

(a) the choice of when to interrupt the task currently using the CPU

(b) the choice of when to allow a new task to bring its pages to real memory and compete for the use of the CPU

(c) the choice of when to page-out a task not ready to use the CPU.

Decisions are made with these algorithms at specific "control points" in the model by routing task XACTs based on the value of task characteristics (XACT parameters), system load factors (QUEUE lengths and STORAGE usage), and decision-aiding parameters (SAVEVALUEs). The control points occur when the executing task requests a virtual memory page which is not in real memory, when a task has used a certain amount of CPU time, when real memory usage drops enough to allow room for the pages of another task, when a task begins an I/O operation, and when an executing task is interrupted.

VI. System Parameters

Decision-making algorithms for operating systems are normally designed to use a number of parameters--numerical values which can be changed as frequently as desired to improve performance as more is learned about how the system behaves, as the hardware configuration changes, and as the workload develops different patterns of demand for system resources. Examples of such parameters existing in this model are

- the maximum amount of real memory available to a single task at one time
- the maximum length of time a task may continuously use the CPU
- the maximum number of real memory page requests allowed to enqueue in the system when core is full
- the number of real memory pages required to be free before a new task is allowed to compete for the CPU.

These values are stored in GPSS SAVEVALUE locations. They are initialized for each run, and can be changed during the simulated operation if necessary.

VII. Statistics Gathering

Many statements in the model exist for the purpose of measuring and summarizing its behavior. Of particular interest are the performance of the system in terms of the percentage of time the CPU and other facilities are used (by means of standard data accumulated by GPSS for STORAGEs, FACILITYs, and user chains), the characteristics of congestion occurring in various places as tasks enqueue for use of devices (using standard GPSS data for QUEUEs, interarrival times and interarrival rates), and the relationship of task characteristics to the quality of service obtained (by TABULATEing task data at points in the model).

The information produced by statistics-collecting model statements is displayed in terms of frequency distributions, mean values, cumulative percentages, and maximum and minimum values. Although such data is much easier to collect from a simulated system than a real system, care must be taken in choosing relevant data and the appropriate places to measure it. In a typical run statistics are displayed for a number of intervals, using the GPSS RESET feature, so that they may be examined for consistency and stability. Statistics must also be ignored at startup to allow transient effects to disappear (hence the interval before the first RESET is ignored).

VIII. The Program

This section consists of a listing of the GPSS/360 model used for the simulation study.

```
        REALLOCATE XAC,1399,BLO,210,FAC,3,STO,4,QUE,5,LOG,6,TAB,16
        REALLOCATE FUN,1,VAR,3,FSV,10,HSV,35,CHA,10,GRP,0,BVR,0
        REALLOCATE FMS,0,HMS,0,COM,340000

        SIMULATE

* XACT PARAMETERS
*        PAGE TRANSFER PARAMETER:
*   1      READ REQUEST=NEGATIVE OR 1,  WRITE REQUEST=2
*        CPU SERVICE PARAMETERS:
*   2      NUMBER OF XACT TO ASSEMBLE BEFORE READY FOR CPU
*   3      AMOUNT OF CPU SERVICE REQUIRED THIS TIME
*   4      LENGTH OF I/O WAIT FOLLOWING CPU SERVICE, IF ANY
*   5      REASON CPU SERVICE TERMINATED
*               1=ASYNCHRONOUS I/O (COMPLETION)
*               2=SYNCHRONOUS I/O
*               3=PAGE IN REQUEST
*               4=TIME SLICE END (TSE)
*        MASTER TASK RECORD PARAMETERS:
*   6      TOTAL NUMBER OF PAGES IN TASK
*   7      NUMBER OF PAGES CURRENTLY IN CORE
*   8      TOTAL AMOUNT OF TIME USED SO FAR
*   9      AMOUNT OF CURRENT INTERVAL USED SO FAR
*  10      INDEX OF CURRENT TIME INTERVAL PARAMETER
*  11      FIRST TIME INTERVAL
*  12      REASON FOR TERMINATION
*  13      SECOND TIME INTERVAL
*                   .
*                   .
*                   .
*  END PARAMETERS WITH ZERO TIME INTERVAL
*                   .
*                   .
*                   .
*  98      CUMULATIVE NUMBER OF CPU INTERVALS ACTUALLY TAKEN
*  99      CUMULATIVE NO OF SYNCHRONOUS I/O WAITS TAKEN
* 100      INDEX OF LAST NON ZERO PARAMETER IN INTERVALS LIST
*     LAST XACT ON TAPE IS ONLY ONE WITH A WRITE REQUEST P1
* VARIOUS TYPES OF PARAMETERS FOR ADJUSTING MODEL OPERATION

* CONTROL PCLICY PARAMETERS
       INITIAL    XH2,16          TIME SLICE LENGTH
       INITIAL    XH20,2          MAXIMUM LENGTH OF CORE QUEUE
       INITIAL    XH21,38         MAXIMUM NO OF PAGES ALLOWED
       INITIAL    XH25,12         FULL CORE THRESHOLD

* GROSS TECHNIQUE PARAMETERS
       INITIAL    XH1,2           READ BEFORE WRITE DRUM Q (FIFO)
       INITIAL    XH5,1           SHORTEST QUEUE FOR DRUM WRITES
       INITIAL    XH6,1           PREEMPTIVE CPU PRIORITY TO TASKS
       INITIAL    XH31,2          USE DELAYED POSTING OF PAGES
       INITIAL    XH32,1          USE OF LCS AS PAGING DEVICE

* DATA DEPENDENT PARAMETER
       INITIAL    XH10,20         PERCENTAGE OF UNCHANGED PAGES

* HARDWARE CONFIGURATION PARAMETERS
       INITIAL    XH3,9           NUMBER OF DRUM SLOTS
       INITIAL    XH15,80         SIZE OF CORE STORAGE

* MODEL ADJUSTMENT PARAMETERS
       INITIAL    XH11,3          FIRST COMPLETIONS REQUIRED
       INITIAL    XH17,8          NO OF INITIAL TASKS
       INITIAL    XH18,250        NO OF COMPLETIONS BEFORE RESET
       INITIAL    X4,7714         MAXIMUM TIME UNITS TO RUN
       INITIAL    XH28,20         SYSQ LENGTH REQUIRED
       INITIAL    XH29,1200       TASK OVERFLOW IN SYSQ CRITERION
```

```
 CORE  STORAGE    80                NUMBER OF CORE PAGES
 SYSTM STORAGE    100               MAXIMUM NUMBER OF TASKS IN SYSTEM
 PAGEQ STORAGE    100               NUMBER OF PAGES AWAITING TRANSFER

     1 VARIABLE   (C1@XH3)+1        CHOICE OF DRUM SECTOR
     2 FVARIABLE  RN4*(1/10)+1      CHANGED OR UNCHANGED PAGE
     3 VARIABLE   C1*389            RUN SEGMENT TIME

     1 FUNCTION   RN1,D10           UNIFORM DISTRIBUTION ON 1,2,...,9
0,1/.111111,1/.222222,2/.333333,3/.444444,4
.555555,5/.666666,6/.777777,7/.888888,8/1.0,9

     1 TABLE      P6,K1,K1,K101    TOTAL NUMBER OF PAGES IN TASK
     2 TABLE      P8,K10,K10,K116 TOTAL AMOUNT OF CPU TIME USED
     3 TABLE      P98,K1,K1,K49    TOTAL CPU INTERVALS USED
     4 TABLE      P99,K1,K1,K33    TOTAL I/O WAITS USED
     5 TABLE      P5,K1,K1,K116    LENGTHS OF CPU INTERVALS WANTED
     6 TABLE      P4,K10,K10,K116 LENGTHS OF SYNCH I/O WAITS TAKEN
     7 TABLE      M1,K1,K1,K116    LENGTHS OF READ REQUEST WAITS
     8 TABLE      M1,K1,K1,K116    LENGTHS OF WRITE REQUEST WAITS
     9 TABLE      P3,K1,K1,K101    LENGTHS OF ACTUAL CPU INTERVALS
    10 TABLE      M1,K100,K100,K116  TOTAL TIME IN THE SYSTEM
    11 TABLE      IA,K1,K1,K101    INTERARRIVAL TIME OF TASKS
    12 TABLE      P7,K1,K1,K101    NUMBER OF PAGES TASK PUTS IN CORE
    13 TABLE      IA,K100,K20,K116  INTERARRIVAL TIME TAIL
    14 QTABLE     CPUQ,K0,K5,K116 CPU QUEUE DELAY DISTRIBUTION
    15 QTABLE     COREQ,K0,K5,K116   CORE QUEUE DISTRIBUTION
```

```
* STARTUP PROCEDURE TO INTRODUCE TASKS
      GENERATE   ,,,1             CREATE INITIALIZATION XACT
      LOGIC S    PCI              POSTING OF PAGES AT FIRST
 ALLOW LOGIC S   SOURC            ALLOW A TASK TO ENTER SYSTEM
      ADVANCE    1                LET IT PERCOLATE A LITTLE
      GATE SNF   CORE             WAIT UNTIL IT CAN GET A CORE PAGE
      TEST GE    N$CNT,XH17,ALLOW   CHECK IF SPECIFIED NUMBER
      TERMINATE  ,                STOP THE INITIALIZATION

* CONTROL THE ENTRY OF TASKS TO THE SYSTEM
      JOBTAPE    JOBTA1,INPUT,0,12  LET A TASK IN NOW AND THEN
 INPUT TEST G    P1,K1,INGO       CHECK IF END OF TAPE OF NEW TASKS
      LOGIC S    EOF              NOTE THE END OF FILE ON JOBTAPE
      TERMINATE  ,                THROW AWAY THE LAST TASK ON TAPE
 INGO QUEUE      SYSQ             GET IN LINE FOR SYSTEM ENTRANCE
      GATE LS    SOURC            LET NEW TASK IN NOW AND THEN
      ENTER      SYSTM            COME IN PLEASE
      DEPART     SYSQ             AND LEAVE THE LINE
      MARK       ,                NOTE WHEN TASK ENTERS
 CNT  LOGIC R    SOURC            CLOSE SYSTEM AFTER ONE ENTRY
      TABULATE   11               SAVE INTERARRIVAL TIME IN TABLE
      TABULATE   13               WITH AN EXTRA TABLE FOR THE TAIL
      GATE LS    EOF,NOEND        PROCEED UNLESS END OF TAPE FILE
      TEST E     Q$SYSQ,K0,NOEND  PROCEED UNLESS TASK QUEUE EMPTY

* TAKE SOME NOTES AT THE END OF A RUN SEGMENT
 END  LOGIC S    STAT             MAKE SURE INITIALIZATION OVER
      LOGIC S    INIT               AND MORE TASKS CAN ENTER SYSTEM
      SAVEVALUE  13,S$SYSTM,H     COUNT CURRENT TASKS
      SAVEVALUE  1,V3             COMPUTE THE TOTAL TIME OF RUN
      SAVEVALUE  6,N$PRPR         NOTE THE NUMBER OF PAGE INS
      SAVEVALUE  12,N$OUT,H       SAVE THE NUMBER OF TERMINATIONS
      SAVEVALUE  16,N$DONE,H      AND THE TASK CPU INTERVALS COUNT
      SAVEVALUE  3,C1             NOTE THE TIME AT WINDUP
      SAVEVALUE  26+,K1,H         ADD ONE TO JOB SEGMENT NUMBER
      TERMINATE  1                STOP A RUN SEGMENT HERE

* BRING A NEW TASK INTO THE SYSTEM
 NOEND ASSIGN    9,K0             SET CPU TIME ALREADY USED TO ZERO
      ASSIGN     10,K11           SET PARAMETER POINTER
      ASSIGN     98,K0            SET INITIAL CPU USE TO ZERO
      ASSIGN     99,K0            SET INITIAL I/O WAIT USE TO ZERO
      SAVEVALUE  4,K3,H           NOTE THE ENTRY TO COMMON SECTION
```

```
* ESTABLISH THE LENGTH OF THE NEXT CPU INTERVAL
 NEXT  ASSIGN     5,P*10           GET THE CPU INTERVAL LENGTH
       TABULATE   5                ENTRY IN CPU INTERVALS TABLE
       ASSIGN     5-,P9            SUBTRACT THE AMOUNT ALREADY USED
       TEST G     P5,XH2,NXTOK     DOES TIME LEFT EXCEED ONE SLICE?
       ASSIGN     3,XH2            SET NEXT CPU INTERVAL=TIME SLICE
       ASSIGN     9+,XH2           ADD TO AMOUNT OF INTERVAL USED
       ASSIGN     5,K4             SET TSE AS REASON FOR TERMINATION
       TRANSFER   ,NXTGO           PROCEED TO LEAVE COMMON SECTION
 NXTOK ASSIGN     3,P5             SET THE CPU INTERVAL LENGTH
       ASSIGN     9,K0             CLEAR TIME FOR NEXT PARAMETER
       ASSIGN     10+,K1           MOVE TO REASON FOR TERMINATION
       ASSIGN     5,P*10           AND SET IT UP AS THE CURRENT ONE
       ASSIGN     10+,K1           MOVE TO NEXT TIME INTERVAL
 NXTGO TEST NE    XH4,K1,PRGO      BRANCH IF ENTRY FROM PAGE IN
       TEST NE    XH4,K2,CORWT     BRANCH IF ENTRY FROM I/O WAIT
       TEST G     P7,XH21,NXTIN    SEE IF MAXIMUM PAGES EXCEEDED
       TRANSFER   ,BYE             TERMINATE THIS ONE IF SO
 NXTIN ASSIGN     2,P7             SET NUMBER OF INITIAL PAGES
       SPLIT      P7,PAGEI,,K6     SEND OUT PAGE IN REQUESTS

* HOLD TASK HERE UNTIL IT GETS A CPU INTERVAL
 TASKS MATCH      DONE             SAVE THE TASK DATA WHILE AT CPU Q
       ASSIGN     98+,K1           NUMBER OF CPU INTERVALS
       ASSIGN     8+,P3            TIME USED SO FAR
       GATE LR    STAT,CKCK        IS THE INITIALIZATION OVER?

* DETERMINE THE APPROACH TO EQUILIBRIUM
       TEST GE    N$CNT,XH17,CKOUT  INITIAL READING NOT OVER
       TEST GE    N$TERM,XH18,CKIN  TERMINATIONS FOR RESET
       SPLIT      K1,END,,K1       STOP FOR THE RESET
 CKIN  TEST GE    N$TERM,XH11,CKOUT  HAS SYSTEM BEEN OPENED AGAIN?
       LOGIC S    INIT             YES, WE HAVE DONE THAT
 CKCK  GATE LS    INIT,CKOUT       ENTER ONLY IF INITIALIZATION OVER
       TEST G     R$CORE,XH25,CKOUT  IS CORE USE BELOW THRESHOLD?
       LOGIC S    SOURC            ALLOW ANOTHER TASK IN IF SO
```

```
* DECIDE WHAT TO DO WITH A TASK WHICH HAD THE CPU
 CKOUT TEST NE    P5,K3,PREQ        BRANCH IF PAGE IN
       TEST NE    P5,K2,SIN         BRANCH IF SYNCH I/O
       TEST NE    P5,K4,OUT         BRANCH IF TIME SLICE END
       TEST NE    P5,K1,OUT         BRANCH IF ASYNCH I/O
 OUT   ASSIGN     1,K2              ESTABLISH A WRITE REQUEST
       SPLIT      P7,PAGEO,,K6      SEND PAGES TO DRUM QUEUE
       TABULATE   1                 ADD ENTRY IN PAGE TABLE
       TABULATE   2                 TOTAL CPU TIME USED BY TASK
       TABULATE   3                 ADD ENTRY TO CPU INTERVALS TABLE
       TABULATE   4                 ADD ENTRY TO I/O WAITS TABLE
       TABULATE   10                TOTAL TIME TASK IN SYSTEM
       TABULATE   12                COMPUTE NUMBER OF PAGES INTO CORE
 BYE   LEAVE      SYSTM             NOTE THAT A TASK IS GONE
       TEST LE    Q$SYSQ,XH29,END    SEE IF WE FACE TASK OVERFLOW
 TERM  TERMINATE  ,                 DESTROY MASTER RECORD

* PROCESS A SYNCHRONOUS I/O INTERRUPT
 SIN   ASSIGN     99+,K1            ADD ONE TO I/O WAITS PARAMETER
       ASSIGN     4,P*10            SET LENGTH OF I/O WAIT
       ASSIGN     10+,K1            MOVE TO NEXT TIME INTERVAL
       TEST G     P*10,K0,OUT       PAGE OUT IF NO MORE DATA
       SAVEVALUE  4,K2,H            NOTE WHERE TO RETURN AFTER COMMON
       TRANSFER   ,NEXT             SET UP NEXT CPU INTERVAL

* WAIT IN CORE FOR END OF SYNCHRONOUS I/O OPERATION
 CORWT QUEUE      IOQ               ENTER I/O WAIT
       ADVANCE    P4                WAIT FOR SYNCHRONOUS I/O
       DEPART     IOQ               NOW THROUGH WITH I/O
       TABULATE   6                 ADD ENTRY TO I/O WAITS TABLE
       TEST E     XH6,K1,CORPR      I/O GIVES CPU PRIORITY?
       SPLIT      K1,READY,,K4      SEND XACT TO CPUQ
       TRANSFER   ,TASKS            WAIT FOR CPU SERVICE TERMINATION
 CORPR SPLIT      K1,PREAD,,K4      SEND XACT TO CPU PREEMPTIVELY
       TRANSFER   ,TASKS            AND WAIT UNTIL SERVICE OVER
```

```
* PROCESS A PAGE IN REQUEST
 PREQ  GATE SF     CORE,PRPR        PROCEED IF THERE IS ROOM IN CORE
       TEST GE     Q$COREQ,XH20,PRPR  CHECK FOR QUEUE OVERFLOW
       SAVEVALUE   8+,K1,H          COUNT THE TIMES THIS HAPPENS
       TRANSFER    ,OUT             AND TERMINATE THIS TASK
 PRPR  TEST G      P7,XH21,PRON     PAGE OUT TASK WHICH EXCEEDS SIZE
       SAVEVALUE   24+,K1,H         MAKE A NOTE OF THIS EVENT
       TRANSFER    ,OUT             AND TERMINATE THE RESULT
 PRON  ASSIGN      7+,K1            NUMBER OF PAGES IN CORE
       SAVEVALUE   4,K1,H           NOTE WHERE TO RETURN AFTER COMMON
       TRANSFER    ,NEXT            SET UP NEXT CPU INTERVAL
 PRGO  ASSIGN      1,K1             SET READ REQUEST PARAMETER
       TEST G      XH1,K2,PRFF      BRANCH IF NOT PRIORITY READ
       ASSIGN      1,K0             SET REQUEST TO ZERO
       ASSIGN      1-,P7            SET PRIORITY OF READ REQUEST
 PRFF  ASSIGN      2,K2             SET NUMBER OF PAGES TO ASSEMBLE
       TEST G      P7,P6,PRRP       SEE IF OVER TOTAL PAGE COUNT
       ASSIGN      6,P7             SET NEW COUNT IF SO
 PRRP  SPLIT       K1,PAGEI,,K6     SEND PAGE REQUEST TO DRUM
       SPLIT       K1,IPAGE,,K4     WAIT FOR THE PAGE
       TRANSFER    ,TASKS           WAIT FOR CPU SERVICE TERMINATION

* SET UP A PAGE IN REQUEST
 PAGEI QUEUE       COREQ            WAIT FOR A FREE CORE PAGE
       ENTER       CORE             ALLOCATE THE CORE PAGE
       DEPART      COREQ            THROUGH WAITING FOR CORE
 RPAGE SAVEVALUE   9,FN1,H          CHOOSE A DRUM SLOT AT RANDOM

* ENQUEUE A PAGE TRANSFER REQUEST
 PAGE  MARK                         NOTE TIME PAGE GOES IN
 PAGE4 ENTER       PAGEQ            NOTE NEW PAGE IN QUEUES
       TEST NE     XH32,K2,PAGE3    SEE IF LCS PAGING
       TEST L      XH1,K2,PAGE2     CHOOSE DRUM QUEUE DISCIPLINE
 PAGE1 LINK        XH9,FIFO           FIFO DISCIPLINE
 PAGE2 LINK        XH9,P1             READ BEFORE WRITE DISCIPLINE
 PAGE3 LINK        K10,FIFO           LCS INSTEAD OF DRUM
```

```
* SET UP A PAGE OUT REQUEST
 PAGEO TEST G      V2,XH10,PDONE   CHECK IF AN UNCHANGED PAGE
       TEST NE     XH32,K2,PAGE4   SEE IF LCS PAGING
       TEST NE     XH5,K2,RPAGE    CHOOSE WRITE Q CHOICE RULE
       ASSIGN      2,K2            SET COUNTER TO TWO
       ASSIGN      3,CH1           SET VALUE TO FIRST CHAIN LENGTH
       ASSIGN      4,K1            SET FIRST CHAIN NUMBER TO ONE
 PPBK  TEST L      CH*2,P3,POUT    SEE IF CURRENT CHAIN IS SHORTER
       ASSIGN      3,CH*2          IF SO SAVE THE LENGTH
       ASSIGN      4,P2            AND THE CHAIN NUMBER
 POUT  TEST L      P2,XH3,PPGO     SEE IF WE ARE AT END OF CHAINS
       ASSIGN      2+,K1           MOVE TO THE NEXT CHAIN IF NOT
       TRANSFER    ,PPBK           AND PROCEED
 PPGO  SAVEVALUE   9,P4,H          SAVE THE RIGHT CHAIN NUMBER
       TRANSFER    ,PAGE           AND PROCEED TO SEND IT OUT

* SELECT NEXT REQUEST ON APPROPRIATE SLOT QUEUE
       GENERATE    ,,,1,,1,F       GET TIMING PULSE
       TEST NE     XH32,K2,TIME2   SEE IF LCS PAGING
 TIMER UNLINK      V1,CHANL,1      LOOK FOR REQUEST IN SLOT QUEUE
       ADVANCE     1               HOLD TIMER BACK
       TRANSFER    ,TIMER          AND LET THE CLOCK TICK AGAIN
 TIME2 UNLINK      K10,CHANL,1     GET PAGE REQUEST FROM LCS SLOT
       ADVANCE     1               HOLD TIMER BACK
       TRANSFER    ,TIME2          AND LET THE CLOCK TICK AGAIN

* CONTROL PAGE POSTING AND MAXIMUM RUN LENGTH
       GENERATE    ,,,1,,1,F       GET TIMING PULSE
 POST  LOGIC S     PCI             RELEASE PAGES
       PRIORITY    K0,BUFFER       LET THEM BE USED
       LOGIC R     PCI             AND CLOSE THE DOOR AGAIN
       TEST GE     C1,X4,TNEXT     ARE WE OUT OF TIME?
       SPLIT       K1,END          SEND OUT A TERMINATION IF SO
 TNEXT ADVANCE     9               HOLD TIMER BACK
       TRANSFER    ,POST           AND THEN POST AGAIN
```

```
* SERVICE AND DISPOSE OF PAGE TRANSFER REQUEST
 CHANL SEIZE      DRUM              HERE GOES THE ACTUAL TRANSFER
       LEAVE      PAGEQ             NOTE THAT WE ARE OUT
       ADVANCE    1                 FOR ONE TIME UNIT
       RELEASE    DRUM              AND LET THE NEXT ONE HAVE IT
       TEST E     XH31,K2,PCICK     SEE IF WE ARE DELAYING POSTING
       GATE LS    PCI               WAIT HERE IF SO
 PCICK TEST G     P1,K1,IPGIN       BRANCH IF PAGE IN REQUEST
       TABULATE   8                 OTHERWISE NOTE THE WRITE WAIT
 PDONE LEAVE      CORE              THEN RELEASE THE CORE SPACE
       GATE LS    INIT,GONE         SEE IF INITIALIZATION IS OVER
       TEST G     R$CORE,XH25,GONE   CORE USE BELOW THRESHOLD?
       LOGIC S    SOURC             LET ANOTHER TASK IN IF SO
 GONE  TERMINATE  ,                 AND DISCARD THE PAGES INVOLVED
 IPGIN TABULATE   7                 ADD ENTRY TO PAGE READ WAIT TABLE

* ASSEMBLE PAGES AND WAIT TO OBTAIN CPU SERVICE
 IPAGE ASSEMBLE   P2                COLLECT PAGES FOR TASK
 READY QUEUE      CPUQ              WAIT FOR CPU
       SEIZE      CPU               OBTAIN CPU SERVICE
       DEPART     CPUQ              STOP WAITING
       ADVANCE    P3                CPU SERVICE INTERVAL
       RELEASE    CPU               RELINQUISH CPU SERVICE
 EREAD PRIORITY   K0                MAKE SURE ALL TASKS ARE NOW EQUAL
       TABULATE   9                 ADD ENTRY TO CPU INTERVALS TABLE
 DONE  MATCH      TASKS             FIND TASK INFORMATION
       TERMINATE  ,                 DISCARD CPU REQUEST

* PREEMPT CPU TASK FOR SERVICE
 PREAD SAVEVALUE  7+,K1,H           SET PRIORITY FOR PREEMPTION
       PRIORITY   XH7               AND MAKE IT PRIORITY OF TASK
       PREEMPT    CPU,PR            NOW GET THE CPU
       ADVANCE    P3                AND USE IT
       RETURN     CPU               BEFORE GIVING IT BACK
       SAVEVALUE  7-,K1,H           REDUCE PREEMPTION LEVEL
       TRANSFER   ,EREAD            AND CONTINUE AS IF NORMAL SERVICE
       INITIAL    XH24,0
       INITIAL    XH8,0
       START      1

       END
```

Appendix F: The Simulation Results

This section gives the data obtained from GPSS/360 simulations made with the model described in the previous appendix. The results of these runs are discussed in Chapter 4. A number of simulation runs were made using a pair of JOBTAPEs of task transactions from the MTS data. These tapes were obtained by abstracting data sets # 0 and # 1 during the analysis of that data for the presentation in Appendix D. Each simulation run begins at the front of one of these two tapes and uses as many transactions as necessary in order to simulate system operation for a specified amount of elapsed time.

For initial runs, both JOBTAPEs were run with a pair of "extremal" choices of the model parameters:

(a) a Basic configuration of hardware and programming techniques, using

80 core pages
a 9-sector drum
16 time unit time-slice
read before write drum queue discipline
drum writes to shortest queue
FIFO CPU queue discipline
posting of pages once per revolution

(b) an Advanced configuration, which differs from the Basic one in the following choices of parameter values

144 core pages
large core storage instead of drum
24 time unit time-slice
priority in drum queues to large tasks
immediate posting of pages

The remaining runs were made using parameter choices differing from Basic in only one or two parameter values, and generally with values chosen from the Advanced model substituted for the Basic values.

Each run begins with an initialization period, after which two or three sets of data are taken for intervals of 30 simulated seconds. Because the initialization intervals were taken to be rather long, their data (which is not included here) agrees quite closely with the values observed for the regular intervals. The Basic and Advanced models were each run for three intervals, and the remaining models for two. In several cases, however, runs were terminated after a somewhat shorter last interval because of a problem with controlling the input rate of task data from the JOBTAPEs. Thus the data displayed in the figures of this section is normalized by the length of the run segment.

Figures F-1 and F-2 show the results of running the Basic model with the MTS # 1 and MTS # 0 JOBTAPEs, respectively. The same data for the Advanced model is given in Figures F-3 and F-4. Figure F-5 lists the values obtained by using the Basic model except for the larger core size taken for the Advanced model. Similarly, Figure F-6 gives the data for the Basic configuration altered only by immediate page posting. Subsequent figures show other variations. Only one run (described in Figure F-9) was made with values deliberately chosen outside the techniques used in the Basic and Advanced models. In this case several poorer techniques were used to gauge the sensitivity of paging drum processing to such changes.

Figure F-11, which was run with a 4-sector drum in the (otherwise) Basic configuration, also provides for a fraction of unchanged pages: 20% of the write requests were not executed in this case, assuming that the pages were unchanged since their last trip to the drum.

BASIC SIMULATION MODEL

MTS Data # 1

PERFORMANCE...	I	II	III
• CPU Utilization	.456	.436	.591
Average queue contents	.601	.508	.767
Maximum queue contents	9	7	6
Percent zero entries	53.6	55.3	40.7
• Paging mechanism utilization	.499	.685	.672
Average queue contents	12.5	22.1	19.4
Maximum queue contents	69	69	65
Mean completion time	25.1	32.0	28.9
Tasks Completed/Second	10.5	12.8	16.4
Mean No. of Active Tasks	8.8	7.1	9.2
WORKLOAD	**I**	**II**	**III**
Task No. Range	250-563	564-949	950-1319
No. of CPU Intervals/Second	52.4	43.2	63.8
Mean CPU Service Time	2.24	2.60	2.38
Percent Initial Pages	86.4	89.7	82.3
No. of Pages Written/Second	64.2	88.3	86.2
Mean No. of I/O Operations	5.49	3.38	5.20
Average I/O Time	42.3	41.3	41.6

Figure F-1. Basic Simulation Data

MTS Data # 0

PERFORMANCE...	I	II	III
• CPU Utilization	.253	.311	.189
Average queue contents	.304	.261	.136
Maximum queue contents	8	6	5
Percent zero entries	37.6	51.0	55.3
• Paging mechanism utilization	.439	.401	.263
Average queue contents	11.0	5.56	3.48
Maximum queue contents	71	47	48
Mean completion time	24.6	13.8	13.2
Tasks Completed/Second	11.0	7.8	4.9
Mean No. of Active Tasks	7.5	8.7	6.0
WORKLOAD	**I**	**II**	**III**
Task No. Range	252-581	582-815	816-901
No. of CPU Intervals/Second	47.7	62.8	36.5
Mean CPU Service Time	1.37	1.28	1.34
Percent Initial Pages	51.3	34.2	34.9
No. of Pages Written/Second	56.5	51.7	33.4
Mean No. of I/O Operations	3.73	5.55	3.24
Average I/O Time	102.4	68.4	82.5

Figure F-2. Basic Simulation Data

ADVANCED SIMULATION MODEL

MTS Data # 1

PERFORMANCE...	I	II	III
• CPU Utilization	.717	.621	.856
Average queue contents	2.19	2.14	5.08
Maximum queue contents	16	20	16
Percent zero entries	28.9	34.9	15.2
• Paging mechanism utilization	.804	.779	.721
Average queue contents	32.9	36.8	21.1
Maximum queue contents	≥100	≥100	95
Mean completion time	40.2	47.2	29.3
Tasks Completed/Second	13.7	13.8	16.2
Mean No. of Active Tasks	14.5	12.9	18.3
WORKLOAD	**I**	**II**	**III**
Task No. Range	251-660	661-1076	1077-1483
No. of CPU Intervals/Second	81.5	64.8	100.6
Mean CPU Service Time	2.26	2.47	2.19
Percent Initial Pages	78.2	76.1	74.6
No. of Pages Written/Second	104.5	99.2	92.3
Mean No. of I/O Operations	6.69	3.98	7.60
Average I/O Time	38.1	37.5	32.0

Figure F-3. Advanced Simulation Data

MTS Data # 0

PERFORMANCE...	I	II	III
• CPU Utilization	.642	.630	.555
Average queue contents	1.55	1.79	1.74
Maximum queue contents	16	17	17
Percent zero entries	36.3	37.7	40.2
• Paging mechanism utilization	.900	.881	.957
Average queue contents	25.4	29.4	45.2
Maximum queue contents	96	≥100	≥100
Mean completion time	28.0	33.2	47.2
Tasks Completed/Second	21.1	16.9	20.5
Mean No. of Active Tasks	18.5	15.7	17.9
WORKLOAD	**I**	**II**	**III**
Task No. Range	251-883	884-1391	1392-2007
No. of CPU Intervals/Second	114.7	100.5	88.7
Mean CPU Service Time	1.44	1.61	1.61
Percent Initial Pages	46.6	48.4	60.3
No. of Pages Written/Second	116.2	112.5	123.1
Mean No. of I/O Operations	7.32	3.69	2.48
Average I/O Time	59.2	37.0	32.9

Figure F-4. Advanced Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using 144 core pages)
MTS Data # 1

PERFORMANCE...	I	II
• CPU Utilization	.601	.707
Average queue contents	1.78	4.08
Maximum queue contents	15	17
Percent zero entries	38.0	21.8
• Paging mechanism utilization	.854	.856
Average queue contents	49.3	56.2
Maximum queue contents	≥100	≥100
Mean completion time	56.8	65.5
Tasks Completed/Second	17.6	20.0
Mean No. of Active Tasks	12.8	13.3
WORKLOAD	**I**	**II**
Task No. Range	250-776	777-1378
No. of CPU Intervals/Second	68.1	72.2
Mean CPU Service Time	2.27	2.52
Percent Initial Pages	91.0	89.2
No. of Pages Written/Second	110.3	110.2
Mean No. of I/O Operations	6.78	4.79
Average I/O Time	42.9	30.5

Figure F-5. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using immediate page posting)
MTS Data # 0

PERFORMANCE...	I	II
• CPU Utilization	.484	.310
Average queue contents	.250	.099
Maximum queue contents	6	5
Percent zero entries	73.9	82.8
• Paging mechanism utilization	.705	.387
Average queue contents	16.6	5.96
Maximum queue contents	69	55
Mean completion time	23.5	15.4
Tasks Completed/Second	14.0	6.1
Mean No. of Active Tasks	7.7	7.6
WORKLOAD	**I**	**II**
Task No. Range	201-620	621-805
No. of CPU Intervals/Second	98.3	60.1
Mean CPU Service Time	1.44	1.33
Percent Initial Pages	44.3	28.9
No. of Pages Written/Second	90.7	50.0
Mean No. of I/O Operations	4.41	5.05
Average I/O Time	51.9	69.3

Figure F-6. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using LCS for paging mechanism)
MTS Data # 0

PERFORMANCE...	I	II
• CPU Utilization	.362	.202
Average queue contents	.562	.332
Maximum queue contents	10	10
Percent zero entries	36.2	39.8
• Paging mechanism utilization	.598	.369
Average queue contents	10.8	7.15
Maximum queue contents	59	62
Mean completion time	18.0	19.3
Tasks Completed/Second	16.9	9.8
Mean No. of Active Tasks	12.6	7.3
WORKLCAD	**I**	**II**
Task No. Range	251-758	759-1052
No. of CPU Intervals/Second	72.7	37.6
Mean CPU Service Time	1.28	1.39
Percent Initial Pages	53.5	61.6
No. of Pages Written/Second	77.0	47.5
Mean No. of I/O Operations	7.42	3.24
Average I/O Time	94.5	85.2

Figure F-7. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using preemptive CPU queue discipline)
MTS Data # 1

PERFORMANCE...	I	II
• CPU Utilization	.425	.407
Average queue contents	.339	.347
Maximum queue contents	5	8
Percent zero entries	53.0	53.3
• Paging mechanism utilization	.513	.709
Average queue contents	13.5	22.9
Maximum queue contents	68	74
Mean completion time	26.4	32.1
Tasks Completed/Second	10.8	13.0
Mean No. of Active Tasks	8.7	7.0
WORKLOAD	**I**	**II**
Task No. Range	250-573	574-963
No. of CPU Intervals/Second	54.4	42.8
Mean CPU Service T'me	2.01	2.45
Percent Initial Pages	85.4	89.3
No. of Pages Written/Second	66.2	91.4
Mean No. of I/O Operations	5.28	3.30
Average I/O Time	39.8	41.9

Figure F-8. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using random drum writes and FIFO queues)
MTS Data # 1

PERFORMANCE...	I	II
• CPU Utilization	.410	.291
Average queue contents	.455	.154
Maximum queue contents	9	5
Percent zero entries	59.7	65.9
• Paging mechanism utilization	.493	.515
Average queue contents	13.2	15.2
Maximum queue contents	62	61
Mean completion time	26.7	29.4
Tasks Completed/Second	9.7	8.9
Mean No. of Active Tasks	9.1	7.3
WORKLOAD	**I**	**II**
Task No. Range	250-541	542-808
No. of CPU Intervals/Second	52.3	37.8
Mean CPU Service Time	2.02	1.98
Percent Initial Pages	82.9	80.7
No. of Pages Written/Second	63.3	66.5
Mean No. of I/O Operations	4.88	3.24
Average I/O Time	39.3	51.4

Figure F-9. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using queueing options and longer time slice)
MTS Data # 0

PERFORMANCE...	I	II
• CPU Utilization	.324	.238
Average queue contents	.533	.180
Maximum queue contents	10	5
Percent zero entries	33.9	42.2
• Paging mechanism utilization	.478	.296
Average queue contents	9.56	3.72
Maximum queue contents	67	39
Mean completion time	20.0	12.6
Tasks Completed/Second	11.2	5.9
Mean No. of Active Tasks	7.7	7.8
WORKLOAD	**I**	**II**
Task No. Range	251-587	588-765
No. of CPU Intervals/Second	55.1	47.9
Mean CPU Service Time	1.51	1.28
Percent Initial Pages	47.6	31.3
No. of Pages Written/Second	61.7	38.1
Mean No. of I/O Operations	3.68	4.98
Average I/O Time	80.1	79.8

Figure F-10. Modified Simulation Data

MODIFIED BASIC SIMULATION MODEL

(Using 4-sector drum and 20% unchanged pages)
MTS Data # 0

PERFORMANCE...	I	II
• CPU Utilization	.426	.350
Average queue contents	.475	.323
Maximum queue contents	6	7
Percent zero entries	30.6	36.9
• Paging mechanism utilization	.574	.458
Average queue contents	11.2	6.08
Maximum queue contents	70	57
Mean completion time	19.5	13.3
Tasks Completed/Second	13.9	9.7
Mean No. of Active Tasks	9.2	8.2
WORKLOAD	**I**	**II**
Task No. Range	253-668	669-959
No. of CPU Intervals/Second	81.4	68.2
Mean CPU Service Time	1.35	1.32
Percent Initial Pages	63.4	62.2
No. of Pages Written/Second	64.6	52.7
Mean No. of I/O Operations	5.32	5.05
Average I/O Time	45.4	67.4

Figure F-11. Modified Simulation Data

BIBLIOGRAPHY

1. Amdahl, G. M. "Effects of Certain Parameters on Time-Sharing Systems." Geophysical Theory and Computers. Edited by C. L. Pekeris. Oxford: Blackwell Scientific Publishers, 1966.

2. Amdahl, G. M., and Behman, S. B. "A Simulation of the Effects of Dynamic Storage Allocation Hardware and Scheduling Algorithms on Time Sharing Systems." Unpublished paper, 1966.

3. Arden, B. W. "Time-Sharing Systems: A Review." Proceedings of the IEEE International Convention, 15 Part 10 (March, 1967) 23-25.

4. Arden, B. W., Galler, B. A., O'Brien, T. C., and Westervelt, F. H. "Program and Addressing Structure in a Time-Sharing Environment." Journal of the ACM, 13 (January, 1966) 1-16.

5. Belady, L. A. "A Study of Replacement Algorithms for a Virtual-Storage Computer." IBM Systems Journal, 5 (July, 1966) 78-101.

6. Bryan, G. E. "JOSS: 20,000 Hours at a Console—A Statistical Summary." Proceedings of the Fall Joint Computer Conference, 31 (November, 1967) 769-778.

7. Codd, E. F. "Multiprogram Scheduling." Communications of the ACM, 3 (June, 1960) 347-350.

8. Coffman, E. G. "Stochastic Models of Multiple and Time-Shared Computer Operations." Report No. 66-38, Department of Engineering, UCLA. 1966.

9. Coffman, E. G. and Wood, R. C. "Interarrival Statistics for Time-Sharing Systems." Communications of the ACM, 9 (July, 1966) 500-503.

10. Coffman, E. G. "A Simple Probability Model Yielding Performance Bounds for Modular Memory Systems." Unpublished Note, June, 1967.

11. Coffman, E. G., and Varian, L. C. "An Empirical Study of the Behavior of Programs in a Paging Environment." _Proceedings of the ACM Symposium on Operating System Principles_, Gatlinburg, Tennessee, October, 1967.

12. Cohen, J. "A Use of Fast and Slow Memories in List-Processing Languages." _Communications of the ACM_, _10_ (February, 1967) 82-86.

13. Comeau, L. W. "A Study of the Effect of User Program Optimization on a Paging System." _Proceedings of the ACM Symposium on Operating System Principles_, Gatlinburg, Tennesee, October, 1967.

14. Corbato, F. J., and Vyssotsky, V. A. "Introduction and Overview of the Multics System." _Proceedings of the Fall Joint Computer Conference_, _27_ (November, 1965) 185-196.

15. Cox, D. R. and Smith, W. L. _Queues_. London: Methuen and Co., Ltd., 1961.

16. Denning, P. J. "Effects of Scheduling on File Memory Operations." _Proceedings of the Spring Joint Computer Conference_. _30_ (April, 1967) 9-21.

17. Denning, P. J. "The Working Set Model for Program Behavior." _Proceedings of the ACM Symposium on Operating System Principles_, Gatlinburg, Tennesee, October, 1967.

18. Denning, P. J. Review Number 12,531. _Computing Reviews 8_ (July, 1967) 393-394.

19. Dennis, J. B. "Segmentation and the Design of Multiprogrammed Computer Systems." _Journal of the ACM_, _12_ (October, 1965) 589-602.

20. Dennis, J. B., and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations." _Communications of the ACM_, _9_ (October, 1966) 143-155.

21. Edwards, D., Kilburn, T., Lanigan, M., and Sumner, F. "One Level Storage System." _IRE Transactions on Electronic Computers_, _EC-11_ (April, 1962) 223-235.

22. Estrin, G., and Martin, D. "Models of Computations and Systems—Evaluation of Vertex Probabilities in Graph Models of Computations." *Journal of the ACM*, *14* (April, 1967) 281-299.

23. Estrin, G. and Turn, R. "Automatic Assignment of Computations in a Variable Structure Computer System." *IEEE Transactions on Electronic Computers*, *EC-12* (December, 1963) 755-773.

24. Evans, D. C., and LeClerc, J. Y. "Address Mapping and the Control of Access in an Interactive Computer." *Proceedings of the Spring Joint Computer Conference*, *30* (April, 1967) 23-30.

25. Feller, W. *An Introduction to Probability Theory and Its Applications*. I. 2nd Ed. New York: Wiley and Sons, 1957.

26. Feller, W. *An Introduction to Probability Theory and Its Applications*. II. New York: Wiley and Sons, 1965.

27. Fife, D. W. "The Optimal Control of Queues, with Application to Computer Systems." *Technical Report 170*. Ann Arbor: Cooley Electronics Laboratory, University of Michigan, 1965.

28. Fine, G. H., Jackson, C. W., and McIsaac, P. V. "Dynamic Program Behavior under Paging." *Proceedings of the ACM 21st National Conference*, (August, 1966) 223-228.

29. Fine, G. H., and McIsaac, P. V. "Simulation of a Time-Sharing System." *Professional Paper SP-1909*. Santa Monica: System Development Corporation, 1964.

30. Gaver, D. P. Jr. "Probability Models for Multiprogramming Computer Systems." *Journal of the ACM*, *14* (July, 1967) 423-438.

31. Gibson, C. T. "Time-Sharing with the IBM System/360: Model 67." *Proceedings of the Spring Joint Computer Conference*, *28* (April, 1966) 61-78.

32. Hammersley, J. M., and Handscomb, D. C. *Monte Carlo Methods*. New York: Wiley and Sons, 1964.

33. Howard, R. *Dynamic Programming and Markov Processes*. Cambridge: MIT Press, 1960.

34. International Business Machines. GPSS/360 User's Manual. IBM Publication H20-0326.

35. International Business Machines. IBM System/360 Model 67: Time-Sharing System Functional Characteristics. IBM Publication A27-2719.

36. Ivanescu, P. L. "Pseudo-Boolean Programming and Applications." Lecture Notes in Mathematics, 9. Berlin: Springer-Verlag, 1965.

37. Ivanescu, P. L., and Rudeanu, S. "Pseudo-Boolean Methods for Bivalent Programming." Lecture Notes in Mathematics, 23. Berlin: Springer-Verlag, 1966.

38. Karp, R. M. "A Note on the Application of Graph Theory to Digital Computer Programming." Information and Control, 3 (September, 1960) 179-190.

39. Karp, R. M. Some Applications of Logical Syntax to Digital Computer Programming. Ph.D. Thesis, Department of Engineering, Harvard University, Cambridge, 1959.

40. Kemeny, J., and Snell, J. Finite Markov Chains. Princeton: D. Van Nostrand Co., 1960.

41. Kleinrock, L. "A Conservation Law for a Wide Class of Queueing Disciplines." Naval Research Logistics Quarterly, 12 (June, 1965) 181-192.

42. Kleinrock, L. "Time-Shared Systems: A Theoretical Treatment." Journal of the ACM, 14 (April,1967) 242-261.

43. Koenigsberg, E. "Cyclic Queues." Operational Research Quarterly, 9 (January, 1958) 22-35.

44. Krider, L. "A Flow Analysis Algorithm." Journal of the ACM, 11 (October, 1964) 429-436.

45. Lauer, H. C. "Bulk Core in a 360/67 Time-Sharing System." Proceedings of the Fall Joint Computer Conference, 31 (November, 1967) 601-610.

46. Livermore, F. G. "A General Approach to Time-Sharing; Algorithms for Scheduling and Control of Computer Resources." Research Publication GMR 549, General Motors Corporation. 1966.

47. McGee, W. C. "On Dynamic Program Relocation." _IBM Systems Journal_, _4_ (July, 1965) 181-199.

48. Marimont, R. B. "Applications of Graphs and Boolean Matrices to Computer Programming." _SIAM Review_, _2_ (October, 1960) 259.

49. Medgyessy, P. _Decomposition of Superpositions of Distribution Functions_. Budapest: Hungarian Academy of Sciences, 1961.

50. Morris, D., Sumner, F. H., and Wyld, M. T. "An Appraisal of the Atlas Supervisor." _Proceedings of the ACM 22nd National Conference_, (August, 1967) 67-75.

51. University of Michigan Computing Center. _MTS: Michigan Terminal System_. Ann Arbor: University Press, 1968.

52. Nielsen, N. R. "The Analysis of General Purpose Computer Time-Sharing Systems." _Document No. 40-10-1_. Stanford: Computation Center, 1966.

53. Pankhurst, R. J. "Program Overlay Techniques." _Communications of the ACM_, _11_ (February, 1968) 119-125.

54. Pinkerton, T. B. "On the Automatic Computation of Integral Homology Groups." _Mathematical Algorithms_, _I_ (January, 1966) 36.

55. Prossner, R. T. "Applications of Boolean Matrices to the Analysis of Flow Diagrams." _Proceedings of the Eastern Joint Computer Conference_, (December, 1959) 133.

56. Ramamoorthy, C. V. "Discrete System Representation and Analysis by Generating Functions of Abstract Graphs." _IEEE International Convention Record_, _13_ Part 6. 1965.

57. Ramamoorthy, C. V. "The Analytic Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers." _Proceedings of the ACM 21st National Conference_, (August, 1966) 229-239.

58. Rosenberg, R. S., and Wallace, V. L. "RQA-1, The Recursive Queue Analyzer." _Technical Report 2_. Ann Arbor: Systems Engineering Laboratory, University of Michigan, Ann Arbor, 1965.

59. Saltzer, J. H. "CTSS Technical Notes." _Report MAC-TR-16_. Cambridge: Project MAC, 1965.

60. Salwicki, A. "On a Certain Theorem of Graph Theory and Its Application to Automatic Programming." _Algorytmy_, _4_ (December, 1965) 69-83.

61. Seaman, P. H. "On Teleprocessing System Design, Part IV: The Role of Digital Simulation." _IBM Systems Journal_, _5_ (September, 1966) 175-189.

62. Scherr, A. L. "An Analysis of Time-Shared Computer Systems". _Report MAC-TR-18_. Cambridge: Project MAC, 1965.

63. Schulman, F. D. "Hardware Measurement Device for IBM System/360 Time-Sharing Evaluation." _Proceedings of the ACM 22nd National Conference_, (August, 1967) 103-109.

64. Schurmann, A. "The Application of Graphs to the Analysis of Distribution of Loops in a Program." _Information and Control_, _7_ (September, 1964) 275-282.

65. Shemer, J. E. "Some Mathematical Considerations of Time-Shared Scheduling Algorithms." _Journal of the ACM_, _14_ (April, 1967) 262-272.

66. Shemer, J. E., and Shippey, G. A. "Statistical Analysis of Paged and Segmented Computer Systems." _IEEE Transactions on Electronic Computers_, _EC-15_ (December, 1966) 855-863.

67. Smith, J. L. "Markov Decisions in a Partitioned State Space, and the Control of Multiprogramming". _Technical Report 9_. Ann Arbor: Systems Engineering Laboratory, University of Michigan, 1967.

68. Smith, J. L. "Multiprogramming under a Page on Demand Strategy." _Communications of the ACM_, _10_ (October, 1967) 636-646.

69. Tukey, J. W., and Wilk, M. B. "Data Analysis and Statistics: An Expository Overview." *Proceedings of the Fall Joint Computer Conference*, *29* (November, 1966) 695-709.

70. University of Michigan Computing Center. *University of Michigan Executive System for the IBM 7090 Computer*. Ann Arbor: University Press, 1966.

71. Varga, R. S. *Matrix Iterative Analysis*. Englewood Cliffs: Prentice-Hall, Inc., 1962.

72. Wonderly, R. A. *A Segmenting Model for Digital Computer Programs*. Master's Thesis, Department of Mathematics, University of North Carolina, Chapel Hill, 1961.

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* The University of Michigan CONCOMP Project	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE
Program Behavior and Control in Virual Storage Computer Systems

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report

5. AUTHOR(S) *(Last name, first name, initial)*
Tad Brian Pinkerton

6. REPORT DATE April 1968	7a. TOTAL NO. OF PAGES 160	7b. NO. OF REFS 72

8a. CONTRACT OR GRANT NO. DA-49-083 OSA-3050 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 4 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
Qualified Requesters may obtain copies of this Report from DDC

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This study attempts to resolve some problems in the allocation of storage to computer programs. In both large and small systems, it is necessary to periodically transfer parts of programs between main and auxiliary storage devices. Several stochastic models are used to represent the behavior of programs with respect to their use of storage. These models are used to suggest ways in which main storage should be allocated in order to maximize the efficiency of the operating system. Both packaging the parts of a program into memory units and scheduling the storage assignment events during execution are considered as optimization problems. A detailed set of data from a particular large timesharing system was taken to illuminate the storage use characteristics of the system load and provide parameter values for the theoretical models. In addition, these data were used directly in a simulation study of storage management techniques which served to bridge the gap between simple analytical models and the complexities of a real system.

DD FORM 1 JAN 64 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Storage Allocation Virtual Memory Simulation Scheduling Multiprogramming						

Unclassified
Security Classification