THE UNIVERSITY OF MICHIGAN

SYSTEMS ENGINEERING LABORATORY

Department of Electrical Engineering
College of Engineering

SEL Technical Report No. 52

SOLUTION OF MARKOV RENEWAL DECISION PROCESSES WITH APPLICATION TO COMPUTER SYSTEM SCHEDULING

by

John Wesley Boyse

under the direction of
Professor Keki B. Irani

June 1971

under contract with:

ROME AIR DEVELOPMENT C
Research and Technology Di
Griffiss Air Force Base, New York
Contract No. F30602-69-C-0214

en sn

UMRØ533

ACKNOWLEDGEMENTS

During the course of this research, help was received from many people. Professor K. B. Irani and the other members of my doctoral committee deserve special mention in this regard.

Needed financial support was provided by Rome Air Development Center (Contract No. F30602-69-C-0214).

Finally, I thank my wife, Georgia, for her help in preparing the manuscript as well as for her support and encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF TABLES (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

ABSTRACT

In multiprogrammed computer systems, the way in which system resources are allocated among tasks can strongly influence system efficiency. In such systems, as opposed to monoprogrammed computers, separate but interdependent consideration must be given to scheduling of two critical system resources:main memory and the central processors. A rather wide class of scheduling procedures is available to the designer of such a system, and so it is useful to be able to determine task priority rules which allow the maximum number of task completions per unit time or throughput. This research pursues the theoretical development and application of Markov renewal decision processes for this purpose.

A major effort is devoted to theoretical development of a new technique for finding the optimal stationary policy in a class of infinite time Markov renewal decision processes in which the criterion is either maximization of expected return per transition or maximization of expected return per unit time. The modeling of stochastic service systems with Markov renewal decision processes commonly leads to transition matrices which are sparse, and the derived optimization method offers definite computational advantages for this wide class of practical problems.

A Markov process is used to model a problem of current import in the software design of multiprogrammed computer systems; namely, the problem of assignment of central processors and main memory to those jobs requesting them. Application of the optimization method to this model allows the determination of scheduling rules which maximize throughput. An additional complexity in scheduling arises when tasks are allowed to share information with other tasks in main memory, and this problem is treated also. Parameters which may be varied in the model include the size of the main memory relative to the sizes of individual tasks, the execution characteristics of tasks themselves, the number of central processors, and the fraction of its total memory requirement each task may share with other tasks. Paging is important in many modern computer systems and is treated indirectly in the model by treating it as a special kind of input/output operation. Results for a range of values of parameters are presented which compare the throughput possible under optimal scheduling rules and under a good heuristic scheduling rule.

In order to obtain results applicable to a typical general purpose multiprogrammed computer system, data from such a system were collected and analyzed in order to set appropriate values for parameters of the model and also to test the validity of certain assumptions made in the model. These data are general enough to be of interest in their own right and are therefore presented in the form of graphs and tables as part of this research effort.

Chapter 1

INTRODUCTION

The work reported here is organized around the problem of determining how much the efficiency of multiprogrammed computer systems may be increased by altering the scheduling rules in these systems and by the use of shared information (e.g. reentrant procedure). In particular, the efficiency is proportional to the throughput (number of jobs processed per unit of time), and it is this quantity we determine as algorithms for scheduling these jobs are varied. Also investigated is the change in throughput when jobs are allowed to share information with one another in the execution store of the computer.

The effort expended in attacking the above problem may be divided into three major categories.

1) A suitable mathematical model was developed to allow comparison of scheduling rules and investigate the effect of shared information.

2) Derivation of an optimization technique was carried out which can be used to find scheduling algorithms which maximize throughput. This optimization method is applicable to a broader class of problems than the one presented in this thesis.

3) Collection and analysis of data from a large multiprogrammed computer system were carried out so that suitable values

could be given parameters of the mathematical model. An ancillary benefit of this effort was validation of some assumptions made in the mathematical model.

In the following sections the work outlined above is placed in proper context for consideration in detail in other chapters. The problem of modelling a multiprogrammed computer system is placed in the milieu of current systems of this type, and then some previous computer models are reviewed and possible methods of modelling such systems are discussed. The computer model developed in this report is a Markovian queuing model and so for completeness pertinent definitions from queuing and Markov renewal theory are presented. The final section is a brief discussion of computer performance data.

1.1 The Computer System

In this work we are primarily concerned with characterizing multiprogrammed computer systems. Multiprogramming is the simultaneous residence of more than a single program or task in the execution store (core) of the computer. Many modern computing systems operate in this mode in order to make more efficient use of system resources. In older monoprogrammed systems it was found that often the central processing unit (CPU) was idle a significant portion of the time while a job was carrying on an input/output (I/O) operation. On the other hand, input/output devices were often idle while processing was going on. Even overlapping processing with I/O left these devices unused a significant time and so the high cost of having these devices stand idle (especially the central processor) led designers to systems which allow

more than a single job in main memory. This allows the CPU to switch to another task when the task currently in execution is held up by an I/O request. This may considerably increase CPU utilization and that of other devices as well. On the other hand, the amount of main memory required for successful multiprogramming is considerably greater than that required when only a single program is in core. Since core memory is also an expensive resource, it is desirable to use this as efficiently as possible as well.

The key to efficient use of the main memory is the proper scheduling of tasks wishing to use this resource. In fact, getting the greatest possible system throughput or CPU utilization at peak load times requires careful choice of scheduling rules for both the main memory and central processors, and in order to attack the problem of making these choices intelligently, it is necessary to understand the structure and characteristics of multiprogrammed computer systems. The remainder of this section is devoted to a description of the structure of typical multiprogrammed computers as background for the computer system model which is developed in Chapter 3 and which is used to evaluate task scheduling rules in such computers.

In many multiprogrammed computer systems many tasks require either identical procedure (code) or possibly make use of the same data as other tasks. In a general purpose computing system, this is most likely to occur because many users require the same routines such

as compilers and assemblers. In a more specialized environment, data bases may be required by multiple users. If two or more such tasks are multiprogrammed together then it may be possible for the two tasks to share a single core copy of the information common to both. This may result in significant savings of memory. It should be noted, however, that the additional hardware and/or software needed to implement this capability may be non-trivial in some systems.

Reentrant or pure procedure is procedure which does not modify itself in the course of execution. This makes it possible for more than one user to execute the same copy of this code without fear that it has been modified by another user. There are other advantages to the use of such procedure besides the savings in memory requirement. The traffic on the channel used to bring this procedure into memory is reduced because only one copy is brought in for multiple users. The traffic may be further reduced because this information need never be removed from main memory if the task is swapped or paged out. Since the procedure is never changed, a copy can be maintained external to main memory and the copy in main memory may simply be overwritten. Another advantage is the reduction in time required to load a task into the core when the procedure required by that task has already been loaded.

The same comments apply for the sharing of data among tasks except that additional complexities may arise if some or all tasks are

allowed to make changes in these data. To avoid errors it may be necessary for a task making changes to the data to prevent other tasks from accessing the data until changes are complete. This level of detail is not considered further in this report.

It was mentioned above that the cost of sharing information among users may be the introduction of additional operating system overhead. Many multiprogrammed computers in use today, however, have features which minimize this additional cost; and in fact, were designed to make such sharing easy to implement. The two-level address translation hardware in use on many machines today facilitates the sharing of pages by users. This addressing structure and the hardware necessary to implement it are described in papers by Arden, et. al.[2] and by Dennis [14]. The IBM/360 Model 67 has such hardware.

Machines like this also include the ability to maintain in core only those parts of a program which are currently in use, and to use an address space much larger than the physical memory available. This separation of the physical address from the logical address gives rise to the term virtual memory. Programs are broken into fixed-size units called pages and these units are brought into core when needed and removed from core when they are no longer being used. Commonly, pages of a task that are not in core are brought into core when they are referenced by the task. At this point the program is ineligible to execute until the page has been moved into main memory from some secondary storage medium. This is, in essence, simply another input/output operation

although in this case the number of such input/output operations will depend on the memory management policies used by the operating system rather than solely on the program being run. Determining what pages to remove from core to make room for newly requested pages is more complex, but may involve attempting to identify and remove those pages which have been least recently referenced.

Paging allows a greater number of tasks to share core simultaneously than would be possible in a multiprogrammed system operating in a real core environment. In this sort of system the entire task must be in main memory before execution can begin although of course programmer controlled overlays are possible. The use of paging and virtual memory become especially attractive when the multiprogrammed system is also time-shared. In a time-shared system, many of the terminal users are likely to be running highly interactive tasks during most of the time they spend at the terminal. With tasks like these, the mean real time between references to a page is long (seconds) even though the mean CPU time between such references may be short (milliseconds). This occurs because of the long time required by users at terminals to make responses relative to response times of other input/output devices, and also because an interactive task is quite likely to reference only a relatively small fraction of its total pages between two terminal interactions. This is especially likely, for example, during on-line debugging of a program, where the user may interrupt the program frequently to display and alter variables and where program in-

terrupts may occur. In situations like this, the mean number of pages in memory may be substantially less than the total number of pages the task has, and therefore a higher degree of multiprogramming (i.e. more programs may simultaneously share core) is possible than in a system where the entire task must be in core. Looked at in another way, only the pages requested for a single interaction need be swapped in and out of core, rather than the entire task.

The mathematical model developed in this report applies to both virtual and real memory machines which may be multiprogrammed. Use of a real core machine is more likely in a batched system, and of course, this is a very important application of computer systems also. Other important facets of modern computing machines include the possibility of multiple central processing units and a diversity of devices for input and output; neither of these causes any difficulty in the mathematical model.

The strategies used by the operating system are also important to the characterization of the computing system. In particular, the task scheduling strategies may be quite important to the system's operation. Scheduling of one form or another must occur whenever there are resources to be allocated; the three major classes of resources are the main memory, the central processors, and input/output channels and devices. We are explicitly concerned with scheduling the first two of

these in this report. The scheduling of input/output devices may usually be treated satisfactorily as a problem local to that device: the diversity and complexity of many of these devices make it extremely difficult to treat them in detail in the context of the system as a whole.

Good procedures for scheduling tasks become less obvious as we move from monoprogrammed, batch systems up to multiprogrammed, time-shared systems with virtual memory. The rule may simply be first-come-first-serve (FCFS) in a monoprogrammed, batch system or, if job lengths can be determined a priori, a shortest job first policy may be followed since this minimizes mean turn-around time. In this system, memory scheduling and CPU scheduling are synonymous. If this system is time-shared, we may wish to give each queued job a short period of execution, called a time slice, then swap it out to make room for another if it has not terminated. This is called round-robin service and prevents a long job from causing a long delay to a short job although the mean response time to users remains about the same as for FCFS service [22]. The choice of time slice is important because there is an overhead time cost every time tasks are swapped while unhappy users may result if the time slice is too long.

When we move to a multiprogrammed system, the problem increases in complexity because now the CPU and memory must be scheduled separately, although there is some interdependence since tasks not in memory cannot use the central processor. FCFS queues may

be used, but it is not at all clear that this policy makes best use of system resources. In fact, a last-come-first-serve (LCFS) policy is used by the computing center at the University of Michigan because a job usually arrives at this queue after completion of an input or output operation and is therefore assumed likely to be an I/O bound job (i.e., a job which uses large amounts of I/O time relative to CPU time used) which will relinquish the CPU after a very short CPU interval. This prevents compute bound jobs from keeping I/O bound jobs queued at the CPU and therefore unable to carry out their I/O operations. If the multiprogrammed system is time-shared as well, we may want to consider applying a time slice to tasks using the CPU and perhaps to the use of main memory as well.

1.2 Modelling Computers

Many mathematical models, both for computer systems as a whole and for selected aspects of computer systems (e.g. drums) have been developed and can provide valuable insight into the operating characteristics of such systems and subsystems. These systems may generally be characterized as stochastic service systems and analytic or simulation models may be used to study relations between parameters of the systems. Analytic models, which usually fall into the class of Markovian queueing models, may be further subdivided into those which yield a closed form solution to quantities of interest in terms of input parameters, and those which do not yield to closed form analysis

in this way but may be solved using numerical methods. The model developed in this report falls into the latter class. The remainder of this section is devoted to a review of models which have some applicability to the problem attacked in this thesis; namely, the effects of shared information and scheduling rules on throughput of a multiprogrammed computer system.

Few analytical results are available which are applicable to the problem of sharing in computer systems. Furthermore these results apply only to a monoprogrammed system where the saving from using shared procedure accrues by not having to re-load procedure used by two successive arrivals to the system.

A rather large volume of literature [9, 25, 27, 35, 44] has been published which gives analytical results from models of time-shared computer systems. Unfortunately, the results are generally for round-robin or some similar form of service in a monoprogrammed system, and so the results are not useful for investigation of the additional complexities which arise with multiprogramming.

Gaver [20] develops a cyclic queueing model of a multiprogrammed computer system. He assumes a fixed number of indistinguishable tasks in memory which alternate between periods of CPU use and I/O operations except where they are queued at these devices. Duration of I/O operations is assumed exponentially distributed, but several distributions are given for the duration of a CPU operation. Results are in terms of CPU

utilization which, after multiplication by a constant, may be equated directly to system throughput. Tables of results show CPU utilization as a function of number of I/O devices, number of tasks in memory, I/O device speed, and distribution on CPU interval length. Scheduling is not an issue in this model because tasks are indistinguishable from one another.

If we wish to investigate scheduling rules in a system in which tasks are differentiated, we must turn to a multi-queue model. Results are extremely limited and difficult to get. Eisenberg [15] and Leibowitz [31,32] both obtain results to multi-queue problems. Eisenberg's results are most useful for application to scheduling tasks although the results are again limited to a monoprogrammed computer system. Eisenberg treats the case of two queues with general distributions on the service time and Poisson arrivals to each queue. Only one job may receive service at a time. If at the completion of a task in one queue it is decided to service another task in the same queue, service begins immediately. If, on the other hand, it is decided to service a task in the other queue, a changeover time is required before service may begin in that queue. Applying this model to the problem of sharing in a monoprogrammed computer system, the two queues represent the two types of tasks which are arriving at the system. The time to move from service of queue 1 to queue 2 is just the time required to

load the sharable information for task type 2 and vice-versa. The service time for a task is just the time to load the non-sharable information for that task and execute it. Eisenberg obtains expressions for the mean waiting times of tasks in each of the queues under two service disciplines: alternating priority and strict priority. Under alternating priority, tasks of one type are serviced until there are no more tasks of that type in the system. A changeover to the other queue then occurs and service of tasks in that queue proceeds until it is empty, etc. Under strict priority, queue 1 is always given priority over queue 2. The expressions for the mean waiting times are very complex, even for this case of only two queues (two types of tasks). These results and the difficulties encountered in obtaining them indicate the futility of trying to obtain closed form results from a queueing model representative of a multiprogrammed computer with several types of tasks which may share information.

One method of modelling which may be used to investigate scheduling policies when tasks are differentiated is simulation. Among others, Pinkerton[41] and Nielsen[36] have both developed simulation models of multiprogrammed computer systems. Lasser[30], however, points out that a simulation model which attempts to account for both "macro-queuing" (e.g. queuing for memory) and "micro-queuing" (e.g. queuing for the CPU) may be very expensive to run because of the large number of "micro" events one must consider in order to obtain a sufficient

sample of "macro" events. Nielsen [36] reports using 5-10 minutes of Burroughs B5500 processor time to simulate each minute of operation of a Burroughs B6500 system. Furthermore, simulation models do not lend themselves easily to determination of optimal system configurations and operating policies. Simulation can, however, be a very useful tool, and can provide a level of detail in modelling not attainable by any other means.

The approach to the problem taken in this report is to develop a large Markovian queueing model which yields the quantities of interest by application of numerical methods. Wallace and Rosenberg [56] discuss this approach to modelling which represents a middle ground between closed form solution of queuing models and simulation. Furthermore, Markov models of this sort lend themselves to application of optimization techniques if appropriate objectives can be found. In this work the objective is maximization of system throughput which is linearly related to CPU utilization, and the problem is then formulated as a Markov renewal decision process (MRDP).

1.3 Queues, Markov Processes, and Optimization

It is the purpose of this section to provide definitions and background for material which appears in later chapters. Some mathematical rigor may be sacrificed to simplify the exposition.

A network of queues and service facilities may be used to characterize a computer system. Queues may occur at central processors,

I/O devices, and at the main memory of the computer, which are in turn the service facilities; and these service facilities may have arbitrary probability distributions on their service times. These arbitrary distributions may often be reasonably approximated by a network of servers with exponentially distributed service times [34]; this is done to make the model Markovian and therefore tractable to analysis.

As a simple example, consider the cyclic queuing system shown in Figure 1.1. The circles indicate queues and each square an exponential server. The mean service times are $1/\mu$ and $1/\sigma$ in service facilities 1 and 2 respectively. Whenever a service facility becomes free, a new customer immediately enters it providing, of course, that there is a customer enqueued. A state for this simple network can be specified by a single number giving the number of customers queued at or in service at service facility 1. Since the network is cyclic, there is a constant number of customers in the system (say N); and so with m customers at facility 1, there are N - m at facility 2. We have N+1 states, 0, 1, ..., N; and may be interested in determining, for example, the probability of being in state j at time t given that state i is occupied at time 0.

If service distributions are not exponential in the physical system being modelled, the exponential servers may be replaced by networks of exponential servers in order to approximate the desired distribution. This is discussed in Morse [34] and a simple example is shown in

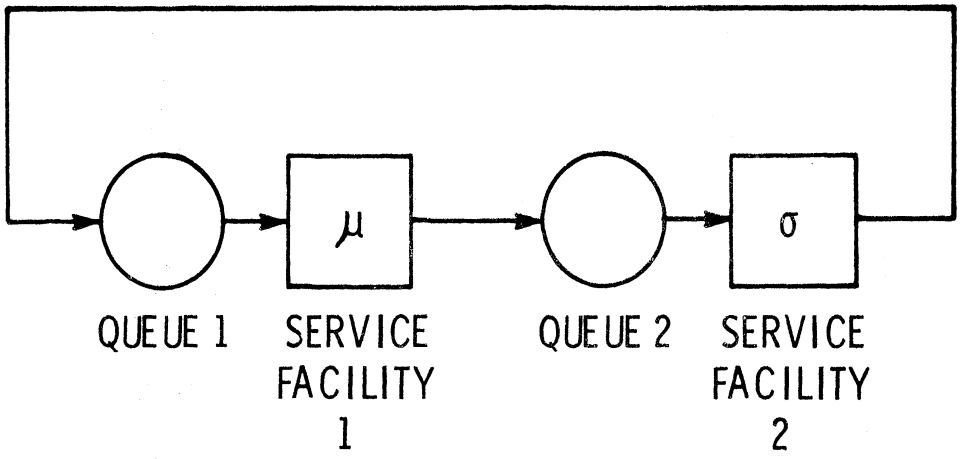

Figure 1.1. Cyclic Network of Queues and Servers

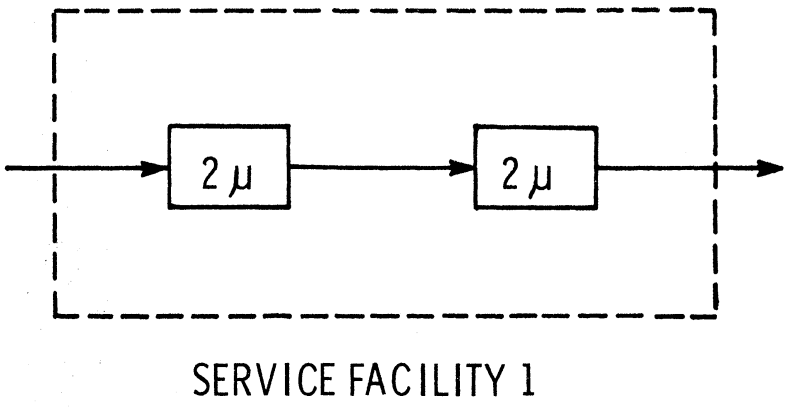

Figure 1.2. Replacement of Exponential Service With Erlang - 2 Service

Figure 1.2 where the exponential service facility 1 has been replaced by Erlang-2 service. The dashed lines are intended to indicate that only one of the servers may be occupied at any one time.

The exponential server with mean service time $1/\mu$ has the useful property that the remaining service time is exponentially distributed with mean $1/\mu$ independent of the quantity of service already received. This property makes probabilities of being in particular states at any time in the future dependent only on the current state, and not on any past history of the queuing network. This is the Markov property and is very useful and widely used in the modelling of stochastic service systems.

The definition of a Markov process is formalized as follows [38]. In this work we consider only discrete state Markov processes (often called Markov chains) with a finite number of states. We may use the set of integers $\{0, 1, \ldots, N\}$ to denumerate the states of the Markov process and denote the state of the process at time t by X(t). If the state, X(t), is observed at any sequence of time instants $0 \leq t_1 < t_2 < \ldots < t_n$, and the relation

$$\Pr[X(t_n) = x_n \mid X(t_1) = x_1, \ldots, X(t_{n-1}) = x_{n-1}]$$

$$= \Pr[X(t_n) = x_n \mid X(t_{n-1}) = x_{n-1}] \qquad (1.1)$$

holds, then the process is a continuous parameter Markov chain. When the t_i are restricted to integer values, the process is called a discrete parameter Markov chain. If, in addition, the conditional probability in Equation (1.1) is dependent only on the difference, $t_n - t_{n-1}$, the Markov process is homogeneous and has stationary transition probabilities.

Denote by $p_n(t)$, the probability that at time t, state n is occupied in the example of Figure 1.1. The probability of going from state n to state n-1 (n>0) in a very short interval Δt is approximately

$$1 - e^{-\mu\Delta t} \approx \mu\Delta t$$

Using this approximation, the following equation may be written.

$$p_n(t + \Delta t) = p_{n-1}(t)\,\mu\Delta t + p_n(t)\left[1 - (\mu+\sigma)\,\Delta t\right] + p_{n+1}(t)\,\sigma\Delta t\,; \qquad 0 < n < N$$

Taking $\lim_{\Delta t \to 0}$ we get

$$\frac{dp_n(t)}{dt} = \mu p_{n-1}(t) - (\mu+\sigma)\,p_n(t) + \sigma\,p_{n+1}(t)\,; \qquad 0 < n < N.$$

Similarly, equations for n = 0 and n =N lead to

$$\frac{dp_o}{dt} = -\sigma\,p_o(t) + \sigma\,p_1(t)$$

$$\frac{dp_N(t)}{dt} = \mu\,p_{N-1}(t) - \mu\,p_N(t)$$

Letting $p(t) = \left(p_0(t), p_1(t), \ldots, p_N(t)\right)$, an N+1 dimensional row vector, and calling A the matrix of coefficients in the differential equations above, we have

$$\frac{dp(t)}{dt} = p(t)\,A$$

The coefficients are called transition intensities and A a transition intensity matrix. The procedure outlined above may be applied to more complex networks and the equations solved to find, for example, the

probability that state j is occupied at time t given that state i is occupied at time 0, i.e. given that $p_i(0)=1$. Solution to such systems of differential equations will be required in the computer system model developed in Chapter 3.

We make use of Markov renewal processes as well as the continuous parameter Markov processes defined above where again only homogeneous processes with a finite number of states are of interest. Markov renewal processes are generalizations of the Markov chains discussed above. In Markov renewal processes the successive states form a Markov chain while the time between successive transitions is a random variable determined by a stationary probability distribution function which may depend on both the current state and the state to which transition is made. In the Markov chain, the probability of making a transition from state i to state j may be denoted by p_{ij}. The matrix of such transition probabilities, (p_{ij}), is sufficient to generate successive states once the initial state is known, and these transition probabilities are independent of the length of the interval between successive transitions. The distribution function used to determine the probability that the time required to make a transition directly from state i to state j is equal or less than τ is denoted by $F_{ij}(\tau)$. Thus, given that i→j transition is about to occur, the duration of this transition has probability distribution $F_{ij}(\tau)$ while the probability of making an i→j transition is p_{ij} independent of the duration of the transition. These quantities are sufficient to define a Markov renewal

process which is a useful extension to the theory of Markov processes. Markov renewal processes are treated in detail in [42] .

In Chapter 3 a Markov renewal process with a finite number of states will form the computer system model. There is a one-one correspondence between a state in the model and a physical configuration of the computer system only at the time points where state transitions occur (called regeneration points). A careful definition of system states is required to insure that the probability of making a transition to a particular state is dependent only on the current state and not on any other information about the past behavior of the system. After the states are defined it is necessary to find the transition probabilities among the states (p_{ij}) and the distributions on the intervals between transitions ($F_{ij}(\tau)$). Determination of the transition probabilities necessitates the formation of a number of sub-models, each of which is a continuous parameter Markov chain.

A return function, $R_{ij}(\tau)$, may also be defined for each transition i→j. It is then possible to determine both an average return per unit time or an average return per transition. If, in addition to this, choice of any one of several alternatives is available in each state, an optimization problem may be formulated. Suppose that in state i, alternatives $1, 2, \ldots,$ K (i) are possible. p_{ij}^{k} , $F_{ij}^{k}(\tau)$, and $R_{ij}^{k}(\tau)$ are the transition probability, distribution on transition time, and return using alternative k in state i.

Optimization then involves choosing alternatives in each state to maximize expected return per unit time or per transition. These concepts are defined in more detail in Chapter 2.

1.4 Computer Performance Data

There is a paucity of data available which measure the performance of the various aspects of computer systems. One major reason for this is the difficulty and expense of collection and analysis of such data. Previously published data on one aspect or another of system performance include Pinkerton [40, 41], and Walter and Wallace [57] at the University of Michigan; Varian and Coffman [10]; and Fine, Jackson, and McIsaac [18].

Data may be useful in providing parameter values to a model of a system and for validating assumptions made in a model. Of course, this is true only in a limited sense, since one is not usually interested in modelling an existing system (the ultimate model has already been built) but in modelling a system similar to one in existence.

Data available in the literature were not satisfactory for the purposes of this study and so data were collected and analyzed from the Michigan Terminal System (MTS). This is a large time-shared system implemented on an IBM System/360 Model 67 at the University of Michigan.

Chapter 2
MARKOV RENEWAL DECISION PROCESSES

In this chapter we present a useful technique for the solution of a class of Markov-renewal decision processes (MRDP's) first formulated by Jewell [24]. These processes are a generalization of the Markov decision processes described by Bellman[6] and Howard [23] . Briefly, MRDP's are Markov-renewal processes with a finite number of states in which the parameters of the process (p_{ij} and $F_{ij}(t)$) are dependent on a controller which makes one of a finite number of choices at each transition. With each decision made by the controller are associated transition dependent rewards and the solution to the optimization problem consists of finding the set of decisions which maximizes the expected return as determined by the rewards.

When occupying state i($i = 1, \ldots, N$), we choose one of a finite number of alternatives k(i) ($k(i) = 1, 2, \ldots, K(i)$). For $k(i) = k$ we have associated a transition probability p_{ij}^k, and distribution on transition interval, $F_{ij}^k(t)$. The reward under alternative k for making a transition from state i to state j is $R_{ij}^k(t|\tau)$ where $t \leq \tau$ is the elapsed time since the beginning of the transition interval which is of length τ. Define

$$R_{ij}^k(\tau) = R_{ij}^k(\tau|\tau)$$

Then

$$q_i^k = \sum_{j=1}^{N} p_{ij}^k \int_o^\infty R_{ij}^k(\tau)\, dF_{ij}^k(\tau) \qquad (2.1)$$

is the expected one transition return starting in state i and using alternative k.

Note that if the transition interval is fixed and of length T then we have the special case

$$q_i^k = \sum_{j=1}^{N} p_{ij}^k r_{ij}^k \qquad (2.2)$$

where $r_{ij}^k = R_{ij}^k(T)$. This is the case considered by Howard [23] and for this situation the more general rewards reduce to the above.

We may define two distinct objectives for the controller.

1) Maximize the expected return per transition (called the transition-optimal problem here.)

2) Maximize the expected return per unit time (called the time-optimal problem here.)

Bellman's "Principle of Optimality" [5] may be used to derive recurrence relations for these processes. These recurrence relations will specify the alternatives to be chosen at each transition in order to maximize expected returns. The recurrence relations for the two problems are given below.

1) The transition optimal problem. The recurrence relation is

$$v_i(n) = \max_k \left[q_i^k + \sum_{j=1}^{N} p_{ij}^k v_j(n-1) \right] \qquad (2.3)$$

where

$v_i(n)$ is the expected return given that an optimal policy is followed, we start in state i, and the process makes n transitions before terminating.

$v_i(0)$ must be specified and is the boundary reward received for being in state i at the termination of the process.

q_i^k is the expected return for transition n as determined from Equation (2.1).

2) The time-optimal problem. The recurrence relation is

$$w_i(t) = \max_k \sum_{j=1}^{N} p_{ij}^k \left\{ \int_t^\infty d\, F_{ij}^k(\tau)\, [R_{ij}^k(t|\tau) + S_{ij}^k(t,\tau)] \right.$$

$$\left. + \int_0^t d\, F_{ij}^k(\tau)\, [R_{ij}^k(\tau) + w_j(t-\tau)] \right\} \qquad (2.4)$$

$w_i(t)$ is the expected return given that an optimal policy is followed, we start in state i, and terminate the process after time interval t.

$w_i(0)$ must be specified and is the boundary reward received for being in state i at termination of the process.

$S_{ij}^k(t,\tau)$ is a boundary reward received when the process terminates. It is a function of the two states involved in the transition which is interrupted at the time of termination and of the total time and elapsed time of this interrupted transition.

The first term in the recurrence relation represents the expected

return if no transitions are made before the process terminates. The second term represents the return from one transition of time $\tau < t$ plus the expected return in the remaining time, $t - \tau$.

If there are no restrictions on the probability distribution functions, $F_{ij}^{k}(t)$, it is clear that, in general, Equation (2.4) cannot be solved. Appropriate discrete approximations on the continuous time variable may, however, be made in order to obtain a solution. This is done by replacing the variable t by $t = n\Delta$; $n = 1, 2, 3, \ldots$ and for some $\Delta > 0$; and by using the approximation

$$\int_{0}^{n\Delta} dF_{ij}^{k}(\tau)\, w_j(n\Delta - \tau) \cong \sum_{\ell=1}^{n} \left[F_{ij}^{k}(\ell\Delta) - F_{ij}^{k}(\ell\Delta - \Delta) \right] w_j(n\Delta - \ell\Delta). \tag{2.5}$$

With this restriction and approximation Equation (2.4) becomes

$$w_i(n\Delta) \cong \max_{k} \sum_{j=1}^{N} p_{ij}^{k} \left\{ \int_{n\Delta}^{\infty} dF_{ij}^{k}(\tau) \left[R_{ij}^{k}(n\Delta \mid \tau) + S_{ij}^{k}(n\Delta, \tau) \right] \right.$$

$$+ \int_{0}^{n\Delta} dF_{ij}^{k}(\tau)\, R_{ij}^{k}(\tau)$$

$$\left. + \sum_{\ell=1}^{n} \left[F_{ij}^{k}(\ell\Delta) - F_{ij}^{k}(\ell\Delta - \Delta) \right] w_j(n\Delta - \ell\Delta) \right\} \tag{2.6}$$

which can be built up in the usual recursive manner. Of course, the accuracy of this approximation depends strongly on the choice of Δ. If all $F_{ij}^{k}(t)$ are restricted to be lattice distributions with distribution span Δ, then the approximation becomes exact.

2.1 Infinite Time Processes

The recurrence relations given for the transition optimal and time optimal problems formulated in the preceding section provide a practical method of solution for these problems providing the number of iterations does not grow too large. In many practical problems, however, this is not the case and in fact we want to know the behavior of the recurrence equations as $n \to \infty$ and $t \to \infty$ in Equations (2.3) and (2.4), respectively. The remainder of this chapter is devoted to solution of these problems.

Define a stationary policy, α, as an N-tuple which specifies a particular alternative in each of the N states. A recurrence equation for the expected return in n transitions when following stationary policy α may be written

$$v_i^{\alpha}(n) = q_i^{\alpha} + \sum_{j=1}^{N} p_{ij}^{\alpha} v_j^{\alpha}(n-1) \tag{2.7}$$

Again, of course, a boundary reward $v_i^{\alpha}(0)$ must be specified for all i. Similarly, the expected return in time t when following stationary policy α is

$$w_i^{\alpha}(t) = \sum_{j=1}^{N} p_{ij}^{\alpha} \left\{ \int_t^{\infty} dF_{ij}^{\alpha}(\tau) \left[R_{ij}^{\alpha}(t \mid \tau) + S_{ij}^{\alpha}(t, \tau) \right] \right.$$

$$\left. + \int_0^t dF_{ij}^{\alpha}(\tau) \left[R_{ij}^{\alpha}(\tau) + w_j^{\alpha}(t - \tau) \right] \right\} \tag{2.8}$$

It has been shown [24] that for reasonable restrictions (which will be given when needed in what follows) on the various parameters in (2.7) and (2.8) the following relations hold when a stationary policy is in use.

$$v_i^{\alpha}(n) \rightarrow g^{\alpha} n + v_i^{\alpha} \qquad \text{as} \quad n \rightarrow \infty \tag{2.9}$$

$$w_i^{\alpha}(t) \rightarrow \psi^{\alpha} t + w_i^{\alpha} \qquad \text{as} \quad t \rightarrow \infty \tag{2.10}$$

for all $1 \leq i \leq N$ and for appropriate constants v_i^{α} and w_i^{α}. g^{α} is called the gain or gain per transition, and ψ^{α} the gain rate or gain per unit time under policy α. For large n it is seen that g^{α} is the expected return per transition, and for large t, ψ^{α} is the expected return per unit time. Except in the unlikely event that g^{α} or ψ^{α} equal zero, the total expected return, $v_i^{\alpha}(n)$ and $w_i^{\alpha}(t)$, both go to infinity as n and t go to infinity in Equations (2.9) and (2.10). Because it has been shown by Fife [16] that there exists an optimal stationary policy for each of the infinite stage and infinite time optimization problems, it is clear that the objective in these problems is to find that policy, α, which maximizes g^{α} or ψ^{α} in Equations (2.9) or (2.10), respectively.

The following theorem proves the result given in Equation (2.9) along with some related results which will be useful in the following sections. The result given in Equation (2.10) is not needed in what follows but has been proved by Jewell [24] .

Theorem 2.1

Given a recurrence relation

$$v_i(n) = q_i + \sum_{j=1}^{N} p_{ij}\, v_j(n-1) \qquad (2.11)$$

$$v_i(0) = c_i \quad ; \quad i=1,2,\ldots,N \qquad (2.12)$$

in which the NxN stochastic matrix $P=(p_{ij})$ is acyclic and has a single closed communicating class of states so that there exists a vector of limiting state probabilities

$$\pi = (\pi_1, \pi_2, \ldots, \pi_N).$$

Such a recurrence relation could occur in a MRDP operating under a stationary policy, but no explicit indicator of policy is given in (2.11).

Define

$$g_i(n) = v_i(n) - v_i(n-1);$$

$$i=1,2,\ldots,N;\ n=1,2,\ldots . \qquad (2.13)$$

Then there exists a constant g such that

$$\sum_{i=1}^{N} \pi_i g_i(n) = g \quad \text{for all } n, \qquad (2.14)$$

and furthermore the following equations must be true.

$$\lim_{n\to\infty} [\, v_i(n) - gn - v_i] = 0 \qquad (2.15)$$

for all i and for some constant v_i.

$$\sum_{i=1}^{N} \pi_i q_i = g \qquad (2.16)$$

Proof

Define a vector

$$G(n) = \begin{pmatrix} g_1(n) \\ \vdots \\ g_N(n) \end{pmatrix} \tag{2.17}$$

and note that from the recurrence relation, the following equation must hold for all n.

$$G(n) = PG(n-1) \tag{2.18}$$

Pre-multiplying by the vector, π, get

$$\pi G(n) = \pi PG(n-1)$$

By definition of the limiting state probabilities, the equation

$$\pi = \pi P$$

must hold so that

$$\pi G(n) = \pi G(n-1) \tag{2.19}$$

Since this is true for any n, it is clear that $\pi G(n)$ is a constant independent of n which proves Equation (2.14).

Again, from the recurrence relation, it must be true that

$$G(n) = P^{n-1} G(1)$$

and since

$$\lim_{n \to \infty} P^n = \begin{pmatrix} \pi \\ \vdots \\ \pi \end{pmatrix}$$

it is clear that

$$\lim_{n \to \infty} g_i(n) = g \quad \text{for all i.}$$

From the definition of $g_i(n)$, it follows that

$$\lim_{n \to \infty} [\, v_i(n) - v_i(n-1)] = g \quad \text{for all i} \tag{2.20}$$

and this can be true only if there is some constant v_i such that

$$\lim_{n \to \infty} [\, v_i(n) - gn - v_i] = 0 \tag{2.15}$$

In order to derive Equation (2.16) it is convenient to define V(n) as a column vector of the $v_i(n)$, and Q as a column vector of the q_i:

$$V(n) = \begin{pmatrix} v_i(n) \\ \vdots \\ v_N(n) \end{pmatrix} \quad ; \qquad Q = \begin{pmatrix} q_1 \\ \vdots \\ q_N \end{pmatrix}$$

Then the following equations may be derived iteratively.

$$\begin{aligned} V(n) &= Q + PV(n-1) \\ V(n) &= Q + PQ + P^2V(n-2) \\ &\vdots \\ V(n) &= Q+PQ+P^2Q+\cdots+P^{n-1}Q+P^nV(0) \end{aligned}$$

Similarly, an equation for V(n-1) is written:

$$V(n-1) = Q+PQ+P^2Q+\cdots+P^{n-2}Q+P^{n-1}V(0)$$

These last two equations may be subtracted and since $G(n) = V(n)-V(n-1)$, we get

$$G(n) = P^{n-1}Q + (P^n-P^{n-1})V(0) \qquad (2.21)$$

Therefore,

$$\lim_{n\to\infty} G(n) = \begin{pmatrix} \pi \\ \vdots \\ \pi \end{pmatrix} Q$$

and each element of this vector equality gives the desired result,

$$g = \sum_{i=1}^{N} \pi_i q_i$$

Howard [23] obtains this last result in a different way.

In the next two sections we make the recurrence relations given in Equations (2.3) and (2.4) useful for determining the policy to be used when the process is to run an infinite time in the transition-optimal or time-optimal problem. This is done by obtaining at each stage in the iteration an upper bound on the gain of the optimal policy and a lower bound on the gain of the current policy. It is shown that these bounds converge to the gain or gain rate of the optimal policy and that the difference between these bounds at each successive stage forms a monotone non-increasing sequence.

2.2 Transition-Optimal Problem

In this section we look at the problem of maximizing the gain per transition. We obtain bounds on the gain of both the current and the optimal policies and show that these bounds converge to one another as the number of transitions approaches infinity. In the recurrence relation

$$v_i(n) = \max_k \left[q_i^k + \sum_{j=1}^{N} p_{ij}^k v_j(n-1)\right], \tag{2.3}$$

the policy chosen at stage n may be denoted by the N-tuple

$$\alpha = (k_1(n), \ldots, k_N(n))$$

and consists of the alternatives chosen in each of the N states at the n-th stage. Note that in (2.3) policy α is chosen to maximize $v_i(n)$ for all $i=1,2,\ldots,N$ and that in general greek superscripts specify policies while lower case roman superscripts specify alternatives. Define

$$\begin{aligned} g_i(n) &= v_i(n) - v_i(n-1) \\ &= \max_k \left[q_i^k + \sum_{j\neq i} p_{ij}^k v_j(n-1) + (p_{ii}^k - 1) v_i(n-1)\right] \end{aligned} \tag{2.22}$$

Let π_i^β be the limiting state probability for state i under arbitrary policy β where, of course,

$$\sum_{i=1}^{N} \pi_i^\beta = 1 .$$

Also define

$$g_i^\beta(n) = q_i^\beta + \sum_{j\neq i} p_{ij}^\beta v_j(n-1) + (p_{ii}^\beta - 1)\, v_i(n-1) \tag{2.23}$$

which implies that

$$g_i(n) = \max_{\beta} g_i^{\beta}(n). \tag{2.24}$$

Theorem 2.1 shows that the gain under stationary policy β, g^{β}, is

$$g^{\beta} = \sum_i \pi_i^{\beta} q_i^{\beta}.$$

Lemma 2.2

$$\sum_{i=1}^{N} \pi_i^{\beta} g_i^{\beta}(n) = g^{\beta}. \tag{2.25}$$

Proof

From Equation(2.23) it follows that

$$\sum_i \pi_i^{\beta} g_i^{\beta}(n) = \sum_i [\pi_i^{\beta} q_i^{\beta} + \sum_{j \neq i} \pi_i^{\beta} p_{ij}^{\beta} v_j(n-1) + \pi_i^{\beta}(p_{ii}^{\beta} - 1)v_i(n-1)] \tag{2.26}$$

Now

$$\sum_i \pi_i^{\beta} q_i^{\beta} = g^{\beta}$$

Also, a finite Markov chain must satisfy the matrix equation $\pi = \pi P$ where $\pi = (\pi_1, \ldots, \pi_N)$ is the vector of stationary state probabilities and $P = (p_{ij})$ is the transition probability matrix. Using this equation it follows directly that

$$\pi_i^{\beta}(p_{ii}^{\beta} - 1) = -\sum_{j \neq i} \pi_j^{\beta} p_{ji}^{\beta} \tag{2.27}$$

Thus

$$\sum_i \pi_i^\beta g_i^\beta(n) = g^\beta + \sum_i \left[\sum_{j \neq i} \pi_i^\beta p_{ij}^\beta v_j(n-1) - \sum_{j \neq i} \pi_j^\beta p_{ji}^\beta v_i(n-1) \right] = g^\beta \tag{2.28}$$

Theorem 2.3

$$\max_i \left[g_i(n) \right] \geq g^* \geq g^\alpha \tag{2.29}$$

$$\min_i \left[g_i(n) \right] \leq g^\alpha \leq g^* \tag{2.30}$$

where α is the policy chosen at stage n from the recurrence relation (Equation 2.3) and * is a stationary optimal policy.

Proof

First we note that $g_i^\alpha(n) = g_i(n)$ since α is the policy which maximizes $g_i(n)$ at the current stage.

From Lemma 2.2 and the fact that

$$\sum_{i=1}^{N} \pi_i^\alpha = 1 \text{ and } \pi_i^\alpha \geq 0, \text{ all i,}$$

we have that $g_i(n) \leq g^\alpha$ for some i and $g_j(n) \geq g^\alpha$ for some j. This proves that part of the theorem which relates to the current policy, α. Suppose α is an optimal policy. Then proof of the theorem is complete at this point.

Next suppose α is not an optimal policy. Then $g^\alpha < g^*$. It must be shown that $g_i(n) \not> g^*$ $\forall i$ and that $g_i(n) \not< g^*$ $\forall i$. First suppose $g_i(n) > g^*$ $\forall i$. But then $g^\alpha = \sum \pi_i^\alpha g_i(n) > g^*$ which is a contradiction.

Next suppose $g_i(n) < g^*$. $\forall i$ recalls

$$g_i(n) = \max_k [q_i^k + \sum_j p_{ij}^k \; v_j(n-1) + (p_{ii}^k - 1) \; v_i(n-1)]$$

so that $g_i(n) \geq g_i^*(n) \quad \forall i$. Therefore

$$\sum \pi_i^* \; g_i(n) \geq \sum \pi_i^* \; g_i^*(n) = g^*$$

and this contradicts the assumption that $g_i(n) < g^* \quad \forall i$.

We have now shown that if one $g_i(n)$ is equal or less than g^* then at least one $g_i(n)$ must be equal or greater than g^* and conversely. This completes a proof of the theorem.

We now would like to show that $\max_i [\, g_i(n)] - \min_i [\, g_i(n)] \rightarrow 0$ as $n \rightarrow \infty$. The following theorem proves this.

Theorem 2.4

If, in a MRDP, there exists a number M, such that under any sequence of policies, there exists a state m reachable with probability $\geq \sigma > 0$ from any arbitrary state i in exactly M transitions; then

$$\max_i [g_i(n)] - \min_i [g_i(n)] \rightarrow 0 \quad \text{as} \quad n \rightarrow \infty. \tag{2.31}$$

Proof

The proof here follows a paper by White [58]. By hypothesis it is possible to reach state m in M transitions with probability $\geq \sigma > 0$, independent of the policy sequence chosen. From the

equation

$$v_i(n) = \max_k [q_i^k + \sum_j p_{ij}^k v_j(n-1)] \tag{2.3}$$

it follows that

$$g_i(n) = v_i(n) - v_i(n-1) \leq \max_k [\sum_j p_{ij}^k g_j(n-1)] \tag{2.32}$$

$$g_i(n) = v_i(n) - v_i(n-1) \geq \min_k [\sum_j p_{ij}^k g_j(n-1)] \tag{2.33}$$

We may iterate the expression (2.32) to get

$$g_i(n) \leq \max_{k_1} [\sum_j p_{ij_1}^{k_1} \ldots \max_{k_M} [\sum_{j_M} p_{j_{M-1}j_M}^{k_M} g_{j_M}(n-M)] \ldots] \tag{2.34}$$

Similarly (2.33) may be iterated to yield

$$g_i(n) \geq \min_{k_1} [\sum_{j_1} p_{ij_1}^{k_1} \ldots \min_{k_M} [\sum_{j_M} p_{j_{M-1}j_M}^{k_M} g_{j_M}(n-M)] \ldots] \tag{2.35}$$

The Inequalities (2.34) and (2.35) each determine a sequence of alternatives to be used on successive transitions in every state i; $1 \leq i \leq N$. This sequence of alternatives maximizes (minimizes) the right hand side of these expressions for every i. The set of alternatives to be used in all states at any given transition specifies a policy for that transition, and for the M successive transitions a M stage sequence of policies is determined by each of (2.34) and (2.35). Denote by A the M stage sequence of policies determined by Inequality (2.34) and by B the M stage sequence of policies determined by (2.35).

Define $p_{ij}^{X}(M)$, the M stage probability of transition from state i to state j given that policy sequence X is followed. That is, $p_{ij}^{X}(M)$, is the probability that if we start in state i and follow M stage policy sequence X, we will be in state j after the M transitions. We have

$$\sum_{j} p_{ij}^{X}(M) = 1 \quad \forall i, X \tag{2.36}$$

and by hypothesis there exists an $\epsilon \geq 0$ such that

$$p_{im}^{X}(M) = \sigma + \epsilon \geq \sigma > 0 \quad \forall i, X \tag{2.37}$$

Using this notation, we get the following inequality from (2.34)

$$\begin{aligned} g_i(n) &\leq \sum_{j} p_{ij}^{A}(M)\, g_j(n-M) \\ &= \sigma\, g_m(n-M) + \epsilon\, g_m(n-M) + \sum_{j \neq m} p_{ij}^{A}(M)\, g_j(n-M) \\ &\leq \sigma\, g_m(n-M) + (1-\sigma) \max_{j} [g_j(n-M)] \end{aligned} \tag{2.38}$$

Similarly from (2.35) we get

$$g_i(n) \geq \sigma\, g_m(n-M) + (1-\sigma) \min_{j} [g_j(n-M)] \tag{2.39}$$

Taking the maximum and minimum, respectively, of $g_i(n)$ in the two expressions we get

$$\max_{i} [g_i(n)] \leq \sigma\, g_m(n-M) + (1-\sigma) \max_{j} [g_j(n-M)] \tag{2.40}$$

$$\min_{i} [g_i(n)] \geq \sigma\, g_m(n-M) + (1-\sigma) \min_{j} [g_j(n-M)] \tag{2.41}$$

Subtracting (2.41) from (2.40) and shifting the arguments of the g_i, we get

$$\max_i[g_i(M)] - \min_i[g_i(M)] \leq (1-\sigma)\,(\max_j[g_j(0)] - \min_j[g_j(0)])$$

This may be iterated r times to yield

$$\max_i[g_i(rM)] - \min_i[g_i(rM)] \leq (1-\sigma)^r\,(\max_j[g_j(0)] - \min_j[g_j(0)])$$

and so

$$\lim_{r\to\infty}[\max_i g_i(rM) - \min_i g_i(rM)] = 0. \qquad (2.42)$$

This completes the proof of the theorem.

The two theorems just proved provide a practical method for use of the recurrence relation, (2.3), to determine an optimal or near optimal policy and to determine the gain of that policy to any desired accuracy in the infinite time transition optimal MRDP. The computational advantages of this method of solution are discussed in more detail later.

It should be noted that the hypothesis in Theorem 2.4 give sufficient conditions for the convergence of

$$\max_i g_i(n) - \min_i g_i(n) \quad \text{as } n \longrightarrow \infty.$$

Convergence may occur even when the theorem hypothesis is not satisfied, and in fact examples of this have been constructed. On the other hand, if we wish to determine whether convergence is assured we may find it difficult to determine whether or not the

hypothesis of the theorem is indeed satisfied. Therefore the remainder of this section is devoted to finding a necessary condition (Theorem 2.6) and a sufficient condition (Theorem 2.7) for satisfaction of the hypothesis. It should be straightforward to determine from physical aspects of the situation being modeled whether of not these conditions are satisfied. We also (Theorem 2.8) give special conditions under which the process may be altered to assure convergence without affecting the optimal gain or the optimal policy.

First we give some definitions from Markov chain theory [38]. In a Markov chain a state j is reachable (accessible) from state i if for some number of transitions $n \geq 1$, $p_{ij}(n) > 0$ where $p_{ij}(n)$ is the probability of going from state i to state j in n transitions. States i and j communicate if i is reachable from j and j is reachable from i. A state j is recurrent under policy α if $f_{jj}^{\alpha} = 1$ where f_{jj}^{α} is the probability of eventually making a transition to state j under policy α given that state j is currently occupied. There must exist at least one non-empty communicating recurrent class of states in any finite Markov chain.

In the work which follows we consider MRDP's in which the Markov chain determined by any stationary policy has only one communicating recurrent class of states. That is, the Markov chain is completely ergodic for any policy. A MRDP which satisfies this condition will be called a single chain MRDP. We now state a definition and a lemma.

Definition

In a single chain MRDP a set of states, R is called recurrent

core if this set of states is recurrent under every stationary policy. That is, $f_{kk}^{\alpha} = 1$ for all $k \in R$ and for all policies α.

Note that the states in R all communicate with one another.

Lemma 2.5

A single chain MRDP has a non-empty recurrent core.

Proof

Call the set of states comprising the recurrent core, R. Then we wish to prove $R \neq \phi$. Suppose $R = \phi$. Then there exists a policy β_1 with a non-empty set of recurrent states R_1 and a policy β_2 with a non-empty set of recurrent states R_2 such that

$$R_1 \cap R_2 = \phi.$$

This is shown graphically in Figure 2.1 where we see the transitions between classes of states possible under policies β_1 and β_2. In the figure T, is the set of states transient under both policies β_1 and β_2.

Now because alternatives may be chosen independently in each state, there exists a policy β_3 with alternatives from β_1 chosen in R_1 and T and alternatives from β_2 chosen in R_2. But under this policy, both R_1 and R_2 are recurrent classes and this contradicts the hypothesis that there is only one recurrent class per policy. The transitions possible under policy β_3 are shown in Figure 2.2 ||

The following theorem states that only in a single-chain MRDP does there exist a state m reachable from any arbitrary state regardless of the policy sequence chosen. (Actually a somewhat

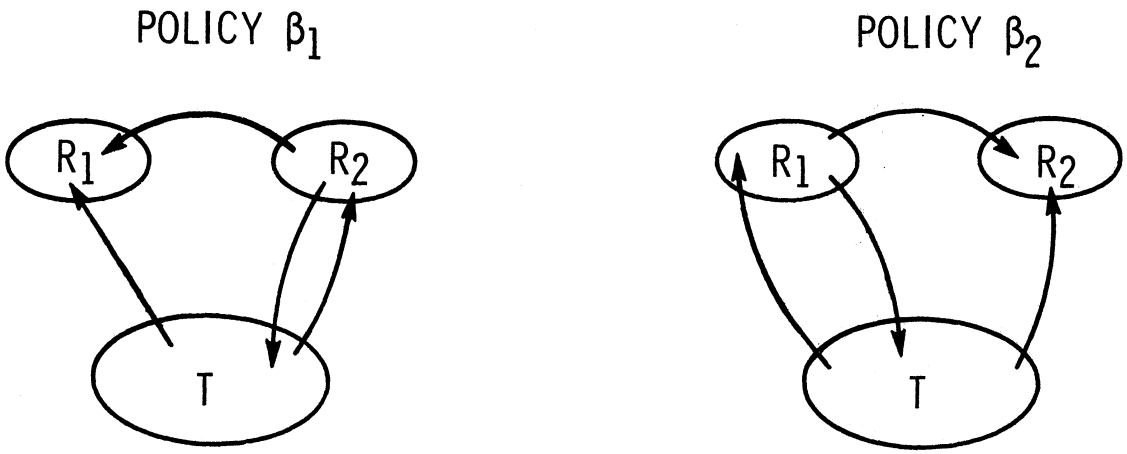

Figure 2.1. Disjoint Recurrent Classes of States Under Two Distinct Policies

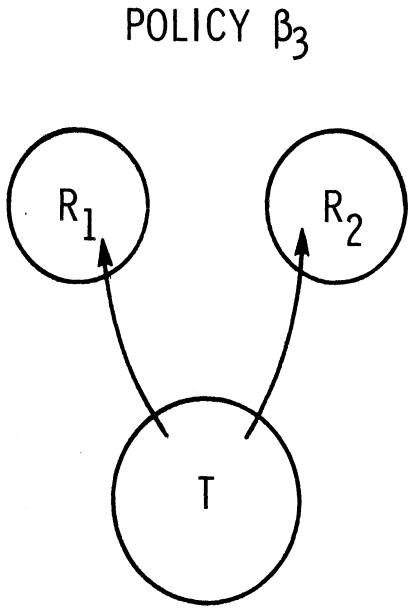

Figure 2.2 Two Recurrent Clas Under Policy β_3

stronger result is proved.) Thus for the hypothesis of Theorem 2.4 to hold, it is necessary that we have a single chain MRDP.

Theorem 2.6

In a MRDP, any state $m \epsilon R$ is reachable from an arbitrary state i regardless of the sequence of policies chosen if and only if the MRDP is single-chain. Furthermore, m is reachable in N-1 or less transitions.

Proof

"Only if" part is trivial. Simply choose a policy for which there is more than one Markov chain and then note that there is not state m reachable from every other state of the MRDP.

To prove any state $m \epsilon R$ is reachable when the MRDP is single chain, choose a state $m \epsilon R$. Suppose the system is initally in state i and some sequence of policies is used as transitions are made. We know that if the policy is stationary then all states in the recurrent core are reachable. We wish to show that this is still true when the policy changes on successive transitions. Assume we start with transition 0 and form sets of states S_n made up of all states reachable with probability greater than zero at transition n but not reachable at transitions $0, 1, \ldots, n-1$ under the sequence of policies used. Thus set S_0 contains the single state i.

Then for some r we have

$$S_r = \phi$$

$$S_{r-1} \neq \phi$$

$$\vdots$$

$$S_0 \neq \phi$$

Such an r must exist and furthermore $0 < r \leq N$. For convenience, define the set S to be the union of the sets above.

$$S = S_0 \cup S_1 \cup \cdots \cup S_{r-1}$$

It is clear that the proof will be complete if it can be shown that $m \in S$.

To show this, determine a stationary policy from the above dynamic policy by choosing the alternatives chosen in the state set S_n at transition n; $n=0,1,\ldots,r-1$; and arbitrary alternatives in states outside S. For this policy, some subset of the set of states S forms a closed, communicating class of states and so $R \subset S$. This is sufficient to prove $m \in S$ since by hypothesis, $m \in R$; and since $r \leq N$, state m is reachable in N-1 or less transitions.

The next theorem gives a sufficient condition for Theorem 2.4 to hold.

Theorem 2.7

In a single-chain MRDP if a state $m \in R$ exists such that

$p^k_{mm} > 0$ for all k, then

$$\max_i[g_i(n)] - \min_i[g_i(n)] \to 0 \text{ as } n \to \infty.$$

Proof

From Theorem 2.6, m is reachable in N-1 or less transitions under any policy. Since $p^k_{mm} > 0$, state m may be occupied with probability >0 after N-1 transitions regardless of the initial state and policy sequence. Therefore the hypothesis of Theorem 2.4 is satisfied.

Thus for convergence to occur it is only necessary to exhibit a state which is recurrent under any policy and which allows transitions to itself under any alternative.

Before turning to the time-optimal problem we look at one case in which the process may be altered to assure convergence of the bounds on the gain. Both the gain and the optimal policy of the orginal process may be determined from those of the derived process.

Theorem 2.8

If the one step rewards $r^k_{ii} = r^\ell_{jj} = r$ for all i,j,k, ℓ in a single chain MRDP, then the bounds on the gain converge in the process defined by setting

$$\hat{p}^k_{ij} = (1-\epsilon)\, p^k_{ij} + \epsilon\, \delta_{ij} \text{ for all } i,j,k \qquad (2.43)$$

$$0 < \epsilon < 1$$

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

Futhermore the gain of the original process for any policy β may be determined from the equation

$$g^{\beta} = \frac{\hat{g}^{\beta} - \epsilon r}{1-\epsilon} \quad , \tag{2.44}$$

and both processes have the same optimal policies. The new process defined by the $\hat{p}_{ij}^{k}$ will be called the derived process.

Proof

For an arbitrary policy β, define the N dimensional row vectors π^{β} and $\hat{\pi}^{\beta}$ to be the stationary state probabilities in the original process and derived process, respectively. Similarly, define P^{β} and $\hat{P}^{\beta}$ to be the transition probability matrices for these two processes. By definition of the stationary probabilities,

$$\hat{\pi}^{\beta} = \hat{\pi}^{\beta} \hat{P}^{\beta} \tag{2.45}$$

Equation (2.43) may be used to replace the $\hat{p}_{ij}^{k}$ in (2.45) to get

$$\hat{\pi}^{\beta} = (1-\epsilon)\hat{\pi}^{\beta} P^{\beta} + \epsilon \hat{\pi}^{\beta} I \tag{2.46}$$

where I is the N x N identity matrix. Thus it is seen that

$$\hat{\pi}^{\beta} = \hat{\pi}^{\beta} P^{\beta}$$

By definition also

$$\pi^{\beta} = \pi^{\beta} P^{\beta}$$

and since the stationary probabilities are unique,

$$\pi_i^{\beta} = \hat{\pi}_i^{\beta} \quad \text{for all i and } \beta.$$

For the derived process it is easily shown that

$$\hat{q}_i^{\beta} = (1-\epsilon)\, q_i^{\beta} + \epsilon\, r_{ii}^{\beta} \tag{2.47}$$

and from Theorem 2.1

$$\hat{g}^{\beta} = \sum_{i=i}^{N} \hat{\pi}_i^{\beta}\, \hat{q}_i^{\beta}.$$

Therefore

$$\begin{aligned} \hat{g}^{\beta} &= \sum_{i=i}^{N} \pi_i^{\beta}\, \hat{q}_i^{\beta} \\ &= \sum_{i=i}^{N} \pi_i^{\beta}\, [(1-\epsilon)\, q_i^{\beta} + \epsilon\, r_{ii}^{\beta}] \\ &= (1-\epsilon)\, g^{\beta} + \epsilon\, r \end{aligned} \tag{2.48}$$

where $r = r_{ii}^{\beta}$ and is constant for all i, β.

It is clear from the above equation that the same policies will be optimal in the derived process as are optimal in the original process since ϵ and r do not depend on β. Furthermore the gain of the original process is easily determined from that of the derived process by means of Equation (2.48). The bounds on the gain must converge in the derived process because $p_{ii}^{k} > 0$ for all i, k, and therefore the hypothesis of Theorem 2.7 is satisfied.

2.3 Time Optimal Problem

In the previous section it was shown how the recurrence relation in Equation (2.3) can be used to determine bounds on the gain of policies which maximize the return per transition in infinite time MRDP's. It was shown that these bounds are guaranteed to converge in certain cases and that they also bound the gain of the policy currently in use in the recurrence relation. Two objectives for MRDP's were given in the introductory section of this chapter: maximization of gain per transition and maximization of gain per unit time. Gain per unit time is called the gain rate and in this section bounds on the gain rate are given which are analogous to the bounds on the gain determined in the previous section. In addition conditions are given which guarantee convergence of these bounds. In this way the recurrence relation in Equation (2.4) is made useful for the infinite time problem when the goal is to maximize expected return per unit time.

The recurrence relation for the time-optimal problem was given in Equation (2.4) and is

$$w_i(t) = \max_k \sum_{j=1}^{N} p_{ij}^k \left\{ \int_t^\infty dF_{ij}^k(\tau) \left[R_{ij}^k(t \mid \tau) + S_{ij}^k(t,\tau) \right] + \int_0^t dF_{ij}^k(\tau) \left[R_{ij}^k(\tau) + w_j(t-\tau) \right] \right\} \qquad (2.4)$$

$w_i(t)$ is the expected return given that we start in state i and the process terminates after t time units. It is clear that it is neccessary to place restrictions on the distribution functions, $F_{ij}^k(t)$, in order

to be able to evaluate the Equations (2.4). In the work which follows $F_{ij}^{k}(t)$ are restricted to lattice distributions with distribution span Δ and for some integer $L<\infty$, $F_{ij}^{k}(L\Delta)=1$ for all i,j,k. Also instantaneous transitions are not allowed and so $F_{ij}^{k}(0)=0$ for all i,j,k.

Define $\phi_{ij}^{k}(t)$ to be the probability density function for the distribution function $F_{ij}^{k}(t)$. The times between transitions are restricted to discrete values and have discrete distributions so that the density functions may be defined by

$$\phi_{ij}^{k}(n\Delta) = F_{ij}^{k}(n\Delta) - F_{ij}^{k}(n\Delta-\Delta);$$

$$n = 1,2,\ldots,L \qquad (2.49)$$

$$\phi_{ij}^{k}(n\Delta) = 0 \quad \text{elsewhere.}$$

It follows immediately that for $n \geq L$, the recurrence relation may be written

$$w_i(n\Delta) = \max_{k} \sum_{j=1}^{N} p_{ij}^{k} \sum_{\ell=1}^{L} \phi_{ij}^{k}(\ell\Delta)\,[\,R_{ij}^{k}(\ell\Delta) + w_j(n\Delta-\ell\Delta)\,];$$

$$n \geq L \qquad (2.50)$$

By definition, $q_i^k = \sum_{j=1}^{N} p_{ij}^{k} \sum_{\ell=1}^{L} \phi_{ij}^{k}(\ell\Delta)\, R_{ij}^{k}(\ell\Delta)$ and furthermore the distribution span, Δ, can be set equal to one without loss of generality. Also define $d_{ij}^{k}(\ell) = p_{ij}^{k}\,\phi_{ij}^{k}(\ell)$. Then

$$w_i(n) = \max_{k} \{\, q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{k}(\ell)\, w_j(n-\ell)\}; \; n \geq L \qquad (2.51)$$

Since the process of interest is to run for infinite time, the total expected return will be infinite, and so the goal is to maximize expected return per unit time. For this situation, any finite boundary reward, $S_{ij}^{k}(t,\tau)$, will have no effect on the gain rate and so $S_{ij}^{k}(t,\tau)$ is set equal to zero for convenience. Furthermore let $R_{ij}^{k}(t|\tau) = R_{ij}^{k}(\tau)$ which means that the entire reward for making a transition from i to j under alternative k is collected at the beginning of the transition interval It is necessary to set $w_i(0)$ in order to evaluate the recurrence relations and therefore let $w_i(0) = 0$ for all i. None of the above assumptions are necessary but only serve to make the form of the recurrence relation simpler for values of $n<L$. Thus with these assumptions

$$w_i(0) = 0, \quad \text{for all } i$$

$$q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{\min[n,L]} d_{ij}^k(\ell)\, w_j(n-\ell)\,; \qquad n = 1,2,\ldots \qquad (2.52$$

It is clear that this expression may be easily evaluated for any integer $n>0$. In the particular case where L=1, the time-optimal problem and the transition-optimal problem are identical because all transitions take equal time. For this case $\phi_{ij}^{k}(1) = 1$ for all i, j, k so that

$$w_i(n) = \max_k \left[q_i^k + \sum_{j=1}^{N} p_{ij}^k\, w_j(n-1) \right].$$

This is, of course, the recurrence relation for the transition-optimal problem.

Before proceeding to a derivation of bounds on the gain rate it is desirable to define two new variables.

Let

$$h_i(z) = w_i(z) - w_i(z-L);\ z \geq L \tag{2.53}$$

A column vector of these $h_i(z)$ of dimension N is defined as

$$H(z) = \begin{pmatrix} h_1(z) \\ \vdots \\ h_N(z) \end{pmatrix} \quad ; \quad z \geq L \tag{2.54}$$

Next for a fixed constant $n_0 \geq L$ define a $L \cdot N$ dimensional column vector with the H-vectors defined above as components.

$$F(n) = \begin{pmatrix} H(n_0+nL) \\ H(n_0+nL+1) \\ \vdots \\ H(n_0+nL+L-1) \end{pmatrix} = \begin{pmatrix} f_1(n) \\ f_2(n) \\ \vdots \\ f_{LN}(n) \end{pmatrix} \quad ;$$

$$n = 0, 1, 2, \ldots;\ n_0 \geq L \tag{2.55}$$

It will be obvious in what follows why the variables defined above are desirable.

Next a definition is given which allows a simple explicit statement of the choice of alternatives at successive stages in the recurrence relation. Recall that a policy, α, was defined as a particular choice of alternative in each of the N states. An extension of this over a number of stages follows.

Definition

An L-stage policy sequence, A, is defined as a choice of policies $\alpha_0, \alpha_1, \ldots, \alpha_{L-1}$ in the L successive stages of the recurrence relation (2.52). A stationary L stage policy sequence, A, is a repetitive choice of the above policies at successive stages so that the policy sequence is $\alpha_0, \alpha_1, \ldots, \alpha_{L-1}, \alpha_0, \alpha_1, \ldots, \alpha_{L-1}, \alpha_0, \ldots$. This stationary sequence may begin at any desired stage.

Evaluation of the recurrence relation for $w_i(n)$ up to some n determines the optimal sequence of policies to be used for the first n stages of the process. Instead of choosing the alternatives at each stage which maximize expected return, it is possible to choose other alternatives in which case the recurrence relation can be used to find expected return for the alternatives chosen. This can be formalized by defining a new variable $w_i^A(n)$. As in the definition of $w_i(n)$, the i refers to the initial state while the n refers to the number of stages remaining before termination of the process. For $w_i^A(n)$, however, the optimal policy sequence as determined from Equation (2.52) is used before stage n_0 where n_0 is some arbitrary constant such that $n_0 \geq L$. Beginning at stage n_0, the stationary L stage policy sequence A is used. As stated above, n_0 is an arbitrary constant such that $n_0 \geq L$ and so $w_i^A(n)$ is actually a function of n_0 as well, but this is not shown explicitly. The recurrence relation for $w_i^A(n)$ is written in terms of the alternatives which make up the policy sequence A.

These policies are $\alpha_0, \alpha_1, \ldots, \alpha_{L-1}$ and for policy α_x the alternative in state i will be called $a_x(i)$. Where no confusion can result, the functional dependence on i will be omitted. Up to stage n_0, the optimal policy sequence is followed and therefore

$$w_i^A(n) = \max_k \left[q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{\min[n, L]} d_{ij}^k(\ell)\, w_j^A(n-\ell) \right];$$

$$n < n_0 \tag{2.56}$$

From stage n_0 on, the L stage stationary policy sequence A is used and so

$$w_i^A(n_0 + xL + n) = q_i^{a_n} + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_n}(\ell)\, w_j^A(n_0 + xL + n - \ell);$$

$$x = 0, 1, 2, 3, \ldots; \quad n = 0, 1, 2, \ldots, L \tag{2.57}$$

In the obvious way, $h_i^A(n)$ is defined by

$$h_i^A(n) = w_i^A(n) - w_i^A(n-L);$$

$$n \geq L \tag{2.58}$$

and $H^A(n)$, $F^A(n)$, and $f_i^A(n)$ are defined analogously to $H(n)$, $F(n)$, and $f_i(n)$ in Equations (2.54) and (2.55).

The reason for an interest in a stationary L stage policy sequence is that when such a policy sequence is used a simple recurrence relation

for the $h_i^A(n)$ can be obtained. This in turn leads to a coefficient matrix for the $h_i^A(n)$ and ultimately for the $f_i^A(n)$. This coefficient matrix is stochastic and is used in the proofs which exhibit bounds on the gain rate and show that these bounds converge. The $f_i(n)$ are shown to be closely related to the gain rate and the bounds on it.

As a first step in obtaining the desired recurrence relation on the $h_i^A(n)$, we have

$$h_i^A(n_0+L+n) = w_i^A(n_0+L+n) - w_i^A(n_0+n)$$

$$= \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_n}(\ell)\left[w_j^A(n_0+L+n-\ell) - w_j^A(n_0+n-\ell)\right]$$

$$= \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_n}(\ell)\, h_j^A(n_0+L+n-\ell);$$

$$n = 0, 1, \ldots, L-1 \tag{2.59}$$

Again n_0 is a constant such that $n_0 \geq L$. The goal is to obtain each of $h_i^A(n_0+x\,L+n)$ for any integer $x \geq 0$ in terms of $h_i^A(n_0+(x-1)L+n)$ for $i=1,\ldots,N$ and $n=0,1,\ldots,L-1$. For $n=0$, our desired relation is

$$h_i^A(n_0+xL) = \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_0}(\ell) h_j^A(n_0+x\,L-\ell) \tag{2.60}$$

For $n=1$, the relation

$$h_i^A(n_0+x\,L+1) = \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_1}(\ell)\, h_j^A(n_0+xL+1-\ell) \tag{2.61}$$

may be obtained from Equation (2.57). In this case it is seen that the right hand side contains $h_j^A(n_0+xL)$; $j=1,2,\ldots,N$; whereas an equation is desired which contains $h_j^A(z)$ with arguments no larger than $z=n_0+xL-1$. To obtain this, substitute the right side of Equation (2.60) into Equation (2.61) to get

$$h_i^A(n_0+xL+1) = \sum_{j=1}^{N}\left[\sum_{\ell=2}^{L} d_{ij}^{a_1}(\ell)\, h_j^A(n_0+xL+1-\ell) + d_{ij}^{a_1}(1)\sum_{k=1}^{N}\sum_{\ell=1}^{L} d_{jk}^{a_0}(\ell)\, h_k^A(n_0+xL-\ell)\right] \qquad (2.62)$$

The same procedure may be followed to obtain $h_i^A(n_0+xL+2)$ in terms of $h_j^A(n_0+(x-1)L+n)$; $i=1,\ldots,N$; $n=0,1,\ldots,L-1$. In this case the first two terms in the summation over ℓ must be replaced by the expressions found for $h_i^A(n_0+xL)$ and for $h_i^A(n_0+xL+1)$ in Equations (2.60) and (2.62). This procedure is repeated for $h_i^A(z)$ up to $z=n_0+xL+L-1$.

In order to simplify notation the relationships above will be given in matrix notation. Define the matrix of $d_{ij}^{a_x}(\ell)$ for policy α_x to be $D_x(\ell)=(d_{ij}^{a_x}(\ell))$. Then the vectors $H^A(n_0+xL+n)$ may be written as follows

$$H^A(n_0+xL) = \sum_{\ell=1}^{L} D_0(\ell)H^A(n_0+xL-\ell) \qquad (2.63)$$

$$H^A(n_0+xL+1) = \sum_{\ell=2}^{L} D_1(\ell)H^A(n_0+xL+1-\ell)+D_1(1)H^A(n_0+xL) \qquad (2.64)$$

$$H^A(n_0+xL+2) = \sum_{\ell=3}^{L} D_2(\ell)H^A(n_0+xL+2-\ell)$$

$$+ D_2(2)H^A(n_0+xL)+D_2(1)H^A(n_0+xL+1) \qquad (2.65)$$

$$\vdots$$

$$H^A(n_0+xL+L-1) = D_{L-1}(L)H^A(n_0+xL-1)+D_{L-1}(L-1)H^A(n_0+xL)$$

$$+ \cdots + D_{L-1}(1)H^A(n_0+xL+L-2) \qquad (2.66)$$

In each of these equations, the terms which must be written in terms of preceding equations have been taken out of the summation over L. The general equation may be written

$$H^A(n_0+xL+n) = \sum_{\ell=n+1}^{L} D_n(\ell)H^A(n_0+xL+n-\ell)$$

$$+ \sum_{\ell=1}^{n} D_n(\ell)H^A(n_0+xL+n-\ell)$$

$$n = 0, 1, \ldots, L-1 \qquad (2.67)$$

The $H^A(n_0+xL+n-\ell)$ in the second summation must all be replaced by expressions for these given in the equations for smaller values of n.

Next, recall the definition for $F^A(x)$:

$$F^A(x) = \begin{pmatrix} H^A(n_0+xL) \\ \vdots \\ H^A(n_0+xL+L-1) \end{pmatrix} = \begin{pmatrix} f_1^A(x) \\ \vdots \\ f_{L\cdot N}^A(x) \end{pmatrix}$$

$$; \; x = 0, 1, 2, \ldots \tag{2.68}$$

In terms of this definition, Equations (2.67) give $F^A(x)$ in terms of $F^A(x-1)$ for $x=1, 2, \ldots$. The LNxLN matrix of coefficients for $F^A(x-1)$ will be called U^A and so

$$F^A(x) = U^A F^A(x-1); \; x=1, 2, \ldots . \tag{2.69}$$

This matrix U^A may be partitioned into L^2 submatrices each of dimension NxN. Thus

$$U^A = \begin{bmatrix} U_{11}^A & U_{12}^A & \cdots & U_{1L}^A \\ U_{21}^A & & & \vdots \\ \vdots & & & \vdots \\ U_{L1}^A & & \cdots & U_{LL}^A \end{bmatrix}$$

Row i ($1 \leq i \leq N$) of this matrix, U^A, may be obtained from the equation

$$h_i^A(n_0+xL) = \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_0}(\ell) h_j^A(n_0+xL-\ell)$$

The elements of this row of U^A sum to one:

$$\sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^{a_0}(\ell) = 1 \qquad (2.70)$$

Similarly, row i+N ($1 \leq i \leq N$) may be obtained from Equation (2.62) and the coefficients again sum to one.

$$\sum_{j=1}^{N} \left[\sum_{\ell=2}^{L} d_{ij}^{a_1}(\ell) + d_{ij}^{a_1}(1) \sum_{k=1}^{N} \sum_{\ell=1}^{L} d_{jk}^{a_0}(\ell) \right] = 1$$

The matrix U^A is given explicitly in Figure 2.3 in terms of the submatrices U_{IJ} where each row of submatrices except the first contains submatrices defined in previous rows. It is now shown that each row of matrix U^A has row sum equal to one and so the matrix U^A is a stochastic matrix.

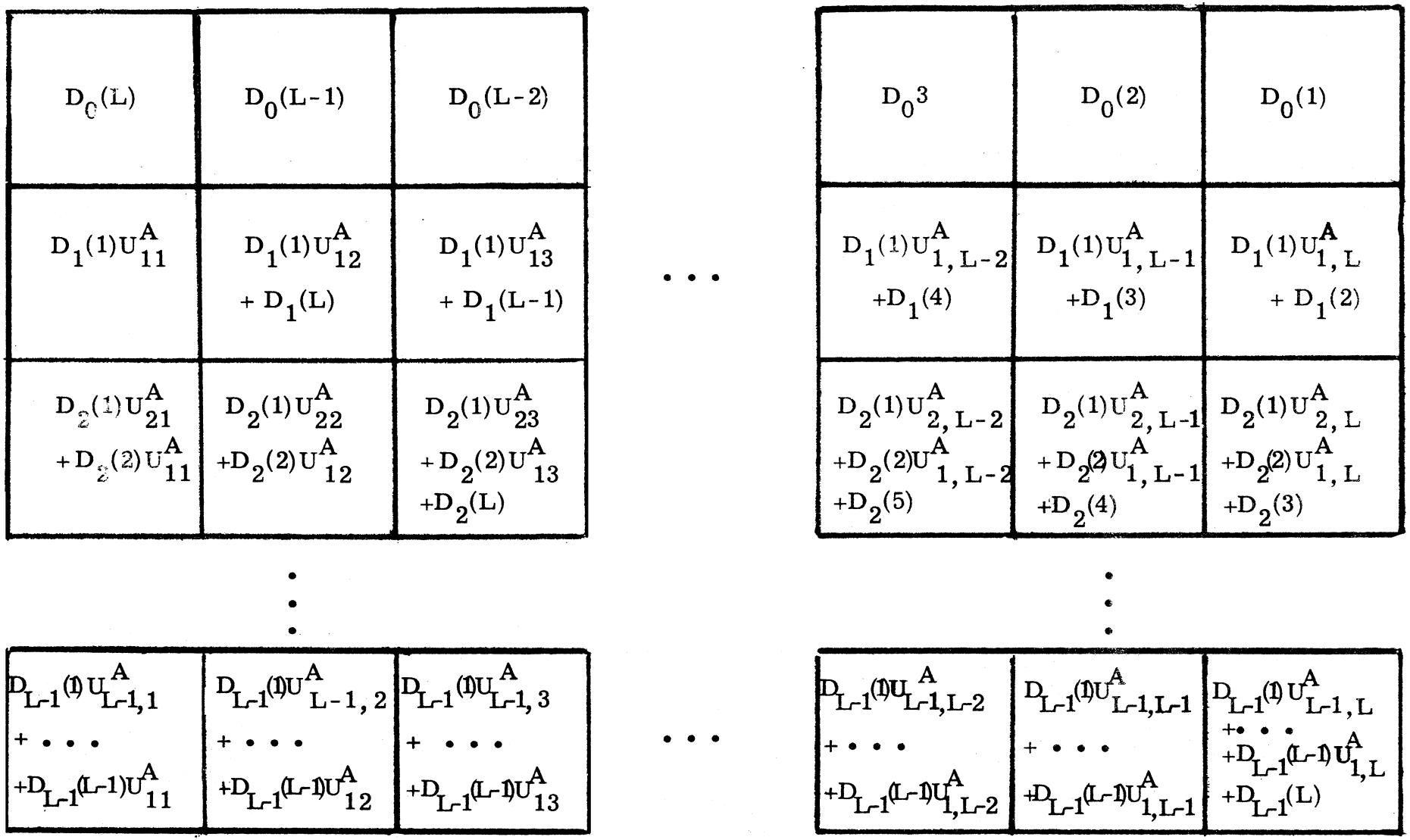

Figure 2.3. The Matrix U.

Theorem 2.9

Let U^A be the LNxLN matrix of coefficients of $F^A(x-1)$ in the equations

$$F^A(x) = U^A F^A(x-1); \quad x = 1, 2, \ldots \quad . \tag{2.69}$$

where A is used to specify an arbitrary stationary L stage policy sequence as discussed earlier. Then U^A is a stochastic matrix, and all its row sums are one.

Proof

It has already been shown that for the first 2N rows of matrix U^A the row sums are one. These 2N rows correspond to the first two rows of the NxN submatrices which comprise matrix U^A. Thus, the row sums in submatrix row 1 are the same as the row sums for the matrix

$$P_0 = \sum_{\ell=1}^{L} D_0(\ell) \quad \text{where}$$

$P_0 = (p_{ij}^{a_0})$ is the NxN transition probability matrix for policy α_0.

Next it will be shown that if the row sums in the submatrix rows $1, \ldots, I$ are one then the row sums in submatrix row I+1 must be one, and thus that U^A is stochastic. It is easily verified that if an arbitrary NxN matrix B with row sum in row i equal to b_i is post-multiplied by an

matrix BS will have sum b_i. In submatrix row I+1 of U^A, $D_I(I+1),\ldots,$ $D_I(L)$ each appear once and are not multiplied by any other matrix while $D_I(1),\ldots,D_I(I)$ each appear L times and are each post-multiplied by matrices whose sum forms a stochastic matrix. The contribution to the row sums in submatrix row I+1 of $D_I(J)$; for some $J \leq I$; is that of the matrix

$$D_I(J) \sum_{K=1}^{L} U_{I-J+1,\,K}.$$

But since $\sum_{K=1}^{L} U_{I-J+1,K}$ is a stochastic matrix, the row sums are the same as those of $D_I(J)$ and since the row sums of $\sum_{J=1}^{L} D_I(J)$ are 1, then the row sums in submatrix row I+1 must be 1. Thus U^A is a stochastic matrix with row sums one.

In order to be useful in the derivation of bounds on the gain rate and convergence of these bounds, the matrix U^A must have certain properties which hold regardless of the L stage policy sequence A used to determine the matrix. More specifically, there is a LN state homogeneous Markov chain induced by the matrix U^A and more generally a LN state non-homogeneous Markov chain induced by using matrices $U^{A_1}, U^{A_2},\ldots$ as transition probability matrices for successive state transitions in a LN state process where $A_1, A_2,\ldots$ is a sequence of L stage policy sequences. It is these Markov chains which are required to possess certain properties, one of which is that for arbitrary L stage policy sequence A, the Markov chain induced by U^A must possess a stationary distribution which will be denoted by the LN dimensional row vector, ρ^A.

The other property needed is that there must exist some state, b, in the LN state process induced by U^{A_1}, U^{A_2}, ... which is reachable from any state after some number M of transitions regardless of the $A_1, A_2, \ldots$ chosen. This second property implies the first because the reachable state, b, exists for $A=A_1=A_2=\ldots$ and this state may be occupied with probability greater than zero for all transitions after the Mth. Therefore the process induced by U^A is acyclic and contains a single closed communicating class of states which means the process has a long-run distribution and therefore a stationary distribution [38] .

Verification directly from the U^A matrix that the properties given above hold may be inconvenient and so in this paragraph restrictions on the density functions for the time between transitions, $\phi_{ij}^k(\ell)$, and on the transition probabilities in the underlying Markov chain of the MRDP, p_{ij}^k, are given which ensure that U^A has the needed characteristics. First of all, the restriction is placed on the underlying Markov chain of the MRDP that it be single chain as defined in section 2. 2 and that for some state $m \epsilon R$, the recurrent core, $p_{mm}^k > 0$ for all k. This condition is identical to the one used in Theorem 2. 7. If now the $\phi_{ij}^k(\ell)$ are restricted so that $\phi_{ij}^k(1) > 0$ for all i, j, k; then state b=m+(L-1)N is reachable after N-1 or less transitions regardless of the policy sequences $A_1, A_2, \ldots$ used to induce the Markov chain. This may be stated formally as a theorem.

Theorem 2. 10

Given a single chain MRDP with recurrent core R and with a

state $m \epsilon R$ such that $p^k_{mm} > 0$ for all k. If $\phi^k_{ij}(1) > 0$ for all i,j,k; then state $b = m+(L-1)N$ is reachable in N-1 or less transitions in any Markov chain induced by using matrices $U^{A_1}, U^{A_2}, \ldots$ as transition probability matrices for successive transitions where $A_1, A_2, \ldots$ are arbitrary L stage policy sequences.

Proof

Define P_x as the transition probability matrix of the underlying Markov chain of the MRDP when policy α_x is in use. In theorem 2.6 it was shown that for the given conditions on the MRDP, state m is occupied with probability greater than zero after N-1 or less transitions regardless of the initial state and regardless of the policies used on successive transitions. A necessary and sufficient condition for state m to be reachable in this fashion is that all elements in the mth column of the product matrix, $\prod_{j=1}^{N-1} P_{x_j}$, be greater than zero for any arbitrary set of policies, $\alpha_{x_1}, \ldots, \alpha_{x_{N-1}}$.

Next inspect the matrix U^A and note that the theorem is proved if it can be shown that the product matrix, $\prod_{i=1}^{N-1} U^{A_i}$, has all elements in column $b = m+(L-1)N$ greater than zero regardless of the policy sequences, $A_1, \ldots A_{N-1}$, chosen. If the product matrix, $\prod_{i=1}^{N-1} U^{A_i}$, is written in terms of submatrices as U^A is in Figure 2.3, then it is seen that all elements in column b are greater than zero if and only if all elements in the mth column of each submatrix in the right most column of submatrices is greater than zero. Therefore attention may be restricted to this column

of submatrices in the following.

From the hypothesis and the fact that $d_{ij}^k(1) = \phi_{ij}^k(1)\, p_{ij}^k$, it is clear that $d_{ij}^k(1) > 0$ if $p_{ij}^k > 0$. Therefore $D_I(1)$; $0 \leq I \leq L-1$; in matrix U^A has non-zero elements in the same positions within the matrix as P_I, the transition probability matrix in the MRDP for policy α_I. Since a N-1 order product of P-matrices has all elements in column m greater than zero, then a N-1 order product of D(1) matrices must have all elements in column m greater than zero. For submatrix U_{IL}^A; $1 \leq I \leq L$; in U^A it is seen that one term consists of the I order product $D_{I-1}(1) \ldots D_0(1)$ and in particular submatrix U_{LL}^A contains the term $D_{L-1}(1) \ldots D_0(1)$. Therefore inspection of U^A shows that the product of two matrices of this form has a L+I order product of D(1) matrices in submatrix row I and column L. The product of N-1 matrices of the form of U^A has a L(N-2)+I order product of D(1) matrices in the submatrix in row I and column L. Since $I \geq 1$ and $L \geq 1$ the order of this product is N-1 or greater and so all elements in column m, of this submatrix are greater than zero. This holds for all I; $1 \leq I \leq L$; and therefore the product of N-1 matrices of form U^A has all elements in column b greater than zero which was to be proved.

The restriction on the $\phi_{ij}^k(1)$ can be relaxed still further by requiring only that $\phi_{mm}^k(1) > 0$ for all k. With this condition and the condition $p_{mm}^k > 0$ it can still be shown that state $b = m+(L-1)N$ can be occupied after N or less transitions regardless of the initial state.

The argument needed to shown this is quite complex and is therefore relegated to Appendix E.

The preceding discussion shows that under reasonable conditions on the variables p_{ij}^k and $\phi_{ij}^k(\ell)$ in the MRDP, it can be guaranteed that some state b in a process induced by matrix U^A for any L stage policy sequence A is occupied with probability greater than zero after N-1 or less transitions of this process. Suppose now that a MRDP is given and so under some L stage policy sequence, A, a matrix U^A is determined. Assume there exists a state b which is reachable from all states of the Markov chain induced by U^A; then this Markov chain has only one communicating class of states. Therefore the Markov chain has a stationary distribution and the row vector of state probabilities of dimension LN making up this distribution will be denoted by ρ^A. In this U^A- induced process the equation

$$F^A(n) = U^A F^A(n-1);\ n \geq 1 \tag{2.71}$$

holds by definition of the matrix U^A. Also, of course, $\rho^A = \rho^A U^A$ by definition of the stationary distribution. In Theorem 2.1 it was shown that $\lim_{n\to\infty} f_i^A(n)$ exists and is independent of i. Define h^A as this limit: $h^A = \lim_{n\to\infty} f_i^A(n)$. In what follows h^A will be related to the gain rate but first the following lemma is shown to be true.

Lemma 2.11

$$\rho^A F^A(n) = h^A; \quad n=0,1,\ldots \quad . \tag{2.72}$$

Proof

The proof follows immediately from the relations among $\rho^A, F^A(n)$, and U^A given above. Thus

$$\rho^A F^A(n) = \rho^A U^A F^A(n-1) = \rho^A F^A(n-1)$$

$$n = 1, 2, \ldots \quad . \tag{2.73}$$

Therefore $\rho^A F^A(n)$ is a constant for all n and since $\lim_{n\to\infty} f_i^A(n)$ is independent of i then

$$\lim_{n\to\infty} [\rho^A F^A(n)] = h^A \tag{2.74}$$

and the lemma follows. ||

For the time-optimal problem the quantity to be maximized is the gain per unit time called the gain rate. In this paragraph the variable h^A as determined for L stage policy sequence A is related to the gain rate in the MRDP when L stage stationary policy sequence A is used. The gain rate for this policy sequence will be denoted by ψ^A. Recall that by definition (see Equations (2.53) - (2.58)) $f_i^A(n)$; $1 \leq i \leq N; n \geq 0$; can be written as $w_j^A(z) - w_j^A(z-L)$ for some j and z. In the lemma just proved, it was shown that

$$\lim_{z\to\infty} [w_j^A(z) - w_j^A(z-L)] = h^A \quad \text{for all j.} \tag{2.75}$$

Equation (2.75) states that for the infinite time process under stationary L stage policy sequence A the expected return in L time units is h^A. Therefore the gain per unit time is

$$\psi^A = \frac{h^A}{L} \tag{2.76}$$

The next theorem exhibits bounds on the gain rate of the current L stage policy sequence and on the optimal policy for the MRDP. This theorem is analogous to Theorem 2.3 for the gain. These bounds are given in terms of elements of vectors H(z), and since these elements are of the form $w_i(z) - w_i(z-L)$, they are easy to determine from the recurrence relation for the MRDP. Before stating Theorem 2.12 formally, some explanatory material is presented which should make the theorem easier to understand.

Suppose the recurrence relation

$$w_i(n) = \max_k \left[q_i^k + \sum_{j=1}^{\min[n,L]} d_{ij}^k (\ell)\ w_j(n-\ell) \right] \tag{2.52}$$

is used up to some stage n_0+L-1; $n_0 \geq L$; in a MRDP. The recurrence equation specifies a particular L stage policy sequence for the last L stages (stage n_0 to stage n_0+L-1), and this policy sequence may be used as a stationary L stage policy sequence by using it repetitively beginning at stage n_0. Use of Equation (2.52) in this way consitutes a particular way of choosing a stationary L stage policy sequence, A, to be used beginning at stage n_0; and this policy sequence in turn defines a matrix U^A. It was stated earlier that it has been shown [16] that an optimal

stationary policy exists for the infinite time problem, and therefore an optimal stationary L stage policy sequence exists in which the policies are identical at each step in the sequence. Call this optimal stationary policy sequence $*$; by use of this policy sequence a matrix U^* is defined in the usual way. With these definitions and concepts in mind, Theorem 2.12 may be stated as follows.

Theorem 2.12

Let the recurrence relation (2.52) be used up to some stage n_0+L-1; $n_0 \geq L$; in a MRDP in which some L stage policy sequence which we will call A, maximized expected return for the last L stages. If a stationary distribution with probabilities ρ^A exists for the Markov chain induced by the matrix U^A, and a stationary distribution ρ^* exists for the Markov chain induced by the matrix U^* where $*$ is some optimum stationary policy sequence, then

$$\max_{\substack{1\leq i\leq N \\ 0\leq n\leq L-1}} \left[h_i(n_0+n) \right] \geq h^* \geq h^A \tag{2.77}$$

$$\min_{\substack{1\leq i\leq N \\ 0\leq n\leq L-1}} \left[h_i(n_0+n) \right] \leq h^A \leq h^* \tag{2.78}$$

and thus, the gain per unit time for the current policy sequence and for the optimal policy are bounded by

$$\max_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[\frac{h_i(n_0+n)}{L}\right] \geq \psi^* \geq \psi^A \tag{2.79}$$

$$\min_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[\frac{h_i(n_0+n)}{L}\right] \leq \psi^A \leq \psi^* \tag{2.80}$$

Proof

In the proof which follows, it is shown that

$$\min_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[h_i(n_0+n)\right] \leq h^A \leq \max_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[h_i(n_0+n)\right]$$

and that

$$\min_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[h_i(n_0+n)\right] \leq h^* \leq \max_{\substack{1\le i\le N\\ 0\le n\le L-1}} \left[h_i(n_0+n)\right] .$$

These inequalities are sufficient to prove the theorem since by definition an optimal policy is one which maximizes the gain rate, and therefore $h^A \leq h^*$.

Because policy sequence A was used for the last L stages, the vector of $h(n_0+n)$ over which the maximum and minimum are to be obtained defines $F^A(0)$:

$$F^A(0) = \begin{pmatrix} H(n_0) \\ \vdots \\ H(n_0+L-1) \end{pmatrix} \tag{2.81}$$

Therefore we can work with $F^A(0)$ rather than $H(n_0+n)$; $0 \leq n \leq L-1$. It was shown in Lemma 2.11 that for stationary probability distribution ρ^A,

$$\rho^A F^A(0) = h^A \tag{2.82}$$

and therefore the inequalities in the theorem pertaining to the current L stage policy sequence must hold. Of course, if the current L stage policy sequence is an optimal stationary policy sequence for the infinite time process, then A is such that h^A is maximum so that $h^A = h^*$ and the proof is complete. Otherwise $h^A < h^*$ and it is necessary to prove that the inequalities hold which contain h^*. This is now done.

The proof of the inequalities containing h^* is by contradiction. Let $f_i^A(0) > h^*$ for all i. Then

$$h^A = \rho^A F^A(0) > h^*$$

which is a contradiction because the gain rate of the policy * must be at least as great as the gain rate of the policy A since *

was assumed to be optimal. Therefore, for some i, $f_i^A(0) \leq h^*$. It remains to show that $f_i^A(0) \geq h^*$ for some i.

Let $f_i^A(0) < h^*$ for all i. This implies that $\rho^* F^A(0) < h^*$. By definition, each $f_i^A(0)$ is equal to $w_j(z) - w_j(z-L)$ for some j; $1 \leq j \leq N$; and for some z; $n_0 \leq z \leq n_0+L-1$; and the recurrence relation which determines $w_j(z)$ gives

$$f_i^A(0) = \max_k \left[q_j^k + \sum_{r=1}^{N} \sum_{\ell=1}^{L} d_{jr}^k(\ell)\, w_r(z-\ell) \right] - w_j(z-L) \tag{2.83}$$

The alternative, k, chosen in state j at step z makes up one part of the current policy sequence, A, and it is seen that this alternative is chosen to maximize $f_i^A(0)$. Instead of choosing alternatives over each of the last L stages which maximize the $f_i^A(0)$, it would have been possible to choose alternatives in each state which make up an optimal stationary policy for the infinite time process and thereby define a vector with elements which may be called $f_i^*(0)$. Obviously $f_i^A(0) \geq f_i^*(0)$ for all i. (2.84) But this inequality contradicts the assumption that $\rho^* F^A(0) < h^*$ because it means that

$$\rho^* F^A(0) \geq \rho^* F^*(0) = h^* \tag{2.85}$$

This completes the proof of the theorem. ||

The next theorem shows that the bounds on the gain rate determined above converge to a common value as the number of steps taken in the

recurrence relation increases provided certain conditions on the possible forms of the matrices U^A are met. These conditions are sufficient to insure convergence of the bounds; convergence may occur, however, even if the conditions are not met. Therefore the usefulness of the recurrence relation (2.52) as a method of finding the gain rate and optimal policy for the infinite time process is not limited only to situations where the bounds can be shown to converge by the following theorem.

Theorem 2.13

Given a MRDP and an arbitrary sequence of L stage policy sequences $A_1, A_2, \ldots$. If the Markov chain induced by using $U^{A_1}, U^{A_2}, \ldots$ as transition probability matrices for successive transitions is such that in this L·N state process there exists a state b which may be occupied with probability $\geq \sigma > 0$ after B transitions regardless of the policy sequences $A_1, A_2, \ldots$ used and the initial state in the U-induced process, then

$$\max_{\substack{1\leq i\leq N \\ 1\leq x\leq L}} [h_i(n+x)] - \min_{\substack{1\leq i\leq N \\ 1\leq x\leq L}} [h_i(n+x)] \to 0 \text{ as } n\to\infty \qquad (2.86)$$

and thus the bounds on the gain rate determined in Theorem 2.12 are guaranteed to converge.

Proof

The proof of this theorem is similar to the proof of Theorem 2.4. Before proceeding with the proof, some new notation is introduced. For policy sequence A_I, define $\alpha_{I0}, \ldots, \alpha_{I,\, L-1}$ to be the L policies used in this policy sequence. Then define $a_{IJ}(i)$ as the alternative chosen in state i for policy α_{IJ}. This defines

$$d_{ij}^{\,a_{IJ}(i)}\ (\ell)$$

for policy α_{IJ} and for all i, j, and ℓ. For any ℓ, the N×N matrix of the

$$d_{ij}^{\,a_{IJ}(i)}\ (\ell)$$

will be written

$$D_{IJ}(\ell) = \left(d_{ij}^{\,a_{IJ}(i)}\ (\ell)\right).$$

If it is desirable to indicate some general policy κ we write $D_\kappa(\ell)$. Other definitions will be introduced when needed. As usual, n_0 is restricted to a value equal to or greater than L. From the definition of $h_i(z)$; $z \geq L$; in terms of $w_i(z)$ and $w_i(z - L)$;

$$h_i(n_0 + xL + n) = \max_k \left[q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\, w_j(n_0 + xL + n - \ell) \right]$$

$$- \max_k \left[q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\, w_j(n_0 + (x-1)L + n - \ell) \right];$$

$$x = 1, 2, \ldots; \; n = 0, 1, \ldots, L-1. \qquad (2.87)$$

The maximum over all possible alternatives may be found for all terms together on the right side of Equation (2.87) above, and this gives the following inequality which is analogous to Equation (2.32).

$$h_i(n_0 + xL + n) \leqq \max_k \left[\sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\, h_j(n_0 + xL + n - \ell) \right];$$

$$x = 1, 2, \ldots; \; n = 0, 1, \ldots, L-1; \; i = 1, 2, \ldots, N. \qquad (2.88)$$

The above inequality as a vector relation may be written as

$$H(n_0 + xL + n) \leqq \max_\kappa \left[\sum_{\ell=1}^{L} D_\kappa(\ell)\, H(n_0 + xL + n - \ell) \right]. \qquad (2.89)$$

where the maximum is over all policies κ or in other words over all

alternatives in each state such that each element of the vector on the right is maximized. Thus the inequality holds for all N elements of the vectors.

Using the inequality relationships above, it is now possible to obtain a $L \cdot N$ dimensional vector inequality in which $H(n_0 + xL), \ldots, H(n_0 + xL + L-1)$ appear on the left and $H(n_0 + xL - L), \ldots, H(n_0 + xL - 1)$ appear on the right. To simplify notation, assume that in the inequality (2.89) for $H(n_0 + xL + n)$; the policy which maximizes the right side is α_{xn}.

In the relations which follow; vectors $H_0(n_0 + xL + n)$ are defined by the quantities they are set equal to. These definitions serve to simplify the notation. Thus

$$H(n_0 + xL) \leqq \sum_{\ell=1}^{L} D_{x0}(\ell)\, H(n_0 + xL - \ell) = H_0(n_0 + xL)$$

$$H(n_0 + xL + 1) \leqq \sum_{\ell=1}^{L} D_{x1}(\ell)\, H(n_0 + xL + 1 - \ell)$$

$$\leqq \sum_{\ell=2}^{L} D_{x1}(\ell)\, H(n_0 + xL + 1 - \ell) + D_{x1}(1)\, H_0(n_0 + xL)$$

$$= H_0(n_0 + xL + 1)$$

$H_0(n_0 + xL)$ is used simply to avoid having to write out the summation,

$$\sum_{\ell=1}^{L} D_{x0}(\ell)\, H(n_0 + xL - \ell),$$

while both $H_0(n_0 + xL)$ and $H_0(n_0 + xL + 1)$ are used similarly in the inequality in which $H(n_0 + xL + 2)$ appears on the left. Thus, for example, $H_0(n_0 + xL)$ may be replaced to obtain

$$H(n_0 + xL + 1) \leqq \sum_{\ell=2}^{L} D_{x1}(\ell)\, H(n_0 + xL + 1 - \ell)$$

$$+ D_{x1}(1) \sum_{\ell=1}^{L} D_{x0}(\ell)\, H(n_0 + xL - \ell).$$

In this form it is clear that $H(n_0 + xL + 1)$ is in terms of $H(n_0 + xL - L), \ldots, H(n_0 + xL - 1)$. $H_0(n_0 + xL + n)$ for $n = 0, 1, \ldots, L - 1$ are defined in a fashion analogous to that above so that the general inequality is

$$H(n_0 + xL + n) \leqq \sum_{\ell=1}^{L} D_{xn}(\ell)\, H(n_0 + xL + n - \ell)$$

$$\leqq \sum_{\ell=n+1}^{L} D_{xn}(\ell)\, H(n_0 + xL + n - \ell) + \sum_{\ell=1}^{n} D_{xn}(\ell)\, H_0(n_0 + xL + n - \ell)$$

$$= H_0(n_0 + xL + n);\ n = 0, 1, \ldots, L-1. \qquad (2.90)$$

The L inequalities shown here are determined by a policy sequence,

A_x, consisting of policies $\alpha_{x0}, \ldots, \alpha_{x,\,L-1}$. A $L \cdot N \times L \cdot N$ stochastic matrix, U^{A_x}, is defined by the coefficients of the H-vectors and so an $L \cdot N$ dimensional vector inequality may be given as

$$F(x) \leqq U^{A_x} F(x-1) \; ; \; x = 1, 2, \ldots \tag{2.91}$$

by definition of $F(x)$ in terms of $H(n_0 + xL + n)$; $n = 0, 1, \ldots, L-1$. Next another $L \cdot N \times L \cdot N$ matrix, $U(B)$, is defined:

$$U(B) = U^{A_B} \ldots U^{A_1} \tag{2.92}$$

Thus
$$F(B) \leqq U(B)\, F(0) \tag{2.93}$$

and by the hypothesis that a state b in the $L \cdot N$ state process may be occupied with probability $\geqq \sigma > 0$ after B transitions, we have

$$f_i(B) \leqq \sum_{j=1}^{L \cdot N} u_{ij}(B)\, f_j(0)$$

$$= (\sigma + \epsilon_i)\, f_b(0) + \sum_{j \neq b} u_{ij}(B)\, f_j(0)$$

where $u_{ij}(B)$ is an element of the matrix $U(B)$ and where $\sigma + \epsilon_i \geqq \sigma > 0$ is the probability of transition from i to b in B transitions using $U^{A_1}, \ldots, U^{A_B}$ as transition probability matrices

for these transitions. It follows that

$$f_i(B) \leqq \sigma f_b(0) + (1-\sigma) \max_j f_j(0); \quad i = 1, 2, \ldots, LN;$$

and so

$$\max_i f_i(B) \leqq \sigma f_b(0) + (1-\sigma) \max_j f_j(0) \tag{2.94}$$

By going back to (2.88) and replacing max by min, the inequality is reversed. Arguments identical to those used above in which all inequalities are reversed and max is replaced by min may be used to obtain

$$f_i(B) \geqq \sigma f_b(0) + (1-\sigma) \min_j f_j(0); \quad 1 \leqq i \leqq LN;$$

and so

$$\min_i f_i(B) \geqq \sigma f_b(0) + (1-\sigma) \min_j f_j(0). \tag{2.95}$$

Inequalities (2.94) and (2.95) may be subtracted to yield

$$\max_i f_i(B) - \min_i f_i(B) \leqq (1-\sigma)\left[\max_j f_j(0) - \min_j f_j(0)\right] \tag{2.96}$$

It is clear that the argument used to obtain this inequality in which $x = 1, 2, \ldots, B$ was used, holds also for $x = y\,B + 1, \ldots, y\,B + B$ for any integer $y \geq 0$ and therefore

$$\max_i f_i(y\,B+B) - \min_i f_i(y\,B+B)$$

$$\leq (1 - \sigma)\left[\max_j f_j(y\,B) - \min_j f_j(y\,B)\right]$$

so that

$$\max_i f_i(y\,B+B) - \min_i f_i(y\,B+B)$$

$$\leq (1 - \sigma)^{y+1}\left[\max_j f_j(0) - \min_j f_j(0)\right].$$

n_0 was any number such that $n_0 \geq L$ and so this is sufficient to prove that

$$\max_{\substack{1\leq i\leq N \\ 1\leq x\leq L}} [h_i(n+x)] - \min_{\substack{1\leq i\leq N \\ 1\leq x\leq L}} [h_i(n+x)] \to 0 \text{ as } n \to \infty \qquad (2.86) \quad ||$$

Bounds on the gain rate of the current L-stage policy sequence and on the gain rate of the optimal policy have been obtained. It has been shown that these bounds converge. The next theorem shows that the recurrence relation converges to a stationary optimal policy.

Theorem 2.14

If bounds on the gain rate converge, then the recurrence relation for the time-optimal Markov-renewal decision process converges to a stationary optimal policy provided ties for choice of alternative are broken by using the same algorithm at each iteration.

Proof

It has been shown (Theorems 2.12 and 2.13) that under certain conditions

$$\lim_{n\to\infty} h_i(n) = h^* \quad \forall\, i. \tag{2.97}$$

The recurrence relation is

$$w_i(n) = \max_k [q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\ w_j(n-\ell)\ ]$$

It is clear that for Equation (2.97) to hold

$$w_i(n) \to h^*\ Ln + w_i \quad \forall\, i \text{ as } n \to \infty \text{ for some constant } w_i$$

Therefore,

$$w_i(n) \to \max_k [q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\ (h^*\ Ln - h^*\ L\ell + w_j)\ ] \text{ as } n \to \infty$$

The summation over terms not dependent on j or ℓ may be given explicitly so that

$$w_i(n) \to h^*\ Ln + \max_k [q_i^k + \sum_{j=1}^{N} \sum_{\ell=1}^{L} d_{ij}^k(\ell)\ (w_j - h^*\ L\ell)\ ] \text{ as } n \to \infty. \tag{2.98}$$

The quantity to be maximized is independent of n so the optimal policy will be stationary if the same algorithm is used to choose k on each iteration. ||

In summary, the algorithm for the time-optimal problem consists of iterating using the recurrence equation (2.52). At every stage n after the first 2L stages we compute

$$\max_{\substack{1\leqq i\leqq N \\ n-L<x\leqq n}} \left[\frac{w_i(x) - w_i(x-L)}{L} \right]$$

$$\min_{\substack{1\leqq i\leqq N \\ n-L<x\leqq n}} \left[\frac{w_i(x) - w_i(x-L)}{L} \right]$$

These are bounds on the gain rate of the current and optimal L-stage policy sequences. The bounds converge and also convergence occurs to a stationary policy providing certain conditions on the form of the MRDP hold.

2.4 Computational Considerations

In this section we look briefly at the computation and storage required to obtain solutions to MRDP's. In typical MRDP's which are models of physical systems (including those in Chapter 3), transitions may be made from the current state to only a few (say m) of the N states so that $m<<N$. With this situation much less computer storage is needed if only non-zero transition probabilities are stored rather than all probabilities. For example,

solution of the transition optimal problem requires only storage of the non-zero p_{ij}^k and of three N-dimensional vectors composed of the $v_i(n)$, $v_i(n-1)$, and q_i^k. Each iteration requires one multiplication for each non-zero p_{ij}^k.

It is interesting to compare the computation required when using Equation (2.3) and the computation when using the algorithm devised by Howard [23] called the policy iteration method. Figure 2.4 shows this algorithm. Howard proves that it does converge to an optimal policy and that each successive policy chosen has a gain at least as great as that of the preceding policy. The computation required in the Policy-Improvement Routine is comparable to that required for an iteration of Equation (2.3). The Value Determination Operation, however, requires the solution of N simultaneous linear equations in N unknowns. Using Gauss elimination to solve these requires about $N^3/3$ multiplications ([53], p. 241). Except in special cases the storage required would be that for a full $N \times N$ matrix and so no advantage is gained by the fact that the matrix is likely to be sparse. No comparison of the number of iterations required for satisfactory convergence using the two methods has been made. This can vary over a wide range depending on the parameters chosen. From a theoretical point of view, the advantage of the Howard algorithm is that after some finite number of iterations one can state with certainty that the optimal policy has been found while

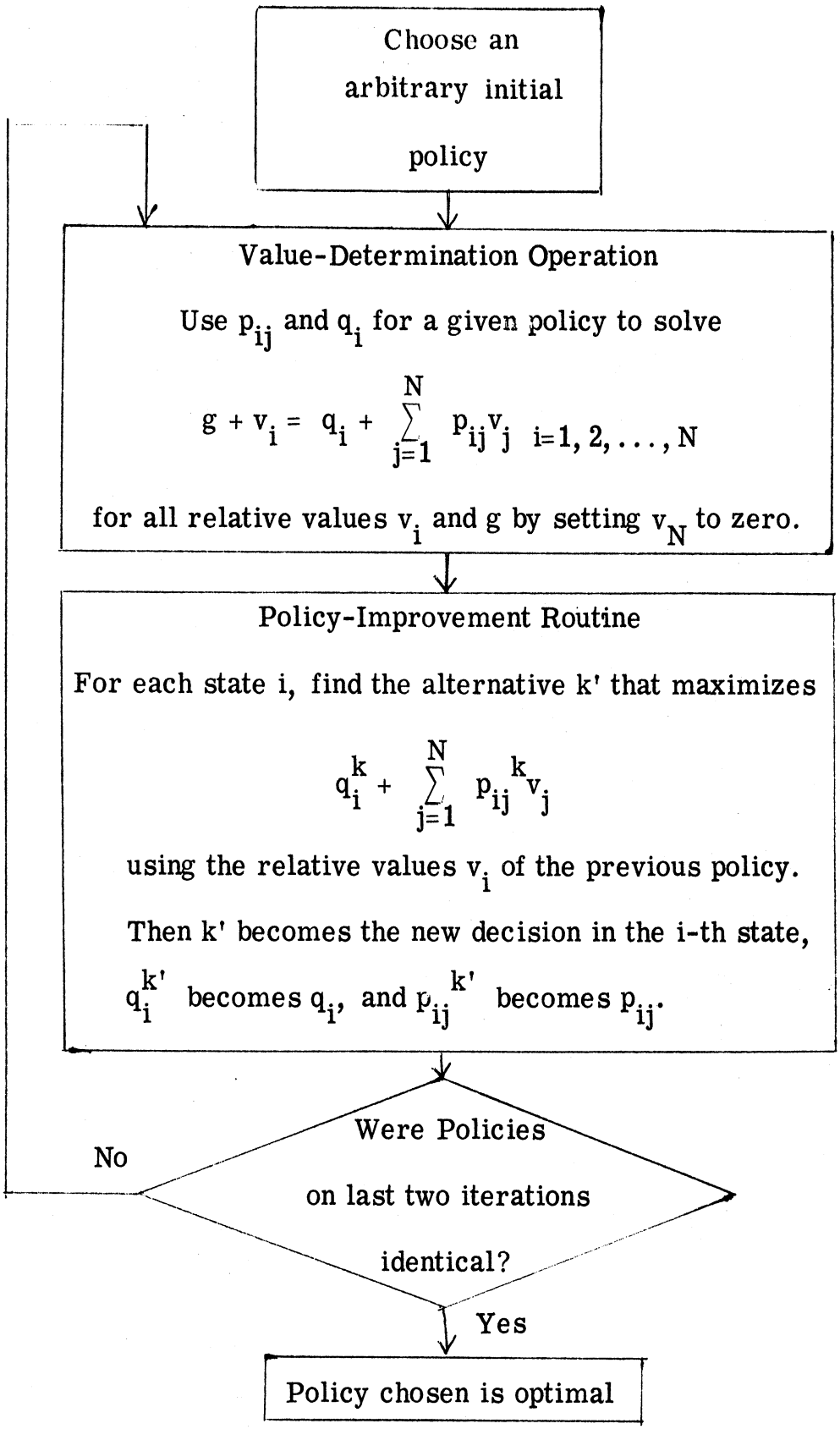

Figure 2.4. The Howard Algorithm

use of the recurrence relation only tells how close the gain of the current policy is to that of the optimal policy but does not guarantee that the current policy is the optimal policy. From a practical point of view, however, the recurrence relation has the advantage that the iterations may be stopped at any point and in particular when the bounds on the gain have been computed with as much precision as that for other parameters of the problem.

At this point we turn to a complete mathematical description of a model which provides an application for the optimization method presented in this chapter.

Chapter 3

MODELS FOR MULTIPROGRAMMING

This chapter contains a detailed description of mathematical models for multiprogrammed computer systems. The models are useful for analysis and optimization of task scheduling policies where an optimum task scheduling policy is defined to be one which maximizes the system throughput.

Many of the computing systems being marketed today have features which facilitate a multiprogrammed mode of operation. This is desirable because effective use of multiprogramming allows more efficient use of the system resources than would otherwise be possible. Multiprogramming makes it possible to increase the productivity of valuable system resources such as CPU(s) and I/O devices (drums, disks, printers, etc.) by increasing the fraction of time tasks are available for these resources to work on. The effectiveness of multiprogramming to a given system's efficiency of operation depends on the task scheduling rules and procedures used in that system.

The model described in this chapter measures the efficiency of a multiprogrammed computer system (in terms of task completions possible per unit time interval or "throughput") as a function of the task scheduling rules used and other parameters. Two salient features of this model are that the model may be used to measure

system throughput when reentrant code or other sharable information is used by tasks; and the model is suitable for application of an optimization procedure so that maximum throughput scheduling rules may be found.

The scheduling model may be described in a rough way with the aid of Figure 3.1. Tasks in main memory alternate among periods of waiting for a CPU, using a CPU, and experiencing a delay for an I/O operation. The CPU scheduler determines which of these tasks waiting for a CPU receive the use of one. New tasks arriving at the system are placed in a queue outside the main memory, and the memory scheduler decides which tasks to move into main memory. It is clear that decisions made by these schedulers are interdependent and so they cannot be treated separately. For example, the task configuration in main memory is affected by both the memory scheduler and by priorities assigned to tasks in main memory by the CPU scheduler.

The memory scheduler is constrained by the sizes of tasks and the number of pages of main memory available where the word pages here may refer to any convenient sized unit of memory if no page unit is defined for the system being modeled. And of course, the CPU scheduler is constrained by the number of CPU's available in the system.

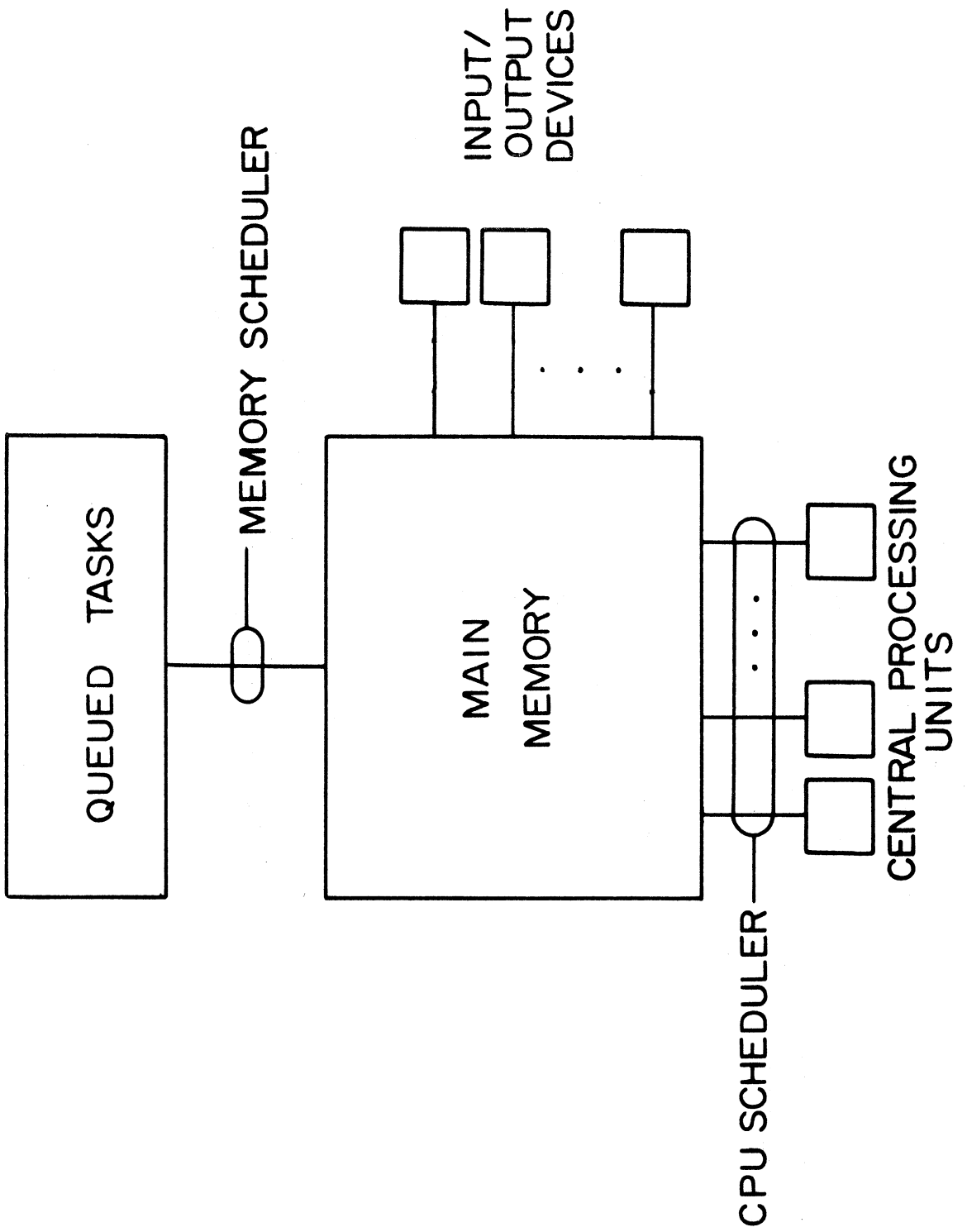

Figure 3.1 System Block Diagram

Figure 3.2 presents the model as a service system and shows more clearly the task flow through the system. New tasks arriving at the system are placed in the memory queue and the memory scheduler controls the movement of tasks to the CPU queue. The CPU scheduler schedules tasks in the CPU queue on the available CPU's. The CPU's are periodically interrupted and at these points may be assigned to other tasks even though tasks currently using them have not finished. An executing task may request an I/O operation (including a page request) and at this point it becomes ineligible to use a CPU until the I/O operation is complete. After completion of the I/O operation the task may either terminate and leave the system,or it may require another CPU interval in which case it reenters the CPU queue. Reference will be made to this figure in the following sections where specific parameters and assumptions are discussed. We now turn to the specific assumptions made in the model.

3.1 Assumptions

In this section, the general assumptions made in the development of the computer system model are given, and some of the parameters of the model are defined. Other specific assumptions are given in relevant sections. Listed with the assumptions are some of the input parameters required for the model and comments on the validity of the assumptions. The assumptions are numbered to make them easier to pick out from the text.

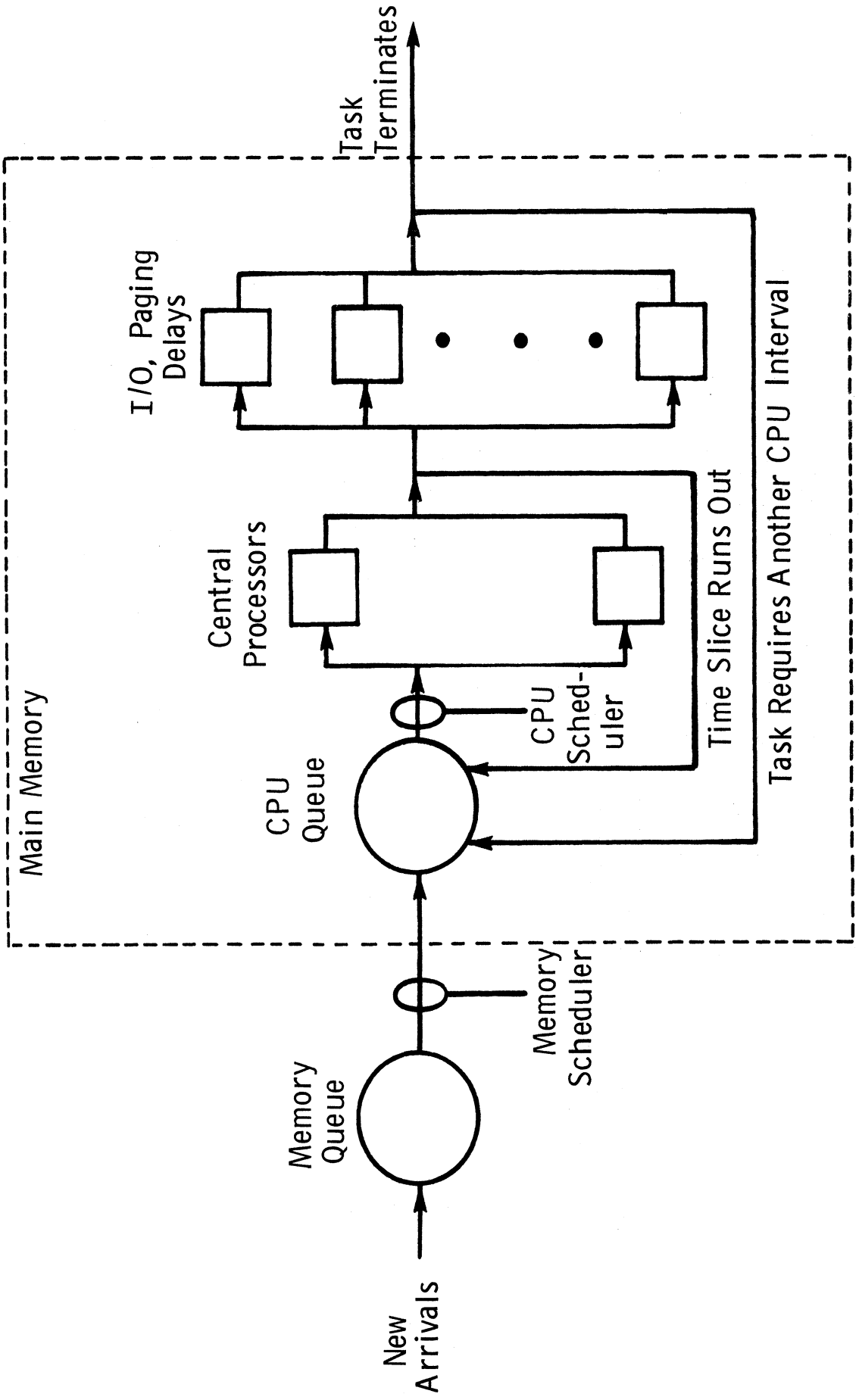

Figure 3.2 Task Flow In System

1) There are M distinct kinds of tasks; and thus at each point in the system shown in Figure 3.2, M task types must be distinguished.

The reason for this distinction is to allow specification of sets of tasks which may share information and specification of sets of tasks with similar execution characteristics. Thus, tasks of a given type may be allowed to share information: for example, the procedure of tasks of the same type may be sharable among these tasks. Furthermore, tasks of a given type are not distinguished from one another in a queue and in the sense that their execution characteristics are all drawn from the same probability distributions.

2) The probability that a new arrival at the memory queue in Figure 3.2 is a task of type i; $i=1,2,\ldots,M$; is independent of the status of the system at the time of arrival.

The probability that a newly arriving task is of type i is an input parameter to the model and is denoted by b_i, and of course,

$$\sum_{i=1}^{M} b_i = 1.$$

The assumption that the types of newly arriving tasks are independent of the status of the system and in particular of the number of tasks of each type already in the system is justified if users are independent of one another.

3) When a task completes and leaves the system, a new task

immediately enters the memory queue.

As stated above, the new entry in the memory queue is not necessarily a task of the same type as the task which terminated. The assumption implies that there are always a fixed number of tasks in the model. This number is an input parameter and is denoted by N. This restricts attention to a system under heavy load, and this is the condition where choice of scheduling rule is especially important to performance of the system. With this assumption we assure that there are always tasks awaiting loading and thus assure a heavily loaded (perhaps overloaded would be a better term) computer system. A situation similar to this actually occurs in a system with a limited number of users (perhaps limited by available terminals), each making a new request shortly after receiving a response to the previous request.

4) The time required to move a task from the memory queue to the CPU queue in Figure 3.2 is a variable which is dependent only on the type of task being loaded and on whether or not other tasks of the same type are in main memory.

Thus, given the type of task to be loaded and knowing whether other tasks of the same type are in main memory, the time required to load the task is a known constant and is another of the parameters required to specify the model. During the course of this study data were collected which included the loading times for many tasks. When these tasks are sorted by type, it is found that loading times for each type do not vary greatly from one load to another. Some of these data are presented in Appendix

D and provide justification for the foregoing assumption.

5) The length of time a task uses a CPU before requesting an I/O operation is exponentially or hyperexponentially distributed with parameters of the distribution dependent on the type of task using the CPU.

Relating this to Figure 3.2, the boxes labelled Central Processors may be considered exponential or hyperexponential servers with the parameters of the service distribution dependent on the type of task in service. These parameters must, of course, be specified and are among the input parameters of the model. The small amount of data previously available [41] on lengths of CPU intervals shows a good fit to the exponential. These data, however, lump all tasks in the computer system together. Data for individual tasks presented in Appendix D fit hyperexponential distributions very well.

6) Input/output delays are exponentially or hyperexponentially distributed with parameters of the distribution dependent on the type of task.

This may be related to the boxes labelled I/O, Paging Delays in Figure 3.2. Each such box holds a task entering it an exponentially or hyperexponentially distributed amount of time, and the parameters of the delay distribution depend on the task type. An I/O delay is the time from request for an I/O operation until that operation is complete and the task either terminates or goes back to the CPU queue. This means that we do not model queuing explicitly at the I/O devices, and that there are

enough I/O delay boxes in Figure 3.2 to handle the maximum number of tasks which may require them at any one time. The I/O delay is dependent on the type of task and on the number of other tasks competing for the available system I/O devices. That is, the effects of queuing on I/O delays are handled indirectly by including both queuing time at I/O devices and the time to perform the I/O operation in the I/O delay. In view of the multiplicity of such devices (e.g. drums, disks, printers, etc.) and the multiplicity of queues at some individual I/O devices (e.g. drum), this is felt to be a reasonable approximation. The data in Appendix D fit hyperexponential distributions quite well and data in [41] fit exponential distributions fairly well.

7) At completion of an I/O operation, a task may terminate and leave the system or may return to the CPU queue for another CPU interval.

The probability of termination is independent of the number of execution intervals the task has already received. This probability is, however, dependent on the type of task and is a parameter of the model which must be specified. The probability that a task of type i terminates after an I/O delay is denoted by p_i ($1 \leq i \leq M$). The assumption that the termination probability is independent of the number of execution intervals already taken means that the total number of times the task goes through the execution-input/output cycle is a geometrically distributed random variable. This distribution was hypothesized first on the basis of data in [41] which show it to be possible. Data were then obtained (presented

in Chapter 4) which show this assumption to be valid.

8) All tasks of a given type require the same amount of main memory for execution.

The amount of sharable and non-sharable memory required by each type of task must be specified. If the memory requirement for a task is broken into a part that can be shared with other tasks of the same type and a non-sharable part, then we assume each task of the same type requires the same amount of non-sharable memory. (The sharable memory requirements are also identical.) This assumption is made so it is always possible to determine whether there is room in the core to load a particular type of task without having to distinguish the memory requirements for each task of the type. The data in Appendix D indicate that this is not an unreasonable assumption. Furthermore, as shown in Appendix C, the variation in memory requirement from one task to another becomes less important in this regard as the number of programs being multiprogrammed increases.

In summary, in the queuing model which we are developing in this chapter we distinguish tasks by type which is another way of saying they are distinguished, for example, by the procedure or code that they use. We would, for example, distinguish an assembler from a PL/1 compiler. The tasks of one particular type, however, are indistinguishable in the sense that no special information (such as their elapsed CPU time) is maintained about each one. The only information available which allows tasks of a given type to be distinguished is the tasks' status in the system. By this is meant an indication of the task's activity:

whether awaiting loading, queued at or using a CPU, or carrying out an input or output operation. Loading times, lengths of CPU intervals between I/O operations, times required to satisfy I/O requests, and the number of I/O operations before terminations have distributions which are the same for all tasks of a single type.

We now turn to a detailed description of the mathematical model.

3.2 Mathematical Models

In Chapters 1 and 2 we discussed the use of Markov models for modeling stochastic service systems. Discussed also were Markov renewal decision processes and techniques for finding optimal policies in a process of this type. We now turn to description of a specific Markov model for a multiprogrammed computer system. This model is suitable for analysis of the computer system throughput resulting from various task scheduling policies and is also suitable for application of the previously discussed optimization techniques in order to find scheduling rules which maximize throughput.

To increase the generality of the results and to allow more comparisons of results, we will describe two closely related models: one to be used for obtaining optimal scheduling policies and the other to model a scheduling policy similar to that in use on a typical large time-shared multiprogrammed computer system. This last will be called the heuristic scheduling policy. In what follows much of the material will be common to both systems, but where a distinction must be made reference will be made to the appropriate system by referring to either the optimization model or to the single queue model. The term

"single queue" arises in the following way. In the optimization model, it is convenient to think of a separate memory queue and a separate CPU queue for each of the M types of tasks in the system. Then at a decision point, the optimum scheduler decides on the best task type to load and/or assign to a CPU and goes to the appropriate memory queue or CPU queue to get a task of that type. On the other hand, the heuristic scheduling policy makes no attempt to distinguish the M types of tasks, and so it is natural to think of all types of tasks occupying a single memory queue and a single CPU queue.

The emphasis in the optimization of the task scheduling rules is on maximization of task completions per unit time in a heavily loaded (or overloaded) system. No consideration is given to response times of individual tasks. Scheduling policies are not influenced by such factors as elapsed waiting times in queues or the length of time a particular task has held the CPU. Thus there is no time slice in the optimization model. On the other hand, the scheduling rules in the single queue model are an approximation to first-come-first-serve (FCFS) for the loading of tasks from the memory queue and last-come-first-serve (LCFS) for tasks arriving at the CPU. In the single queue model, one of the input parameters is the length of the CPU time slice. Results from the optimization and single queue models may be used to compare throughput for the two policies. It should be noted that since optimal policies maximize throughput, the average response to tasks is minimized for these policies. The variance of this response may be

large, but this price may be worth paying in order to maximize throughput during periods of system overload.

It was originally proposed to model demand paging in a system of this type in some detail, but this was dropped after data were obtained which showed that a suitable characterization of the system may be obtained without this detail.

In the remaining sections of this chapter, the goal is to define the model in detail so it is useful for determination of scheduling rules which maximize throughput and for determination of the throughput realized by both optimal scheduling rules and the heuristic scheduling rule which is to be considered. In order that the optimization method described in Chapter 2 be applicable it is necessary to define the states of the model and possible alternatives in each state so that transitions among states form a Markov chain. For each state and alternative the transition probabilities to all other states must be determined. It is also necessary to define a one transition return function such that the expectation of the return per unit time for a given scheduling policy corresponds to the expected system throughput. Then the optimization method may be used to determine a scheduling policy which maximizes this expectation.

3.3 Regeneration and Decision Points

The models of multiprogrammed computer systems we are discussing in this chapter are Markov renewal decision processes (optimiz-

ation) or simply Markov renewal processes if the decision variables are fixed (single queue). In a Markov renewal process the successive states of the model form a Markov chain while the time interval between successive state transitions is described by a distribution function which may be dependent on both the current and the next states. In a Markov renewal decision process this distribution function may depend on the decision variables as well as the state variables. In a Markov renewal process the time instants at which a transition is made are called regeneration points. It is at the instants of transition in a Markov renewal decision process that decision variables are chosen, and so we will call these time points the decision points of the Markov renewal decision process.

Usually at a decision point a decision is made by the memory scheduler to move a task into main memory. The next decision point occurs at the completion of this transfer. The scheduler may also choose to take no action, and in this case the next decision point occurs after a time interval equal to the shortest possible loading time interval.

For the single queue model we approximate the case where all tasks go into a single queue when they arrive at main memory and are loaded on a FCFS basis as space becomes available in main memory. This is discussed in detail in section 3.7. When no load is possible because of inadequate space in core we assume a regeneration interval equal to that of the shortest possible loading time.

The task loading decisions made in the optimization model determine the time interval until the next decision point. Other events and decisions also occur at decision points. The CPU scheduler interrupts and reassigns (possibly to the same tasks) all CPU's. A CPU scheduling policy is chosen at the decision point and is used until the next one. This policy is a rule for assignment of tasks to CPU's when one becomes available. The assignment choice may depend on the status of other tasks in the main memory and in the memory queue but cannot depend on the time remaining till the next regeneration point. That is, a stationary, state-dependent CPU scheduling policy is used between decision points, and at each decision point a new policy may be chosen. A CPU becomes available for assignment by the CPU scheduler when a task using it requests an I/O operation (including a page demand) or is interrupted for some other reason. The CPU scheduler may also preempt a task using a CPU in order to re-assign the CPU to a task of higher priority which has just completed an I/O operation. An example of a CPU scheduling policy is a simple relative priority assigned to each type of task. Then whenever a CPU becomes available, that task in the CPU queue with the highest priority is executed.

For the single queue model we approximate a situation in which tasks are assigned CPU's on a LCFS basis. This is discussed in section 3.6 although it may be stated here that the approximation is exact when a task arrives in the CPU queue after completion of an I/O operation. Again there is only a single queue for the CPU's. When a task completes

an I/O operation and requires more CPU time it is assigned a CPU immediately. If no CPU's are idle at the time of the request, the task preempts a user of one of the CPU's. All tasks using a CPU are equally likely to be preempted in this fashion.

In the models the time between regeneration or decision points is known exactly. At one of these points we may choose either to load a task (if there is room in memory) or not load a task. We set the time between regeneration points when there is no task being loaded to unity and set times required to load tasks to integer multiples of this. Other times in the model are scaled accordingly.

The time required to load a task is deterministic and depends on the type of task and whether or not there are other tasks of the same type in execution store at the time of the load. If some information is shared among tasks of the same type, then the loading time for tasks of a type already in memory may be reduced by the fact that some of the information required by the task is already loaded. The loading times (integer) and space requirements (total memory size = P pages) for a task of type k (k = 1, ..., M) are

c_k : total number of pages of sharable information

d_k: number of pages of non-sharable information

I_{ck} : loading time when no other task of same type is in memory

I_{dk} : loading time when another task of the same type is in memory.

At a decision point we make a decision, denoted by D, which is a

specification among other things of the type of task to be loaded, if any. The current state is denoted by i_1 and specifies what tasks are in memory. The decision, D, and current state, i_1, determine the task to be loaded and the time required to load the task which is the time between decision points. For our model, then, we have a specialization of the most general MRDP because the distribution function for the time between transitions depends only on the initial state, i_1, and not on the final state, i_2. Furthermore it is an especially simple lattice distribution because it is a delayed unit step function. That is, if $I_{i_1}^D$ is the time between transitions

$$F_{i_1 i_2}^D(\tau) = F_{i_1}^D(\tau) = U(\tau - I_{i_1}^D)$$

where

$$U(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

When D specifies no load, $I_{i_1}^D = 1$. When D specifies that a type k task is to be loaded and i_1 indicates there are other tasks of type k in memory then $I_{i_1}^D = I_{dk}$. Complete specification of the decision vector, D, will be given later.

For the single queue model the same loading times are required for tasks, but tasks are loaded into memory on a FCFS basis as space in memory becomes available.

We now give a description and definition for an arbitrary state of the model by means of a set of state variables.

3.4 State Variables

The state of the Markov chain defined at successive decision points is given by specifying the status of the system at these time instants. This status information includes, for each type of task, the number in the memory queue, in the CPU queue and in input/output. The state may thus be specified by the three - tuple,

$$(\alpha, \beta, \gamma)$$

where each of these variables represents a vector:

$$\alpha = (f_1, \ldots, f_M)$$
$$\beta = (g_1, \ldots, g_M)$$
$$\gamma = (h_1, \ldots, h_M)$$

f_k = number of type k tasks queued outside main memory

g_k = number of type k tasks queued at or using a CPU

h_k = number of type k tasks in I/O (inculuding paging)

We also define

$z_k = g_k + h_k$: the number of type k tasks in main memory.

For convenience we define a function $T(\alpha, \beta, \gamma)$ which maps each state into a unique integer state designator i. For the particular state $(\alpha^1, \beta^1, \gamma^1)$ we write

$$i_1 = T(\alpha^1, \beta^1, \gamma^1)$$

An assumption was made that whenever a task terminates and leaves the system a new task immediately arrives at the system. This

leads to the following condition on the state variables:

$$\sum_{k=1}^{M} (f_k + g_k + h_k) = N$$

where N is the constant number of tasks queued at or in memory.

It should also be noted that in specifying the state, no indication is given of the status of tasks individually: but rather the status of each type of task is given. Thus the aggregate number of memory pages allotted to data for all type k tasks is assumed to be divided equally among these tasks as mentioned earlier. This brings us to another restriction on the state variables; namely, that the total number of pages used by the tasks in main memory must be less than or equal to the total pages in memory available for these tasks, P. This restriction is given explicitly following a discussion of the memory space required by tasks.

If we are modeling a system where an entire task is moved into main memory before execution begins, then we may use c_k as the number of pages of sharable information required for type k tasks and this number is independent of the number of tasks of type k in core. On the other hand we may be modeling a system operating in a page-on-demand mode. In this situation we may assume that a single type k task has c_{1k} pages of sharable information in core. With two type k tasks in main memory we have a total of c_{2k} pages of sharable information in core because the second task may be using some pages which are different than those in use by the first task.

With n type k tasks in core we have $c_{n\,k}$ and of course $c_{nk} \leq c_k$. If tasks are independent of one another and each page of the total c_k pages is equally likely to be in use by each task, we may compute $c_{n\,k}$ based on a knowledge of c_{1k}. The table derived in Appendix A may be used for this purpose. c_{1k} may be estimated from system data and so this is a useful approximation.

The parameter d_k is defined as the number of non-sharable pages a type k task has in core. This may be the task's entire complement of non-sharable pages or perhaps some fraction of them.

We may now write out explicitly the restriction on the state variables which states that the tasks in memory must not require more than the P available memory pages.

$$\sum_{k=1}^{M} (c_{z_k k} + z_k d_k) \leq P$$

where $c_{ok} = 0$.

In systems where all sharable pages are loaded into memory we have

$$c_{z_k k} = c_k \qquad z_k \geq 1$$

$$= 0 \qquad z_k = 0$$

3. 5 Decision Variables

At each regeneration point in the model a memory scheduling and CPU scheduling alternative must be chosen. These alternatives together form the decision made at the regeneration point. In general, decisions at successive regeneration points may be made randomly,

may depend on values of some or all of the state variables, on the time remaining till the system shuts down, or on some combination of these factors. We will always assume that the time until the system shuts down is long enough so that no error will arise by considering it to be infinite. This is perfectly reasonable when one considers that the time between transitions may be measured in milli-seconds while the total continuous running time of the system between shutdowns may normally be measured in hours.

If we specify a decision for each state of the model where this decision is dependent only on the system state, then this decision specifies a stationary policy. An optimal decision vector has been found if no other decision vector exists which results in greater computer system throughput. It can be shown [16] that optimal policies for the model being developed here exist which are both stationary and deterministic. For such a policy, the scheduling decision is completely determined when the state of the system is known. Since policies of this type are also easiest to implement, we will not consider random or time-varying scheduling policies.

Decisions which may be made at a regeneration point may be specified by a vector, (a_1, a_4), in the optimization model. This will be made more clear after we define the components of this vector.

a_1: $a_1 = 0$ means no task is to be loaded in the next interval.

$a_1 = k$ means load a type k task in the next interval.

a_4: $a_4 = (k_1, \ldots, k_M)$ specifies the relative priority at the CPU's for each task type. The variable k_i specifies the type of task which has priority i at the CPU; thus k_1 specifies the task type with highest priority and k_M the task type with lowest priority.

For convenience we define a function D which maps each decision vector into a unique linear decision designator. When we desire to indicate that a variable depends on the decision made, we superscript that variable with a D rather than the entire decision vector. Let us now define the function D. To do this we need to know the values which the decision variables may assume and the range of these values. The decision variables may assume only integer values. a_1 may assume values $0, \ldots, M$ corresponding to the possible decisions to load any type of task and to the decision to load no task. Thus a_1 may take on M+1 values.

The decision variable a_4 was defined as a vector $(k_1, \ldots, k_M)$ where the values of the k_i range from 1 to M and give the priority of each task type at the CPU. There are actually M! different CPU scheduling policies possible which means that the interval (1, M!) is sufficient for all CPU scheduling policies. For convenience, we may map each CPU scheduling vector into a unique integer by forming the integer $\tilde{a}_4 = k_1\ k_2 \ldots k_M$ from the vector $(k_1, \ldots, k_M)$. For $M \leq 10$ this may be a radix 10 number. For $M > 10$ choose the radix M. In any event, the range of values which this integer may take on is now

(12...M, M...1) and this is greater than M!. Only M!, however, of the integers in this range are legal scheduling policies for the central processor.

We may now give D as a function of a_1 and a_4.

$$D = (M + 1)\,\tilde{a}_4 + a_1$$

Many of the values of D which would be generated by allowing a_1 and $\tilde{a}_4$ to take on all values in their interval of definition do not correspond to any physical alternatives in the model. Many other alternatives (values of D) may be chosen only in subsets of the total states of the model. For example, a decision to load a task may not be made unless the state of the model is such that there is room in main memory for the task. Neither of these facts presents any additional complexity from a computational point of view.

In a given state, the components of the decision function (the decision vectors) are constrained as follows.

1) A decision to load a task of type i ($a_1 = i$) may be made only if the memory space required by the task does not exceed the available memory space (space not in use by other tasks).
2) A decision to load a type i task may not be made unless there is a type i task in the memory queue.
3) The CPU priority vector, a_4, is restricted so that priorities of task types which are not in the main memory all have lower priorities than any priority of a task type which is in main

memory.

4) The CPU priorities for task types not in main memory are assigned so that these task types appear in ascending order in the CPU priority vector.

The first two restrictions assure that only legal decisions are made in the states of the model. The last two restrictions assure that there is a one-one mapping between possible values of D and physical scheduling choices at the CPU. This is done by assigning CPU priorities to tasks which are unable to make use of the CPU in a unique way.

3.6 Submodels

One of the main goals in this chapter is to determine the probabilities of transition between all pairs of states under all possible alternatives (D) in the Markov model of the computer system. In section 3.3 it was explained that these transitions occur at particular instants of time which are called regeneration points or decision points. Before giving a derivation of these transition probabilities in the model, it is necessary to discuss the events which occur between two such regeneration or decision points and to describe these events mathematically. Recall that this time interval is dependent on the time required to load a task into main memory; the time is unity if no task is being loaded. In terms of Figure 3.2, concern is with the events which occur inside the dashed line (including the possibility of task terminations) during the time between two regeneration or decision points. In this interval tasks may queue at the CPU, or they may use the CPU for a

period of time and then either request an I/O operation or return to the CPU queue because of preemption by another task or time-slice runout. Tasks which request an I/O operation or which are in I/O at the beginning of the interval may stay in I/O, terminate, or request another CPU interval.

In section 3.1 we stated assumptions about the lengths of CPU and I/O intervals and about the number of CPU-I/O cycles before termination. These assumptions are that CPU and I/O intervals are exponentially or hyperexponentially distributed and that the number of CPU - I/O cycles is a geometrically distributed random variable. These assumptions allow events between regeneration or decision points to be described by a continuous time, finite state Markov process. By modeling the events which occur between regeneration points by means of such processes we obtain results which are needed in the determination of transition probabilities for the model. The parameters of these processes will depend on the state at the regeneration or decision point and on the CPU priority vector, a_4. We will refer to these Markov processes as submodels to distinguish them from the MRP or MRDP of the single queue model and optimization model, respectively. Thus for each decision in each state of the model, a submodel is defined and this submodel is used in the determination of the transition probabilities from the given state under the given decision to all other states of the model.

It may be recalled from Chapter 1 that a continuous time Markov process can be defined by means of an initial vector of state probabilities and by a transition intensity matrix which will be denoted by A . It is the purpose of the remainder of this section to define suitable states for the submodels and to determine transition intensities between all pairs of states in these submodels . An initial vector of state probabilities for each submodel is defined in terms of the state of the model at the regeneration or decision point. It may appear that the number of such submodels and therefore the computation required for them would be excessive since one or more submodels is defined for each state of the model. It is shown, however, that by suitable choice of computational method, solution can essentially be carried out for large numbers of submodels simultaneously. Then the vector of state probabilities for the submodel at the end of the regeneration interval is used as part of the function which specifies the transition probabilities from the initial state to every other state of the model. Most of these probabilities are seen to be zero.

A submodel of the optimization model has states which may be described as a pair of vectors (ζ, η) where

$$\zeta = (u_1, \ldots, u_M)$$

$$\eta = (v_1, \ldots, v_M)$$

u_k = number of type k tasks queued at or using a CPU

v_k = number of type k tasks in I/O.

In the optimization model, the actual assignment of CPU's to tasks is determined by decision variable a_4 for each state. That is, when a CPU becomes free due to termination of an execution interval, the scheduling policy determines which of the available tasks will be allowed to use the CPU. Suppose the CPU scheduling policy is $a_4 = (k_1, \ldots, k_M)$ which is an ordering of task types from highest to lowest priority. Define s_ℓ as the number of type ℓ tasks using a CPU. Obviously $s_\ell \leq u_\ell$. Then if we have L CPU's (we will restrict L to one or two in the model), we get

$$s_{k_1} = \min\ (u_{k_1}, L)$$

$$s_{k_2} = \min\ (u_{k_2}, L - s_{k_1})$$

$$s_{k_M} = \min\ (u_{k_M}, L - \sum_{\ell=1}^{M-1} s_{k_\ell})\ .$$

For the single queue model the CPU assignment is not a deterministic

function of ζ and a_4. Therefore we must add another state vector to the submodels here,

$$\psi = (s_1, \ldots, s_M)$$

s_k = number of type k tasks using CPU's.

For each type of task we must specify the mean CPU interval, the mean I/O delay, the total mean CPU time required before completion, and the probalility that a task using the CPU is interrupted for other than an I/O delay. Define the following parameters:

$1/\mu_k$: mean CPU interval between I/O requests for type k tasks

$1/\nu_k$: mean time required to satisfy an I/O request (including page request) for type k tasks.

p_k : probability that a type k task terminates after completion of an I/O operation.

$1-p_k$: probability that a task requires another CPU interval after an I/O operation is complete.

$1/\rho$: the mean time slice for all types of tasks in the single queue model.

In most systems the time slice is a known fixed interval, but in the model the time slice has an exponential distribution. The use of a time slice may affect the results in the single queue model but has no effect in the optimization model. For an optimal scheduling policy we assign the CPU based on the number of tasks of each type in the memory queues and CPU queues. These numbers do not change when a time

slice runs out and therefore the optimal scheduler will re-assign the CPU to the same type of task.

We now have defined all parameters necessary for a complete specification of the submodels for both the optimization and single queue models. As stated earlier, these submodels are continuous time, finite state, Markov processes. It is necessary to obtain transient solutions to these models for the time between two regeneration points. Let us now take a closer look at these submodels. First expressions are obtained which may be used to get the transition intensities between all pairs of states in any submodel and thus to define a transition intensity matrix, A. Then the initial state probability vector for an arbitrary submodel is obtained as a function of the state of the model at the regeneration or decision point.

By means of vectors ζ, η, and ψ we have defined the states of a submodel. Transition from one state to another may be defined in terms of a transition intensity which is a function of the elements of these vectors and of μ_k, ν_k, and p_k; $k = 1, \ldots, M$. By determining all possible transitions and their intensities for all the states of a submodel, a transition intensity matrix, A, may be formed. Most entries in this matrix will be zero. Next the transitions which may occur in a submodel are given along with the associated transition intensities. It may be helpful to refer to Figure 3.2 in order to put these transitions in the context of the model.

1) A type i task which is currently using a central processor may request an I/O operation.

Given a state in which one or more type i tasks are using central processors, it is possible for one of these tasks to request an I/O operation at some point in time. The state transition here includes the movement of a type i task from a CPU to I/O. This frees a CPU which must now be assigned to one of the tasks in the CPU queue, if any. In the optimization model the assignment of a queued task to the available CPU is deterministic and depends only on the CPU priority vector, a_4, for the submodel. Therefore the change in state for this transition may be written

$$u_i \rightarrow u_i - 1$$

$$v_i \rightarrow v_i + 1$$

If s_i type i tasks are using central processors, then the transition intensity for the optimization model in this case is $s_i\mu_i$. Note, however, that for the optimization model, $\psi = (s_1, \ldots, s_M)$ is not actually part of the state specification but rather is determined from the vectors ζ and a_4. In the single queue model, on the other hand, there is no CPU priority vector and so ψ is needed to specify the state. Also the state variables do not contain a record of the order of the CPU queue but only the number of tasks of each type in the queue. Therefore the CPU's must be assigned probabilistically as they become available. The probability that a task of type j is at the head of the CPU queue is assumed to be proportional to the number of type j tasks in the queue

relative to the total number enqueued. Therefore the probability that the CPU released by the type i task is assigned to a type j task is

$$\frac{u_j - s_j}{\sum_{k=1}^{M} (u_k - s_k)}$$

The change in state for this transition is written

$$u_i \rightarrow u_i - 1$$

$$v_i \rightarrow v_i + 1$$

$$s_i \rightarrow s_i - 1$$

$$s_j \rightarrow s_j + 1$$

and the transition intensity is

$$\frac{s_i \mu_i (u_j - s_j)}{\sum_{k=1}^{M} (u_k - s_k)}$$

It is possible, of course, that no tasks are queued at the CPU when the type i task completes its CPU interval, and then the transition is

$$u_i \rightarrow u_i - 1$$

$$v_i \rightarrow v_i + 1$$

$$s_i \rightarrow s_i - 1$$

and the intensity in this case is $s_i \mu_i$.

2) A task of type i currently in I/O may require another CPU interval.

There may be several type i tasks in I/O, and one of these may complete its I/O operation and request another CPU interval at some instant of

time. At completion of an I/O operation, a task may terminate or request another CPU interval; and a CPU interval is requested with probability $1 - p_i$ for a type i task. In the optimization model, the change in state for the transition is

$$u_i \rightarrow u_i + 1$$

$$v_i \rightarrow v_i - 1$$

and the transition intensity is $v_i \nu_i (1-p_i)$. If the type i task going from I/O to the central processors is of higher priority than one or more of the tasks using CPU's at the time of the transition, then the type i task is assigned the CPU in use by the lowest priority task and this task goes into the CPU queue. For the single queue model, CPU's are assigned to tasks coming from I/O on a LCFS basis and so the type i task preempts a task using a CPU if there are no idle CPU's. Each such task is preempted with equal probability and thus a type j task loses the CPU to the type i task with probability s_j/L. This transition is

$$u_i \rightarrow u_i + 1$$

$$v_i \rightarrow v_i - 1$$

$$s_i \rightarrow s_i + 1$$

$$s_j \rightarrow s_j - 1$$

and the intensity for the transition is $v_i \nu_i (1-p_i) s_j/L$. Of course, if the type i task arrived to find an idle CPU, the transition would be

$$u_i \rightarrow u_i + 1$$

$$v_i \rightarrow v_i - 1$$

$$s_i \rightarrow s_i + 1$$

and the intensity in this case is $v_i \nu_i (1-p_i)$.

3) A task of type i currently in input/output may terminate and leave the system.

At completion of an I/O operation a task of type i terminates with probability p_i. With v_i type i tasks in I/O, the intensity of I/O completions for tasks of this type is $v_i \nu_i$. Thus for both the optimization and single queue models the state change for this transition may be written

$$v_i \rightarrow v_i - 1$$

and the transition intensity for the two models is $v_i \nu_i p_i$.

4) In the single queue model a task of type i currently using a central processor may exhaust its time slice.

For the optimization model the CPU is assigned on the basis of the priorities of the M types of tasks and so a time slice runout in this model would result in the same task type being re-assigned the CPU. Thus the same state would be occupied after time slice runout as before and so a time slice in this model has no effect on the transition intensity matrix for the submodels. For the single queue model, the CPU queue may be empty and in this case also time slice runout for a task is ineffective. Next suppose the CPU queue is not empty in the single queue model, and the time slice for a type i task runs out. As in 1) above the CPU released by the type i task is assigned probabilistically to tasks in the CPU queue. As stated above this is an approximation to the deterministic CPU assignment which would be made in a system where an ordered CPU queue is maintained. As in 1) above, the available

CPU is assigned to a type j task with probability

$$\frac{u_j - s_j}{\sum_{k=1}^{M} (u_k - s_k)}$$

The intensity with which a type i task gives up a CPU due to time slice runout is $s_i\rho$, and so the transition intensity here is

$$\frac{s_i\rho(u_j - s_j)}{\sum_{k=1}^{M} (u_k - s_k)}$$

The state change for this transition is expressed as

$$s_i \rightarrow s_i - 1$$

$$s_j \rightarrow s_j + 1$$

We now have an expression for the transition intensity from any submodel state to any other submodel state. The states which must occur in the state space include all states to which transition may be made from every possible initial state during the regeneration interval. In this interval many transitions may occur. Thus the state space must include at least all possible CPU-I/O configurations of tasks for the initial number of tasks in core and for all lesser numbers of tasks in core.

Our ultimate use for the submodels is to aid in determination of transition probabilities in the model by providing a probability for each task configuration in core at the end of a regeneration interval, given the task configuration at the beginning of the interval. Define a vector of state probabilities, $p(t) = (p_1(t), \ldots, p_z(t))$, for a submodel with z states. $p_i(t)$ is the probability that we are in state i of the submodel at time t. Define $p(0)$ to be the vector of state probabilities at the regeneration point. Then with regeneration interval $I^D_{i_1}$, or simply I_{i_1} for the single queue model, we are interested in determining $p(I^D_{i_1})$ or $p(I_{i_1})$. $p(t)$ must satisfy a system of linear differential equations with constant coefficients,

$$\dot{p}(t) = p(t)\ A \qquad (3.1)$$

Initial conditions necessary to solve this system of equations are provided by the known state of the model (i_1) at the regeneration point and the relationship between this state in the model and states in the submodel. The model state is

$$\begin{aligned} i_1 &= T(\alpha^1, \beta^1, \gamma^1) \\ &= T(f_1, \ldots, f_M, g_1, \ldots, g_M, h_1, \ldots, h_M) \end{aligned}$$

In the optimization model, the initial submodel state can be found very simply from this. There is a one - one correspondence between vectors β, γ in the model and states in a submodel. Suppose we denote the submodel state at the regeneration point by (ζ^1, η^1) where

$$\zeta^1 = (u_1^1, \ldots, u_M^1)$$
$$\eta^1 = (v_1^1, \ldots, v_M^1)$$

Then knowing i_1, we have

$$u_i^1 = g_i^1$$
$$v_i^1 = h_i^1$$

Note that the memory queue configuration does not matter here. Suppose we denote the initial submodel state by $j_1 = V(\zeta^1, \eta^1)$ where V is a function which maps the state vector into a linear index. Then for our initial conditions on the state probability vectors we have

$$p_{j_1}(0) = 1$$
$$p_j(0) = 0 \qquad j \neq j_1$$

With these initial conditions we may now solve the system of differential equations (3.1) and determine $p(I_{i_1}^D)$.

For the single queue model the relation between states in the model and those in the submodel is not quite so simple. In this case we have an additional state vector, ψ, representing the assignment of central processors to tasks. We have the same relationships as before between the vectors β, γ in the model and ζ, η in the submodel. Now, however, there may be more than one possible CPU assignment, ψ, for given ζ, η and so corresponding to vectors β^1, γ^1 of the state i_1, there may be multiple corresponding states; $j_1, \ldots, j_m$; in the submodel

corresponding to the possible initial CPU assignments in the submodel. The probability of occurrence for each initial CPU assignment is determined using the same approximation used earlier to assign a task to a CPU when one becomes available.

Let us consider the initial conditions for one and two central processors. Suppose j_1 corresponds to assignment of a type ℓ task to the single CPU. Then

$$p_{j_1}(0) = \frac{u_\ell}{\sum_{k=1}^{M} u_k}$$

As the other case suppose there are two CPU's. Suppose j_1 corresponds to a state in which both CPU's are assigned to type ℓ tasks and j_2 to a state in which one CPU is assigned to a type ℓ task and the other to a type j task. Then our initial probabilities for these two states in the submodel are

$$p_{j_1}(0) = \frac{u_\ell}{\sum_{k=1}^{M} u_k} \cdot \frac{(u_\ell - 1)}{\sum_{k=1}^{M} (u_k - 1)}$$

$$p_{j_2}(0) = \frac{2u_\ell u_j}{\sum_{k=1}^{M} (u_k) \sum_{k=1}^{M} (u_k - 1)} .$$

As in the optimization model, these initial conditions can be used to obtain solutions to the system of differential equations.

The solution to Equations (3.1) is well-known [19] . We have p(0) and the transition intensity matrix, A, and so the solution is

$$p(t) = p(0)\, e^{At}$$

where

$$e^{At} = I + At + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \ldots$$

Recall that the time between regeneration or decision points is restricted to integer values up to some maximum, say T_{max}. Therefore we need only compute e^{A}, e^{2A}, ..., $e^{T_{max}A}$. Computation of e^{A} is discussed in Appendix B.

We have enough information now to obtain solutions to all needed submodels. The question is, how many such submodels must be solved? Actually a more pertinent question is: for how many transition intensity matrices, A, must we compute e^{A}; since this is the difficult part from a computational viewpoint. We may determine all state probability vectors from a single transition intensity matrix for each CPU scheduling policy if desired. Simply set up a state space which includes the maximum number of every type of task in core with every possible configuration in core. Then in the state space for the matrix, A, we have all possible configurations of tasks in core for the model. Now we may compute e^{A}, $(e^{A})^2, \ldots, (e^{A})^{T_{max}}$ and by setting appropriate initial conditions p(0) for various states, i, of the model, may easily compute p(1), p(2), ..., $p(T_{max})$. It may, however, be more efficient to obtain solutions to a number of smaller submodels.

Let us take a simple example in which M=2 (two task types). The notation (x,y) will be used where x refers to the number of type 1 tasks in core and y refers to the number of type 2 tasks in core. Suppose there are enough pages of core (P) for 3 type 1 tasks (3,0) or for 2 type 2 tasks (0,2). It is also possible for 1 type 1 and 1 type 2 task (1,1) to share core simultaneously. In this case we consider a submodel which includes all possible states in which there are three type 1 tasks and two type 2 tasks in core (submodel(3,2)). For the optimization model we must consider all possible locations for all tasks: CPU or I/O. For type 1 tasks, there may be 0,1,2, or 3 tasks at the CPU at the same time there are 3,2,1, or 0 tasks in I/O. Similar configurations are possible for type 2 tasks. Since any number of tasks may terminate in the interval we must include in our state space all configurations with lesser numbers of tasks in core including the state in which all tasks have terminated and there are no tasks remaining in core . The number of states in this submodel is easily found to be 60.

On the other hand we could have chosen to solve three smaller submodels. Note that the submodel in the previous paragraph contains several states which are unusable since they correspond to no state in the model. We could have chosen to solve submodels (3,0), (0,2) and (1,1) and would have covered all model states this way. Note that the aggregate number of states for all three of these submodels is only 11 and that now we are working with 4x4, 3x3, and 4x4 A matrices for

a total of only 41 elements rather than a single 60x60 matrix or 3600 elements. It is clear that solution of the three smaller submodels is the preferable method.

3.7 Transition Probabilities

In this section we determine expressions for the transition probabilities for both the optimization model and single queue model. We are interested in determining the probability of transition from arbitrary state i_1 to arbitrary state i_2. The transition probability is denoted by $p^{D}_{i_1 i_2}$ for the optimization model and $p_{i_1 i_2}$ for the single queue model. The transition probabilities in the optimization model depend on the decision, D, chosen where $D = D(a_1, a_4)$. Recall that

$$i_1 = T(\alpha^1, \beta^1, \gamma^1)$$

$$i_2 = T(\alpha^2, \beta^2, \gamma^2)$$

That is i_1 and i_2 are determined by the state vectors α, β, and γ where α specifies the tasks in the memory queue, β specifies tasks at CPU's, and γ specifies tasks in I/O.

In the optimization model there is a one - to - one correspondence between states and decisions of the model defined at regeneration points and states of the submodels. That is, for initial state i_1 and decision D there exists a corresponding state j_1 in a submodel. These states are related through the decision D and vectors β^1 and γ^1 in the

model and vectors ζ^1 and η^1 in the submodel. The specific functional relationships are given in Section 3.6. Using the initial state i_1 we determine an initial state for the submodel, j_1. This gives us our submodel initial conditions

$$p_{j_1}(0) = 1$$

$$p_j(0) = 0 \qquad j \neq j_1$$

Knowing these allows solution of the submodel which gives us a probability vector $p(I^D_{i_1})$. This solution is discussed fully in section 3.6 where the submodels are described.

In particular, one element of this solution vector corresponds to state i_2 in the model. Call this element $p_{j_2}(I^D_{i_1})$. This gives the probability that

$$j_2 = V(\zeta^2, \eta^2)$$

occurs at the end of the regeneration interval where

$$\zeta^2 = (u_1^2, \ldots, u_M^2)$$

$$\eta^2 = (v_1^2, \ldots, v_M^2)$$

For state i_2 we have $i_2 = T(\alpha^2, \beta^2, \gamma^2)$

where

$$\beta^2 = (g_1^2, \ldots, g_M^2)$$

$$\gamma^2 = (h_1^2, \ldots, h_M^2)$$

Then the relationship between ζ^2, η^2 and β^2, γ^2 is (in terms of the elements of these vectors)

$$g_k^2 = u_k^2 + 1 \; ; \; k = a_1 \text{ (loading decision)}$$

$$= u_k^2 \qquad ; \; k \neq a_1$$

$$h_k^2 = v_k^2 \qquad ; \text{ all } k$$

Thus β^2 and γ^2 may be determined from ζ^2, η^2 so that the probability of occurrence of β^2, γ^2 may be found at the end of the regeneration interval given β^1, γ^1 at the beginning of the interval. This specifies the configuration of tasks in memory and so what remains is to determine the probability of occurrence of the new memory queue configuration, $\alpha^2 = (f_1^2, \ldots, f_M^2)$.

We may note here that the states to which transition may be made from i_1 are restricted by the possible values of α^2 in the following ways.

$$\sum_{\ell=1}^{M} (f_\ell^2 + g_\ell^2 + h_\ell^2) = N \qquad (3.2)$$

$$\Delta_\ell = f_\ell^2 - f_\ell^1 + \delta_{\ell a_1} \geq 0 \quad \forall \ell \qquad (3.3)$$

$\delta_{\ell a_1}$ is the Kronecker delta and thus is zero unless a type ℓ task was loaded in the regeneration interval, in which case it is one. Δ_ℓ is the number of new tasks of type ℓ that arrive at the memory queue during the interval $I_{i_1}^D$.

Recall that b_ℓ is an input parameter which is the probability that a

newly arriving task at the memory queue is of type ℓ independent of any previous arrivals or the status of the model. Therefore we may compute the probability that vector $\alpha^2=(f_1^2,\ldots,f_M^2)$ occurs conditional on the vector α_1 and the loading decision a_1. The multinomial coefficient, $\frac{(\sum_{\ell=1}^{M}\Delta_\ell)!}{(\Delta_1!)\ldots(\Delta_M!)}$, determines the number of ways in which Δ_i type i tasks arrive at the memory queue; i=1, 2, ..., M. Each order of arrival occurs with probability $\prod_{\ell=1}^{M} b_\ell^{\Delta_\ell}$ and so

$$\Pr\{\alpha^2 \mid \alpha^1, a_1\} = \frac{(\sum_{\ell=1}^{M}\Delta_\ell)!}{(\Delta_1!)\ldots(\Delta_M!)} \prod_{\ell=1}^{M} b_\ell^{\Delta_\ell} \qquad (3.4)$$

where the f_ℓ^2 $(\ell = 1,\ldots,M)$ are constrained by Equations (3.2) and (3.3). The probability of any vector α occuring which does not meet these constraints is zero.

We now give the transition probability which is

$$p_{i_1 i_2}^{D} = \left[p_{j_2}(I_{i_1}^{D})\right]\left[\frac{(\sum_{\ell=1}^{M}\Delta_\ell)!}{(\Delta_1!)\ldots(\Delta_M!)} \prod_{\ell=1}^{M} b_\ell^{\Delta_\ell}\right] \qquad (3.5)$$

To summarize briefly, it will generally be true that for most states, i_2, the transition probability $p_{i_1 i_2}^{D} = 0$. The only transitions of interest are those which can occur with non-zero probability, and these transition probabilities are generated by first using the given relations between model and submodel states to find the initial submodel state. Then numerical methods are used to find the state probability

vector for the submodel, $p(I_{i_1}^{D})$. For each submodel state occurring with probability greater than zero we again use these relations to determine the corresponding state variables in the decision point model. At this point the only state variables remaining to be determined are $\alpha^2 = (f_1^2, \ldots, f_M^2)$; these are constrained by Equations (3.2) and (3.3), and the probability of occurrence of any vector α^2 meeting the constraints is given by Equation (3.4). We now have probabilities of occurrence associated with all the state variables which make up state i_2. These probabilities are multiplied in Equation (3.5) to yield the transition probability $p_{i_1 i_2}^{D}$.

So far we have discussed only the optimization model. Determination of $p_{i_1 i_2}$ for the single queue model is somewhat more difficult for two reasons. First, in the optimization model the decision variable a_1 determines what task type is to be loaded into core from the memory queue while in the single queue model the task type to be loaded is determined probabilistically as outlined in the following paragraph. Second, in the optimization model the CPU assignments are deterministic and can be derived from the decision variable a_4 (CPU priority vector) and the tasks at the CPU's while in the single queue model the CPU is assigned in a more random fashion. This means that submodels require the vector $\psi = (s_1, \ldots, s_M)$ as part of their state specification and so there is now a one-to-many correspondence between states in the model and states in the submodels.

In the single queue model we want tasks entering the memory queue to be serviced on a FCFS basis. Modeling this exactly would require maintaining a record of the order of the memory queue and this would increase the number of states in the model to an unmanageable number. Therefore we approximate FCFS behavior by maintaining only a record of the number of tasks of each type in the memory queue. With f_k $(k=1,\ldots,M)$ tasks of type k in the memory queue we assume the probability that a type i task is at the head of the queue is $f_i/\sum f_k$. If there is room in memory for the type i task we load it; if not there is no load. Thus $f_i/\sum f_k$ is the probability of loading a task of type i providing there is room for it in memory. Assuming there is not room in memory for tasks of type $i_1,\ldots,i_n$ $(n \leq M)$ the probability that no task is loaded is

$$\Pr\{\text{no task loaded}\} = \frac{\sum_{\ell=1}^{n} f_{i_\ell}}{\sum_{k=1}^{M} f_k}$$

Since we do not actually know what task type is being loaded, if any, we do not know the length of the regeneration interval. The above probabilities permit computation of the expected regeneration interval, however, and we use this expectation rounded to the nearest integer as the value for I_{i_1}.

Two problems arise because the tasks are not assigned to CPU's in a deterministic fashion. Our goal is to determine $p_{i_1 i_2}$ and to do

this we must relate inital submodel states to i_1 and final submodel states to i_2. We have a one–one correspondence between β and γ in the model and ζ, η in the submodel as stated earlier. Now, however, there may be more than one possible CPU assignment (ψ). Suppose the initial submodel states corresponding to i_1 are $j_1, \ldots, j_m$ and those corresponding to i_2 are $k_1, \ldots, k_n$. Our initial conditions in the submodel are the possible initial assignment of CPU's to tasks. Proabilities for these assignments were given in Section 3.6 and are used to obtain solutions to the submodels. This may be summarized by saying that the initial state i_1 corresponds to more than one state in the submodel. Therefore we must assign initial probabilities to these states. In the optimization model the probability was simply one for the single corresponding state. The rationale for these particular probabilities was given earlier where we discussed assignment of CPU's to tasks in the CPU queue.

At the end of the regeneration interval we want to know the probability we are in i_2. Submodel states corresponding to i_2 are k_1, $\ldots, k_n$. Therefore we compute

$$\sum_{\ell=1}^{n} p_{k_\ell} (I_{i_1})$$

We are now ready to determine $p_{i_1 i_2}$. Define

$$\Delta_\ell^m = f_\ell^2 - f_\ell^1 + \delta_{\ell m}$$

$$\delta_{\ell m} = 1; \quad \ell = m \text{ and room in memory for type } \ell \text{ task}$$

$$= 0; \text{ otherwise}$$

Then an argument similar to that used to derive Equation (3.5) yields

$$\Pr\{\alpha^2 \mid \alpha^1\} = \sum_{m=1}^{M} \left[\frac{f_m^1}{\sum_{k=1}^{M} f_k^1} \cdot \frac{\left(\sum_{\ell=1}^{M} \Delta_\ell^m\right)!}{(\Delta_1^m!)\ldots(\Delta_M^m!)} \prod_{\ell=1}^{M} b_\ell^{\Delta_\ell^m} \right]$$

In this equation the constraints given in Equations (3.2) and (3.3) hold by replacing Δ_ℓ by Δ_ℓ^m. Any term where the constraints do not hold is set to zero. Finally we get

$$p_{i_1 i_2} = \sum_{\ell=1}^{n} p_{k_\ell}(I_{i_1}) \Pr\{\alpha^2 \mid \alpha^1\} .$$

3.8 Reward Matrix

The reward for making a transition from state i_1 to state i_2 is the number of task completions during the interval between the regeneration points. The goal of the optimization model is to determine a task scheduling rule which will maximize the number of completions per unit time. For the single queue model we wish to determine the number of completions per unit time for the scheduling rule used in this model. In either case, the one step reward is needed to determine this throughput. For the optimization model we apply the optimization method in Chapter 2 (time - optimal problem). For the single queue model the

problem is simply the trivial optimization problem where there is one alternative in each state and the same solution method applies.

The reward or number of completions in an interval for the optimization problem is

$$R^{D}_{i_1 i_2}(I^{D}_{i_1}) = \sum_{\ell=1}^{M} (f^{2}_{\ell} - f^{1}_{\ell} + \delta_{\ell a_1})$$

In the single queue model we may have loaded any one of several types of tasks so we have

$$R_{i_1 i_2}(I_{i_1}) = \sum_{m=1}^{M} \left(\frac{f^{1}_{m}}{\sum_{k=1}^{M} f^{1}_{k}} \sum_{\ell=1}^{M} (f^{2}_{\ell} - f^{1}_{\ell} + \delta_{\ell m}) \right)$$

At this point the model has been completely specified, and the optimization methods derived in Chapter 2 may be applied to it. It may be recalled that for certain conditions on the transition probabilities under the possible alternatuves it was shown that the bounds on the gain and the gain rate (which correspond to throughput here) could be guaranteed to converge. Applicability of these conditions to the current model is discussed in Appendix F.

3.9 Overhead

In the preceding sections of this chaper we have discussed models of multiprogrammed computer systems which utilize reentrant, shared procedure. The savings to be realized by sharing procedure were discussed (memory space savings, reduction of I/O traffic) but no additional costs associated with sharing information among users have been given.

There may be costs associated with sharing, however. Depending on the hardware configuration and how this is used by the operating system, it may be necessary for the supervisor to maintain and update information about pages available to be shared and what tasks are currently using these pages. The addition of these tables to main memory, and maintenance and use of these tables introduce an additional supervisory overhead over that for a system which does not share code. This overhead appears in the form of reduced main memory space available for tasks because of the additional system tables and in the form of increased CPU time necessary for execution of the additional bookkeeping tasks brought on by the use of sharing.

If we wish to compare the operation of a system under conditions where both sharing of information is and is not allowed, then for systems where there is additional overhead required to share information we must account for this in the model in order to obtain a valid comparison. A straightforward way to do this is to assume that CPU intervals are shorter for given tasks when information is not shared and to assume that the total memory requirements for a task's potentially sharable information is greater when the information is actually shared than when it is not. Suppose we are sharing information and our parameters for a type k task are c_k, d_k, and μ_k for the number of pages of sharable information, non-sharable information, and the central processor service rate for

this task. Then for the model where no sharing can occur we have

$$c'_k = 0$$

$$d'_k = c_k + d_k - \lambda_k$$

$$1/\mu'_k = 1/\mu_k - \phi_k$$

Here λ_k is the number of pages of tables needed to allow sharing and ϕ_k is the mean processor time needed in each CPU interval to carry out the supervisor tasks which handle sharing. Quantization of these variables may be difficult since there will usually be no way to observe a system running in both a sharing and non-sharing mode. On the other hand some hardware configurations allow information to be shared with essentially no additional overhead and in this case the relations above are not needed.

3.10 Summary

This chapter contains a detailed description of a mathematical model of multiprogrammed computer systems. The model was developed to allow comparison of task scheduling policies and the use or non-use of shared information and the effects of these on system throughput in a heavily loaded computer system.

Tasks which have been loaded into the execution store of the computer are characterized as alternating between requests for the CPU and requests for input or output operations. A queue is maintained at the CPU (or possibly two CPU's) and variation of task priorities in this queue may be investigated for effects on throughput.

A second queue is maintained for requests for execution of tasks. These requests are not eligible to execute until they are loaded into the execution store. This is again a scheduling decision and effects on throughput when the policy here is changed may also be investigated.

Parameters of the model which may be varied include the size of the execution store, and the use or non-use of shared information (e. g. reentrant procedure) by tasks. In the next chapter we compare system throughput under optimal and heuristic scheduling policies, and as a function of these parameters.

Chapter 4

EFFECTS OF SCHEDULING AND SHARED INFORMATION

In this chapter we present results of application of the models described in Chapter 3. The data collected and analyzed from the MTS system were used to obtain parameters for the mathematical models and results of using these parameters are shown. Much of these data is presented in Appendix D. Parameterization of the model to represent MTS and results of that representation are given in the first ten sections of this chapter. In the last section we hypothesize other systems suitable for use with optimal scheduling rules and shared information and present results for these.

4.1 Scheduled Tasks

Only a subset of all tasks run on MTS were deemed suitable for consideration in the scheduling model, and reasons for this are discussed below. The models described in Chapter 3 are useful for determining the increase in task completions per unit time (throughput) possible by sharing information among tasks and/or scheduling the memory and the central processors optimally for these tasks. The models are aimed at using system resources most efficiently without regard to the response given individual users. Maximizing throughput assures minimum average response to users, but individual response times may be unacceptably large for some of the tasks in a time-shared system

such as MTS. In a system of this type users who make trivial requests expect and should receive short response times, and so for this reason trivial tasks are not suitable for the scheduling model. In addition, since the demand on system resources made by these tasks is minimal, the advantage gained by scheduling them to make best use of reentrant code would be small.

A good example of a task of this type is the line file editor (*ED) in the MTS system. This is a highly interactive program run by terminal users who wish to make alterations, additions, or deletions to line files stored on disks. The user issues a command to make a particular change from the terminal and a response is made to the terminal when the change is complete. A typical command requires a very small amount of CPU time to complete and the editor has small memory requirements as well. Delaying response to a command in order to use system resources more efficiently would result in little savings while the delay to the user who normally must wait for a response before giving a new command might well be intolerable.

For the scheduling model data it was decided to consider only reasonably large (15 or more virtual pages) tasks. Furthermore, attention was restricted to public tasks since these are the only tasks

for which a priori information may reasonably be obtained about their behavior during execution, and knowledge of this behavior is needed in order to make intelligent scheduling decisions. In addition, known interactive tasks were excluded from the model. These were PIL (Pittsburgh Interpretive Language) and DCALC, a program which allows the terminal to be used much like a sophisticated desk calculator.

Thus the tasks considered in the scheduling model were those tasks which have a known behavior and which make non-trivial demands on the system. In the remainder of this chapter, these tasks will be referred to as scheduled tasks while other tasks using the system will be referred to as non-scheduled tasks.

4.2 Use of Reentrant Code

All the MTS data analyzed in this report were collected over a period of about 22 minutes at a time when the system was under heavy load. By far the most heavily used program during the data collection period was the Fortran compiler (41 requests). Next among the tasks considered in the scheduling model (scheduled tasks) was the smaller of the two University of Waterloo Fortrancompilers (*SWAT) with 13 requests. The assembler was run eight times and the Snobol 4 compiler twice. Since data were collected for approximately twenty-

two minutes we see that requests for Fortran arrive at a rate of about two per minute so there would appear to be some advantage in the use of a reentrant compiler. On the other hand, the rate of arrival of the other types of tasks was felt to be so low that they would not warrant the use of reentrant code. This is especially true because the Waterloo Fortran compiler executes the programs it compiles, and of course little sharing would be possible during this phase of execution anyway. Thus the only program assumed reentrant in the model was the IBM Fortran IV compiler.

It has been estimated that if the IBM Fortran IV (G) compiler were made reentrant there would be approximately nine pages of procedure which could be shared. The remainder of the required storage would be used for tables unique to each compilation. The data show that tasks used an average 30 virtual pages during Fortran compilations and an average 25 real pages.

For completeness it should be mentioned that three pages in virtual memory consist of tables used by system routines to monitor the status of the user and are thus not part of the compiler itself. They still must be treated as part of a user's virtual (and possibly real) memory, of course.

If we assume that the ratio of real pages to virtual pages in the Fortran compiler is the same for both procedure and data, we arrive at 7.5 procedure pages in core and 17.5 data pages in core. Seven

procedure pages and eighteen data pages were chosen for the model. If two or more tasks are using the compiler simultaneously they may not be referring to the same procedure pages. Table A.1 in Appendix A may be used to estimate the number of procedure pages in core when more than one compilation is proceeding. In particular we may look at the table for c equal to 9 and interpolate w and q. Rounding, we find that with two compilations in memory we would expect eight procedure pages and with three or more compilations all nine procedure pages.

The remainder of the scheduled tasks are assumed non-reentrant. From the data they were found to have an average of twenty-two pages in core.

4.3 CPU and I/O Interval Distributions

As part of their input data, the models developed in Chapter 3 require for each task the mean CPU interval between I/O requests and the mean interval between request for an I/O operation and completion of that operation. In addition, the empirical probability distribution functions for these intervals are of interest because these distributions may be used to check the assumption that the intervals are exponentially or hyperexponentially distributed.

Figures D.2 and D.3 in Appendix D show the cumulative distributions of CPU intervals and I/O intervals for the Fortran compiler. Exponential and hyperexponential distributions with the same mean

are shown also. It is clear that the hyperexponential curve fits the data much better than the exponential curve in each case. The same quantities are plotted in Figures4. 1 and 4. 2 for the other scheduled tasks. Again the hyperexponential fit is seen to be significantly better than the exponential fit.

The disadvantage of using hyperexpontential rather than exponential distributions in the model is that this significantly increases the number of states in the model. This in turn limits the range of other parameters which may be chosen because of computational costs.

Results were obtained using both exponential and hyperexponential distributions and compared. The throughput was found to differ by less than one percent in the cases compared. On this basis it was decided to assume exponential distributions for the remainder of the models.

4.4 Paging

The MTS data indicate that the number of pages a task has in core does not vary widely during that task's execution, and in fact that the number of virtual and real pages a particular task type is assigned does not vary widely from one execution to the next. For example, Table D.2 in Appendix D shows that over all the Fortran executions during the data collection period, the virtual pages

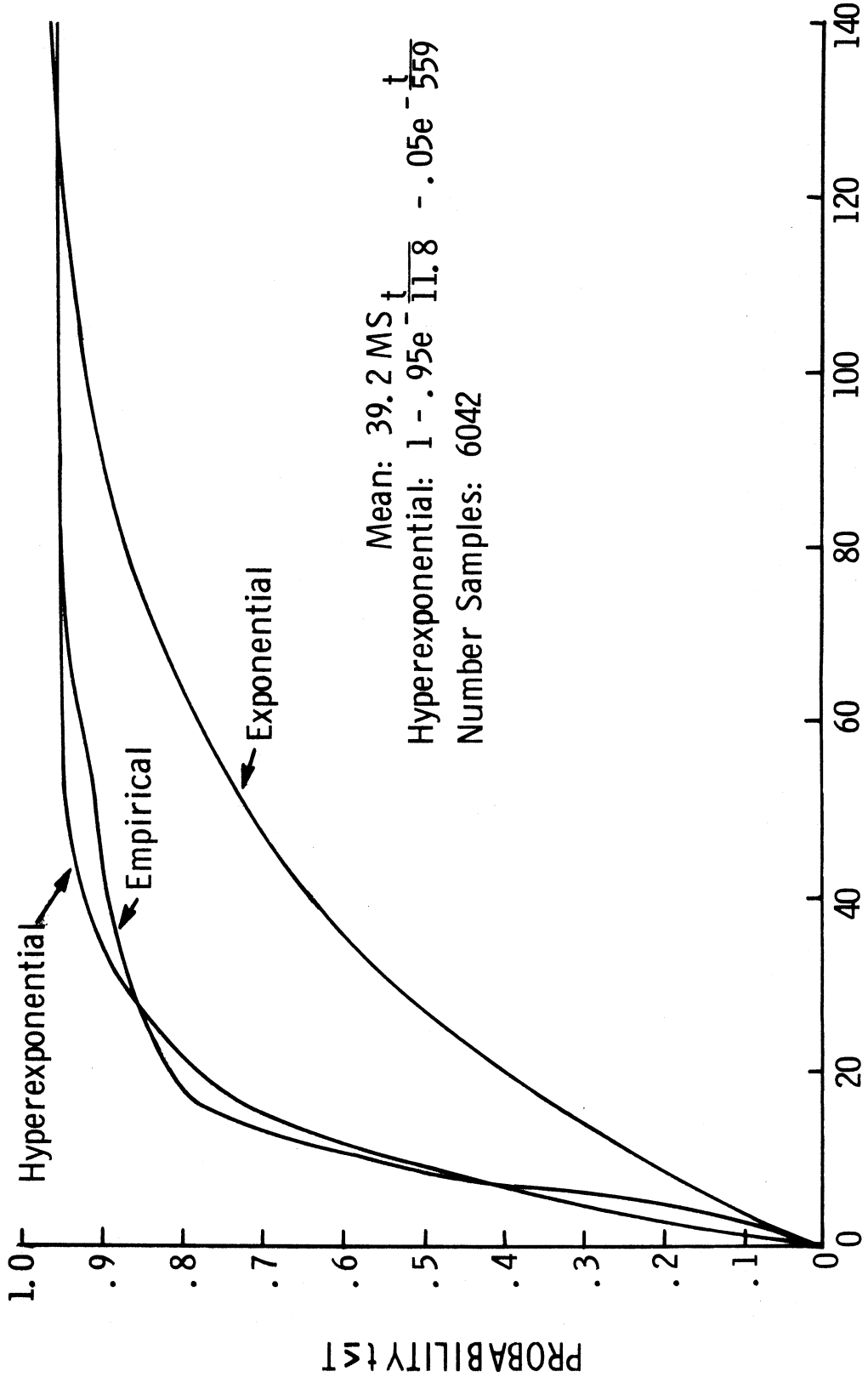

Figure 4.1. Cumulative Distribution of CPU Time (t) Used Between I/O Operations By Scheduled Tasks Other Than Fortran

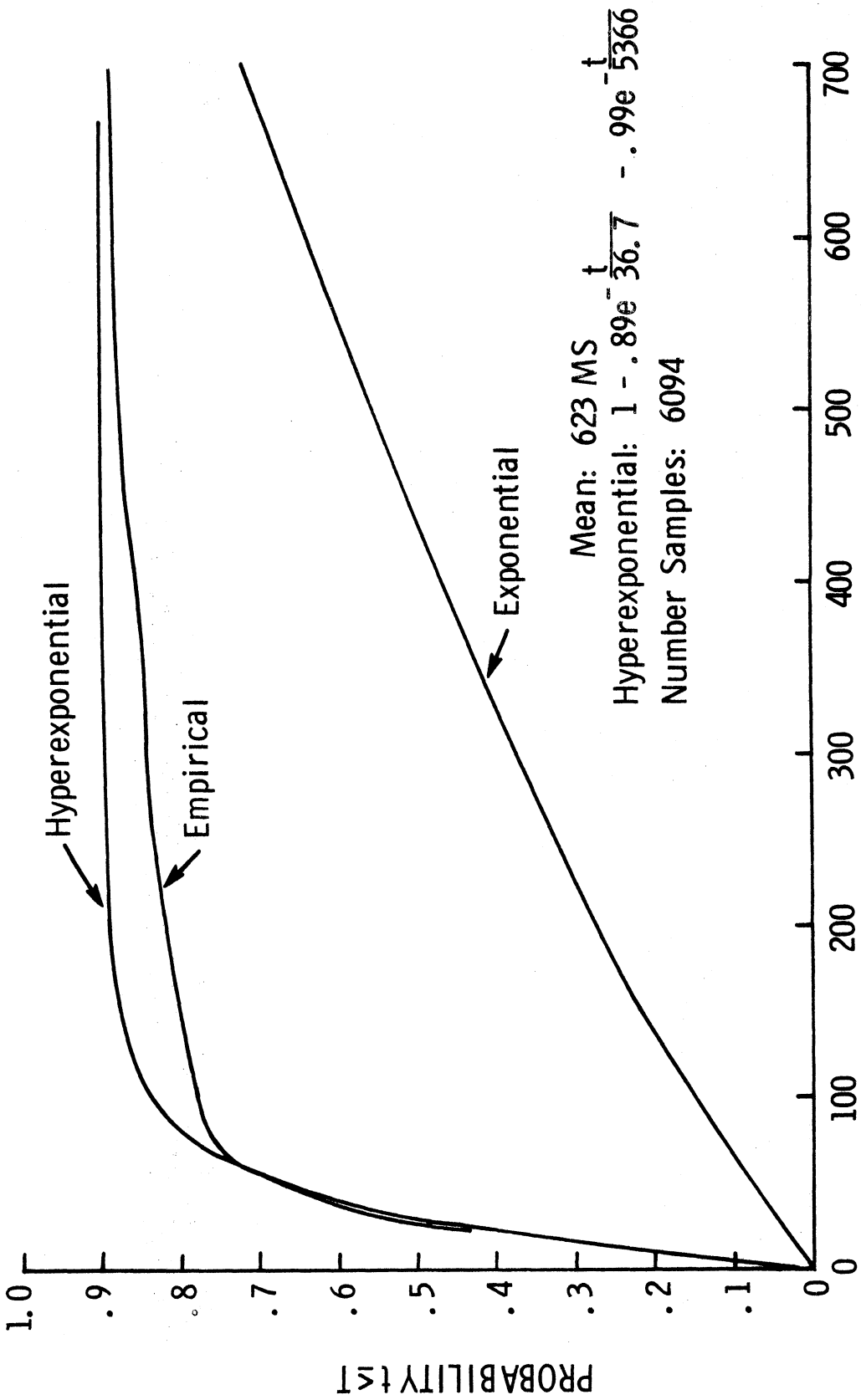

Figure 4.2. Cumulative Distribution Of Time (t) Used Per I/O Request By Scheduled Tasks Other Than Fortran

averaged around 30 with a standard deviation of only 9 while the real pages averaged around 25 with a standard deviation of only 8. This observation makes it reasonable to assume that a task executes with a constant number of pages in main memory. Furthermore it lends credence to the assumption that all tasks of the same type have the same number of pages (further justified in Appendix C).

In the model, page waits were lumped together with other kinds of I/O requests. It is interesting to note that the total time spent in page wait is generally low compared to that spent doing other types of input-output and further there are usually significantly fewer page waits than other I/O operations. Again as an example, Table D.2 shows 8.13 seconds spent in I/O (excluding paging) and only .434 seconds in page wait for the average Fortran execution. In the same table it is seen that on the average there were 63.1 I/O waits per execution but only 14.4 page waits per execution. Physically, these facts indicate that the results would not change greatly if paging were ignored completely in the model. Inclusion of paging in the model amounts to adding a relatively small number of short I/O requests to the other I/O requests made by the task.

4.5 Number of CPU - I/O Intervals

In the model it is assumed that a task of a particular type alternates between use of the CPU and I/O devices until it terminates. The number of such CPU - I/O cycles is assumed geometrically

distributed. Figures 4.3 and 4.4 show the empirical distributions for both Fortran and the other scheduled tasks plotted along with geometric distributions of the same mean. The geometric assumption appears reasonable in both cases. In these curves we have included page waits as well as other I/O. Similar graphs were plotted which exclude page waits and it was found that the fit to a geometric distribution is still good. These latter curves are not shown here because they are very similar to Figures 4.3 and 4.4.

4.6 Central Processors

The Michigan Terminal System has two central processors which have equal status in the system. Therefore two central processors were used in the model of MTS. A difficulty arises with the model of the central processors because the tasks under consideration in the scheduling model are not all the tasks making use of MTS. The scheduled tasks used about 17 % of the total available processor time during the data collection period with the remainder of the processor time being used by other MTS tasks and by the supervisor.

The problem then is to account for the interference with the scheduled tasks by other tasks in the system, and this may reasonably be done by assuming the CPU runs at some fraction of its true rate. This assumption becomes more attractive when it is realized that a task in MTS is unlikely to finish a CPU interval without interruption anyway.

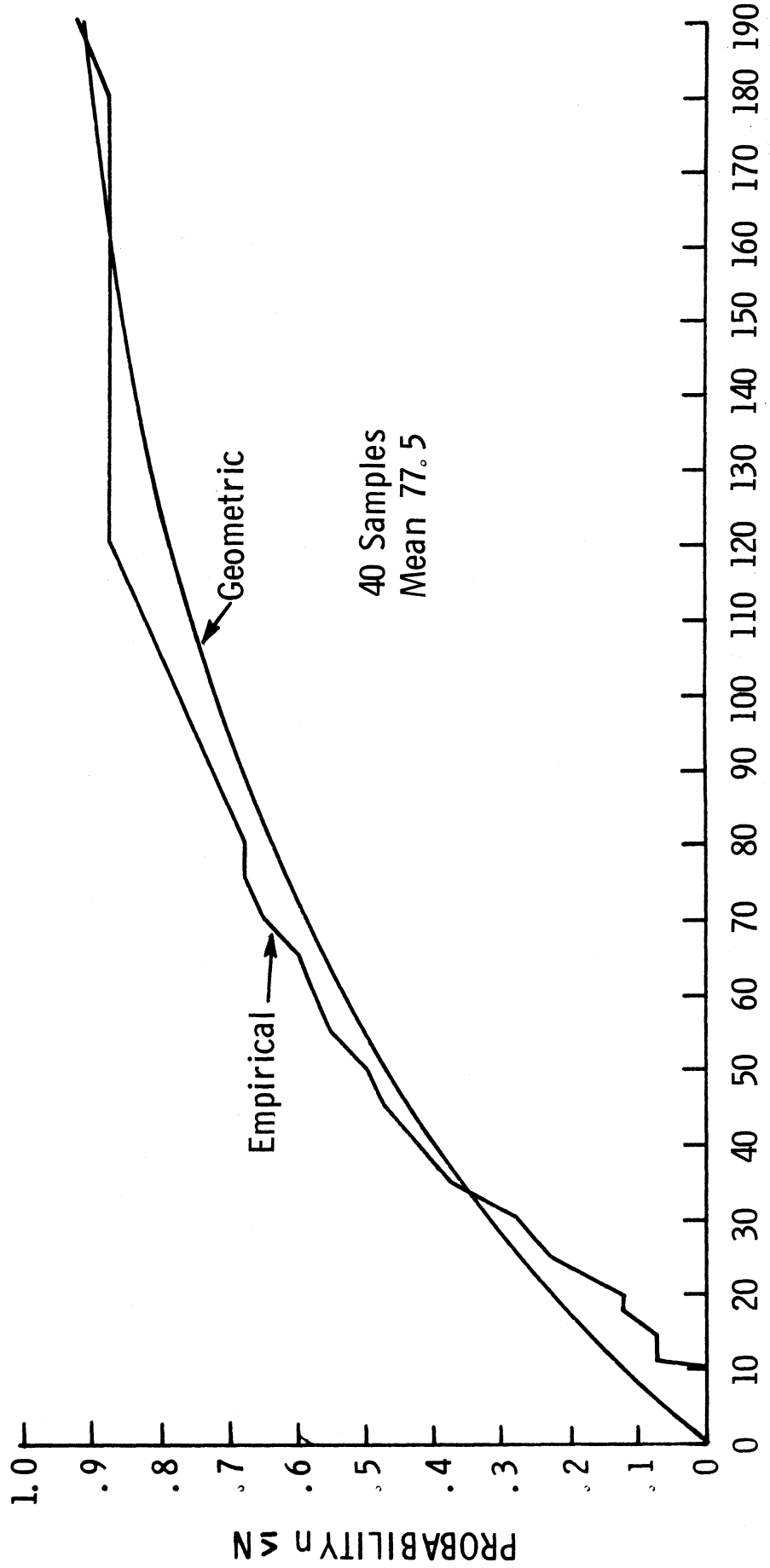

Figure 4.3. Cumulative Distribution of Number (n) CPU - I/O Cycles Before Termination For Fortran Compiler

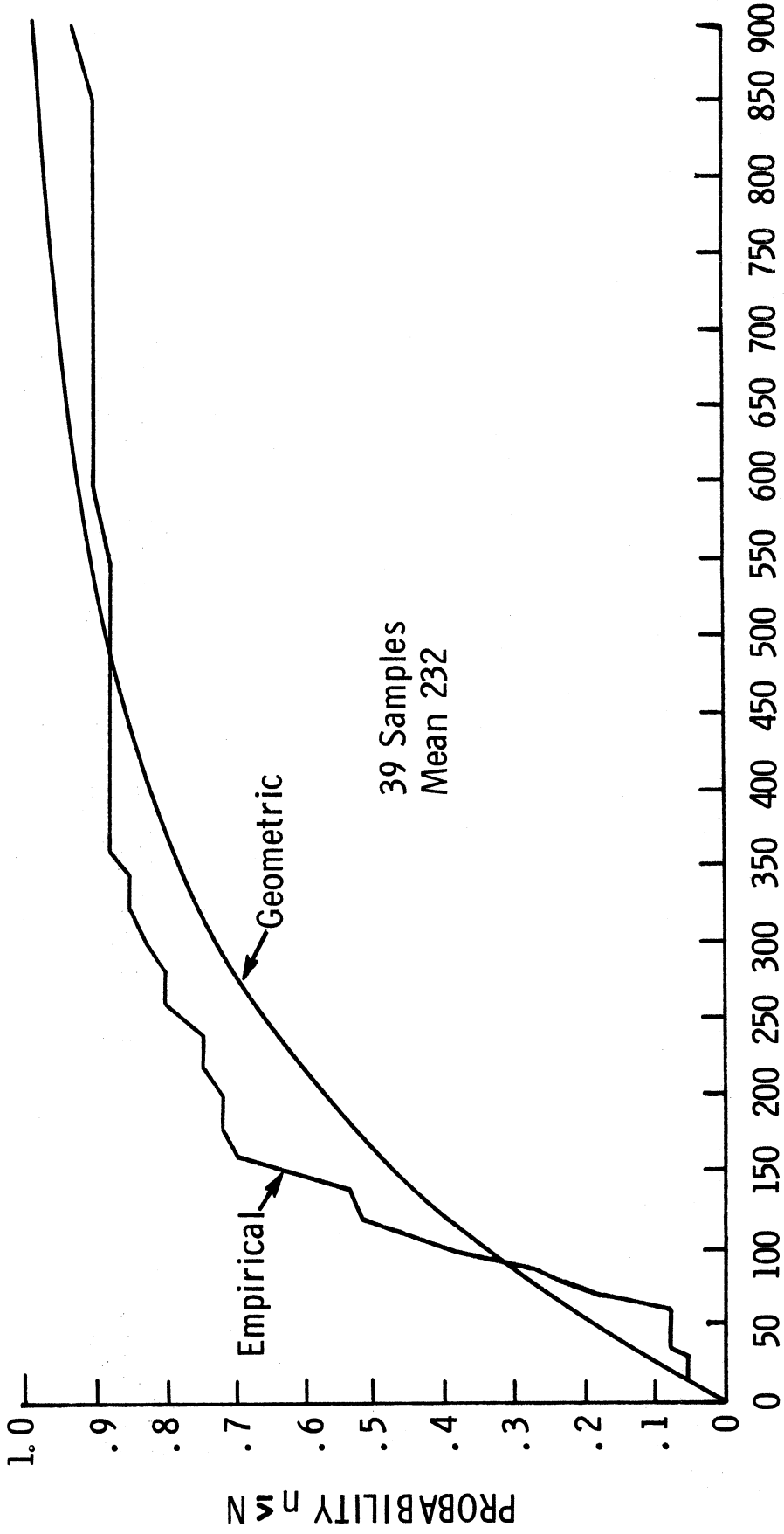

Figure 4.4. Cumulative Distribution of Number (n) CPU - I/O Cycles Before Termination For Scheduled Tasks Other Than Fortran

The CPU switched from one task to another an average of once every 5.7 milliseconds during the data collection period. Thus in MTS a task is very likely to have a CPU interval between I/O requests broken into two or more shorter intervals with periods of being queued at the CPU in between. This happens because tasks which complete an I/O or paging operation receive the CPU immediately (LCFS) rather than going to the end of the CPU queue and thus cause a service interruption for the task using the CPU at the time of the I/O completion.

The difficulty with assuming a CPU running at a fraction of its true rate lies in determining just what that rate should be in order to account properly for interference by other, non-scheduled tasks in the system. Since it was not possible to determine what the effective CPU rate should be, this problem was handled by obtaining results for two CPU rates: one known to be higher and the other lower than the true effective rate. These results are open to other interesting interpretations which are discussed in Section 4.8 where the results are presented. The CPU rates chosen were a rate 17% that of the true CPU rate and a rate equal to the full CPU rate for the following reasons. The effective CPU rate is determined by the fraction of the available CPU time used by scheduled tasks when one or more are queued at the central processors. Since the data show the scheduled tasks use 17% of the total available CPU time, these tasks must use more than 17% of the available CPU time when one or more are queued

at the CPU; and so a 17% rate is lower than the effective rate. On the other hand, when scheduled tasks are queued at the CPU, they undoubtedly do not monopolize the CPU and so a 100% rate is higher than the effective rate. Thus, assuming a CPU running at 17% its full rate overstates the amount of interference expected from non-scheduled tasks while a CPU running at 100% its full rate no doubt understates this interference.

4.7 Loading Scheduled Tasks

The time required to load a task is assumed fixed for each task in the model. For the Fortran compiler this is quite a good assumption because the mean loading time is 5.1 seconds with a standard deviation of 1.8. For the other scheduled tasks the variation in loading time is substantially greater, but the assumption of a constant loading time has little effect on the results because load times are short compared to execution times. Under these circumstances it is the time spent in execution which is the major factor in determining throughput and not the time required to load the task.

The elapsed time required to load Fortran and the other scheduled tasks averaged 5.1 and 9.1 seconds, respectively. Loading times in the model are restricted to integer multiples of the shortest loading time up to a maximum multiple of three. The loading time for Fortran was chosen to be 5.6 seconds and 8.4 seconds for the other scheduled tasks. If procedure is sharable and a user requests the Fortran com-

piler when it is already in use, the procedure need not be loaded. The time required to set up the non-sharable storage for the new request is assumed to be 2.8 seconds.

One other point should be mentioned regarding the loading of scheduled tasks. No direct consideration is given to the CPU time used by the loader. This time is lumped with the CPU time used by tasks other than the scheduled tasks in the model. From the data it was found that 1.67 seconds average CPU time were required to load the Fortran compiler.

4.8 MTS Results

Results are presented in this section which show throughput of scheduled tasks when system parameters are similar to those in MTS. Throughput is plotted for both reentrant and non-reentrant Fortran compilers and for optimal scheduling and scheduling similar to that in MTS. The model for MTS scheduling is presented in Chapter 3 and consists of an approximation to FCFS in the memory queue and LCFS in the CPU queue for jobs arriving in this queue after completion of an I/O operation. MTS is also assumed to give tasks a 0.1 second time slice at the CPU. The results for MTS scheduling are labelled "Single Queue" on the graphs to distinguish them from results obtained when optimal scheduling is used.

The results obtained from the model with values of parameters derived from data obtained from MTS are given in terms of task completions per second in Figures 4.5 and 4.6. With these results it is

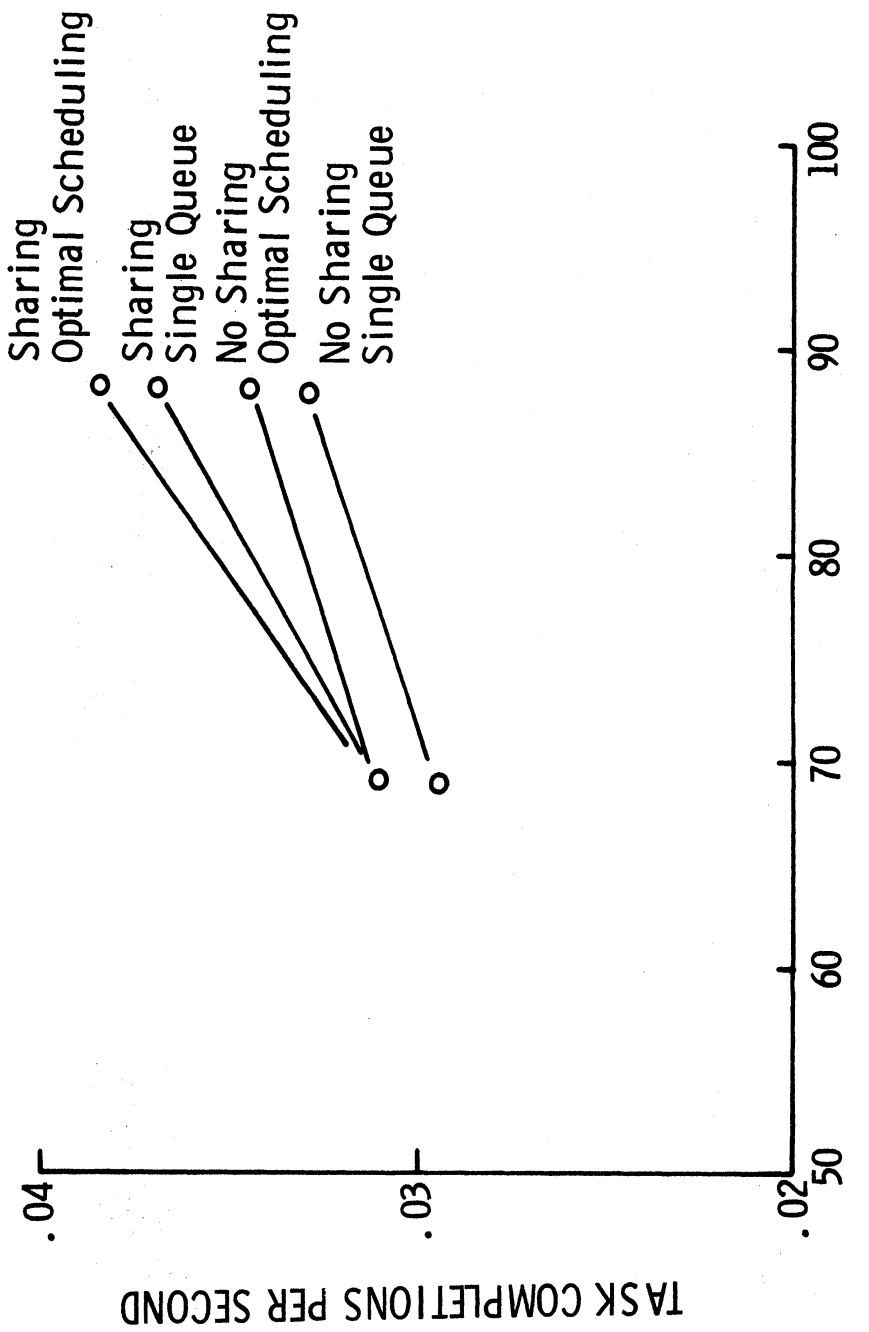

Figure 4.5. Comparison of Throughput Using Parameters Derived From MTS Data

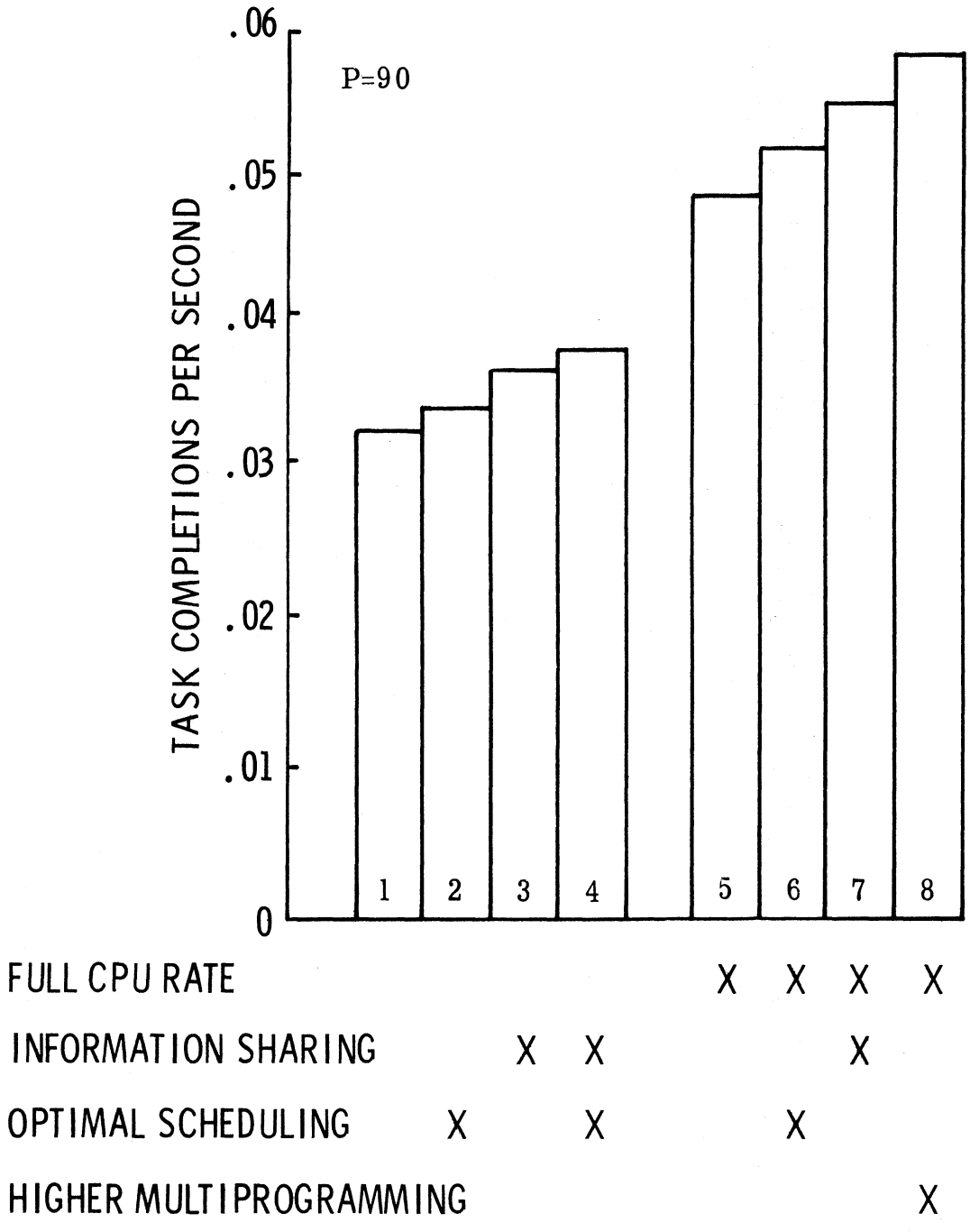

Figure 4.6. Throughput Using Parameters Derived From MTS Data

possible to compare system throughput of scheduled tasks when such things as memory size, scheduling policy, and use of reentrant code are varied. Values for the parameters of these figures are given in Table 4.1, and the results given in the figures are explained and interpreted in the following paragraphs. The parameters in Table 4.1 are those described in Chapter 3 where the scheduling models are developed. Tables in Appendix G give the policies which were found to be optimal and which produced the results shown in the figures. Parameters for the low degree of multiprogramming (columns marked L in Table 4.1) come directly from the data while those marked H are derived in section 4.10. For the low degree of multiprogramming, the termination probabilities are the reciprocals of the mean numbers of CPU - I/O cycles shown in Figures 4.3 and 4.4. The mean CPU interval (full rate) for Fortran is obtained from Table D.2 by dividing the CPU time per run by the sum of the I/O requests and page waits per run. The mean I/O interval for Fortran is also obtained from Table D.2 by dividing the sum of the I/O time and page wait time per run by the sum of the number of I/O requests and page waits per run. The means of the probability distributions shown in Figures D.2 and D.3 were not used for the mean CPU and I/O intervals because these figures do not include page waits. Mean CPU and I/O intervals for the other scheduled tasks were obtained in an analogous fashion although no table such as Table D.2 is included for these tasks.

Table 4.1

Parameter Values Derived From MTS Data

Task types	(M)	2
Tasks	(N)	6
Pages available	(P)	70, 90
Central processors		2
Time slice (sec)		0.1

ITEM	FORTRAN		OTHER	
Arrival probability (b)	.5		.5	
Loading time (sec)				
First task	5.6		8.4	
Additional tasks	2.8		8.4	
DEGREE OF MULTIPROGRAMMING	L	H	L	H
Termination probability (p)	.0129	.0119	.00431	.00403
Mean CPU interval (sec)				
17% CPU rate	.363		.166	
Full CPU rate	.0632	.0583	.0288	.0270
Mean I/O interval (sec)	.110	.104	.418	.393
Non-sharable pages in core	18	16	22	20
Sharable pages in core				
One task in core	7	6	0	0
Two tasks in core	8	8	0	0
Three or more tasks	9	9	0	0

In section 4.6 it was pointed out that non-scheduled tasks may interfere with scheduled tasks at the central processors, and therefore there is a need to modify the CPU rate to account for this interference. In the model two CPU rates were used: one which was 17% of the true rate and another which was the full CPU rate. This appears in the model as a variation in the mean CPU intervals used between I/O operations by tasks. At the full CPU rate we use the mean CPU interval as determined from the data while at a 17% CPU rate we use that interval divided by .17. With the CPU running at a 17% rate we are in effect representing a system more heavily loaded with non-scheduled tasks than the actual system was at the time of the data collection. Another interpretation for this amount of interference with scheduled tasks might be a system in which non-scheduled tasks are given priority at the central processors over scheduled tasks.

Whereas a CPU running at 17% its true rate represents a situation in which more interference with scheduled tasks occurs than is actually the case in MTS, a CPU running at its full rate represents a situation with less CPU interference than is actually the case. This is also open to interesting interpretations. This situation would occur where scheduled tasks are given priority over non-scheduled tasks. By far the most interesting and useful interpretation, however, is that of assuming a system with three central processors where the

interference of scheduled tasks which would occur in a two processor system is taken care of by the third processor. This is an especially interesting interpretation because the natural next step in expansion of the MTS system is to add a third processor, since there is in fact no hope of raising throughput any great amount in the system as presently constituted no matter what kind of sharing of information or scheduling is done. A look at the data reveals that during the data collection period the CPU was idle only five percent of the time and so this is the maximum increase in throughput attainable during this period. Adding a third CPU with no increase in main memory might well make the use of optimal scheduling and sharing of information more attractive as a means of increasing CPU utilization and thereby throughput by more efficient use of real memory which would be relatively less abundant in this situation. As a final word on interpretation of the CPU rates, the true effective rate lies somewhere between the two rates given and so the true throughput should lie somewhere between the extremes given in the results, and more importantly the relative advantage of changing operating policies should lie between these results as given at the two CPU rates.

The quantity of real memory allotted scheduled tasks was chosen to be 70 or 90 pages for all the results presented. The results in Figure 4.5 show the variation in throughput as available memory goes from 70 to 90 pages. The CPU rate for this figure

is 17% the true rate and the number of pages allotted each task was the same as the number allotted by MTS during the data collection period. This corresponds to the column marked L in Table 4.1 and represents the lower of the two degrees of multiprogramming considered. Results with a higher degree of multiprogramming will be discussed in section 4.10 in detail. The mean number of pages actually allotted scheduled tasks by MTS during the time data were collected is close to 90 pages. All results in Figure 4.6 are for this number of available real core pages. The relation between available memory for scheduled tasks and the number of pages of memory allotted each such task determines the number of tasks it is possible to multiprogram. Because of the assumption in the model of fixed memory requirements for each task, throughput as a function of available memory is not continuous but has jumps at points where it is possible to multiprogram some new combination of tasks.

The results in Figure 4.6 show a maximum increase in throughput of scheduled tasks of 16% by going from non-reentrant code and MTS scheduling to reentrant code and optimal scheduling and using a 17% CPU rate. For 17% CPU rate or full CPU rate, the increase in throughput in going from non-reentrant code to reentrant code is around 12% while going from MTS scheduling to optimal scheduling results in a throughput increase of around 5 to 7%. The important thing to note here is that these increases are only for scheduled

tasks, and that throughput of non-scheduled tasks probably would not change significantly. Therefore total system throughput would change significantly less than the amounts given above: for example, a throughput increase of 16% for tasks using 17% of the total CPU time results in an increase in CPU utilization of only around 3%.

4.9 Implementation

In the previous section, discussion centered on the increase in throughput attainable in MTS by use of optimal scheduling rules and by use of a reentrant Fortran compiler. In this section, consideration is given to the ways in which such sharing of information among tasks and such scheduling at the CPU's and main memory might be implemented in MTS.

If the Fortran compiler were reentrant (it currently is not) then there would remain the problem of sharing this among users. There are two ways this may be done. A request for the compiler may result in a search of system tables to determine if the compiler is currently in use by another user, and if not, the compiler may be loaded. If it is in use then a segment table entry is set up to point to the appropriate page table for the sharable pages. (See Arden, et. al. [2] for a discussion of the memory organization assumed here.) This method of operation has been found to require excessive overhead. To get around this the compiler may be made a permanent part of every user's virtual memory. Here the compiler occupies the same segment of

everyone's virtual store.

Because virtual storage is so large, the loss in storage for those who do not use the compiler but still have it in their address space is insignificant. Thus it is reasonable to assume that the Fortran compiler may be shared with no significant increase in system overhead. This mode of operation has the further advantage that the sharable procedure need never be loaded for any user and thus would result in less elapsed time per compilation for users and in a saving of CPU time for the system. (2.6% of the total available CPU time was used in loading the Fortran compiler during the data collection period.)

Implementation of an optimal scheduling algorithm for some MTS tasks while scheduling others with the same algorithm currently in use requires an understanding of how tasks in MTS are currently scheduled. In the following paragraphs a brief explanation of MTS scheduling at memory and the CPU's is given in which we refer to scheduled tasks and the scheduler. As stated earlier, scheduled tasks are those tasks which were considered suitable for scheduling in a way to maximize the throughput without regard to individual response times; and the scheduler refers to an extension to the MTS supervisor which would implement the scheduling policy determined to be optimal for these tasks.

In a virtual memory computer the degree of multiprogramming (number of programs being simultaneously multiprogrammed) is not

limited by the core memory available since the pages of a task do not have to be in core. It would be possible in most cases to operate such a system by loading every new task on demand, thereby eliminating any queuing at the memory. In such a situation the throughput is likely to increase up to a point as the degree of multiprogramming increases from zero. When the degree of multiprogramming increases beyond this point, throughput decreases because of excessive paging. The degree of multiprogramming is such that each task has too few pages in core to allow it to execute for more than a very short time between page faults, and so a situation arises in which CPU utilization is low because much of the time all tasks in the system are waiting for requested pages to be brought from secondary storage. This situation has been called thrashing and is obviously undesirable.

MTS avoids thrashing by controlling the number of users who have access to the memory in several distinct ways. Arriving batch tasks are queued outside main memory and are executed at a controlled rate which decreases as the number of currently active terminal tasks increase while at the same time the number of terminal users is limited by the total number of telephone lines connected to the system (around 80). In addition, if a task attempts to obtain a large number of real pages at a time when other tasks are using a large number of real pages, this task may be prevented from further execution until a time when demand for real core by other tasks is

not so great. This last restriction is the most direct method of preventing thrashing, but the other restrictions are important also.

Thrashing of scheduled tasks under a policy determined to be optimal in the scheduling model is prevented simply by setting an upper limit on the number of tasks which may share core at any one time. This means that direct control of the degree of multiprogramming for scheduled tasks is used in place of controlling the rate of execution of batch tasks as described above. In the model this control is exercised by assuming each scheduled task requires a known amount of storage in order to execute without excessive paging, and this amount of storage must be free before the task is loaded. There is a fixed total quantity of storage available to the aggregate of scheduled tasks so this quantity minus the amount already in use gives the storage available to any task in the memory queue.

Implementation of a scheduling policy similar to that found to be optimal for scheduled tasks in the model would require establishment of two memory queues: one for each type of scheduled task. Requests by batch or terminal users for execution of tasks identified by the supervisor as scheduled tasks would cause an entry in the appropriate memory queue. The scheduler would be table driven and would decide what type of task, if any, to load based on other scheduled tasks already loaded and in the memory queues. Loading of scheduled tasks would be done in this way rather than by loading of terminal tasks on demand and loading of batch tasks at a rate determined by

the terminal user load on the system. The order of service for each of these queues would, however, be immaterial and so terminal tasks in each queue could be serviced ahead of batch tasks if desired. CPU queuing must also be handled specially for scheduled tasks. Entries would be made into separate CPU queues for each type of scheduled task, but an entry would also be made in the CPU queue currently maintained by MTS for all tasks. The entry in this latter queue would simply indicate that a scheduled task occupies the queue position so that the scheduler can determine which type of scheduled task should actually receive the CPU when one of these entries arrives at the head of the queue. In this way the scheduler is able to assign the CPU to either type of scheduled task without interfering with the normal use of the CPU by other tasks in the system.

The scheduling model is cyclic with a constant total number of tasks while the number of scheduled tasks in MTS may vary. Therefore the status of scheduled tasks in MTS may not correspond to any state in the scheduling model, but the policy determined to be optimal in the model with parameter values derived from MTS data may be used as a guide in setting up a table to drive the scheduler. If desired, optimal policies could be determined for several values of N(total number of tasks in system) and in this way as many entries as desired in the scheduling table can be directly determined. Thus the model does not provide an optimal scheduling policy directly but provides

a guide for the scheduling decisions which should be made and also provides a good estimate of the gain in throughput attainable with such a policy.

Although sharing code may result in little additional system overhead, implementation of an optimal scheduling policy is another matter. This would require maintaining special queues for scheduled tasks and observing the status of these queues each time a CPU assignment decision or loading decision is to be made for a scheduled task. The overhead required to do this in MTS might well outweigh any advantage which might be gained. A computing system with a much more restricted class of users would be much more suitable for implementation of a policy like this. Here the optimal scheduler could be used for all tasks in the system and a greater throughput gain could be achieved for the same cost.

4.10 Increased Multiprogramming

We turn now to a discussion of the one case in Figure 4.6 where the number of real pages allotted each task was decreased in order to increase the degree of multiprogramming possible. A simple increase in the degree of multiprogramming would normally tend to increase system throughput, but in this case there is increased multiprogramming with fixed total memory allocation so that the number of page faults during task execution will increase which will tend to decrease throughput. Therefore it is of interest to look at the net effect of this change in core allocation. The

number of pages of core allocated tasks is given in the columns labelled H in Table 4.1 along with other parameter values for this situation: these represent a decrease of about 10% in the number of pages allocated each task.

The major difficulty in obtaining results for reduced core allocation lies in determining the expected number of page faults during execution of a task for this situation. Figures 4.7 and 4.8 are approximate curves showing the CPU time between page faults as a function of the number pf pages in core for the two task classifications used in the model. Figure 4.7 was determined in the obvious way from the data given in Table D.3 while Figure 4.8 was found in a similar fashion for scheduled tasks other than Fortran. These curves are only approximate because the number of pages in core between page faults is not necessarily fixed so that only an average number in core can be obtained, and also because the data shown in Table D.3 give mean time between page faults for a range of pages in core and Figures 4.7 and 4.8 simply plot the mean of that range. Nevertheless it is felt that these curves represent a useful approximation to the paging behavior of the tasks under consideration in the scheduling model, and so these figures will be used to determine the number of page faults during task execution when core allocation is reduced. The new number of page faults may then be used to get a new expectation for the number of CPU - I/O cycles before

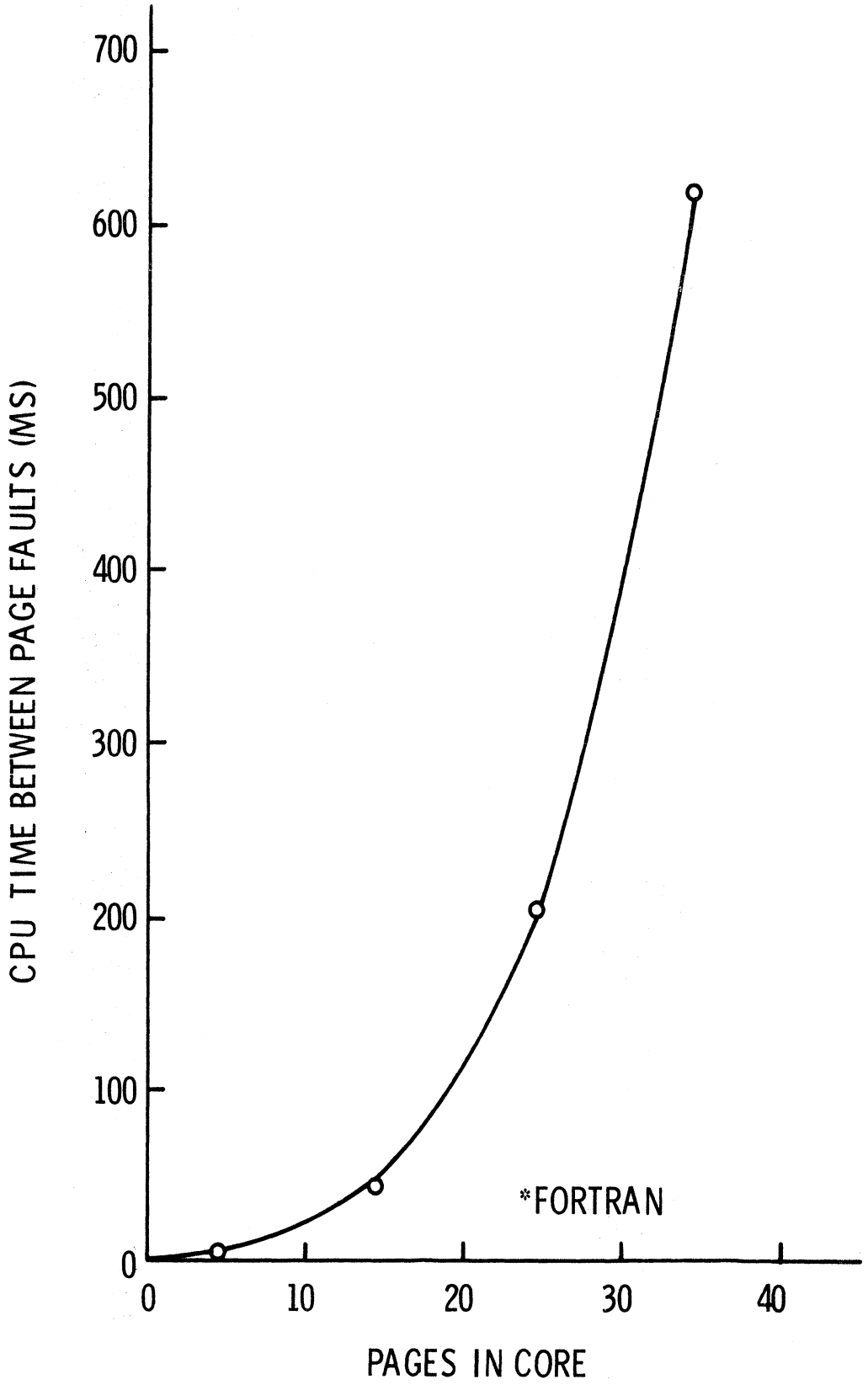

Figure 4.7. CPU Interval Between Page Faults As a Function of Mean Number Pages In Core During Interval

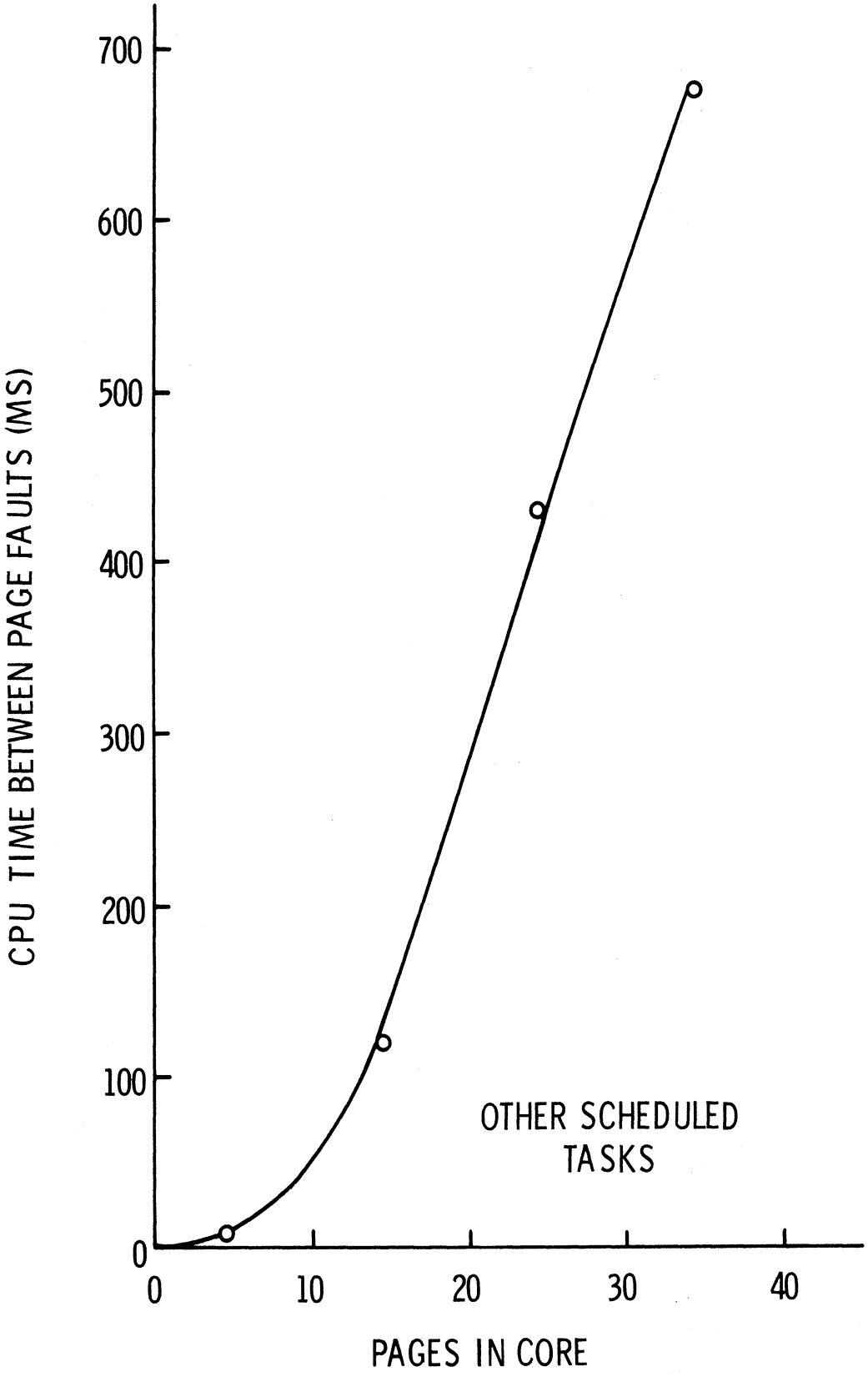

Figure 4.8. CPU Interval Between Page Faults As a Function of Mean Number Pages In Core During Interval

termination, and to get expectations for lengths of CPU intervals and I/O intervals. These are then used as new input parameters to the scheduling model (columns marked H in Table 4.1).

Suppose data for the core allocation for tasks as determined from the MTS data show an average x_1 page faults per execution. By definition of page fault there must have been a CPU interval before the first page fault and after the last one. These intervals will be ignored and we will concentrate on determination of the reduction of the length of the CPU intervals between the first and last page faults when the number of pages in core is reduced. Knowing the original core allocation per task, the CPU time, t_1, between page faults may be determined from Figure 4.7 or 4.8. The CPU time, t_2, between page faults when the core allocation per task is reduced may be determined similarly. The quantity to be determined is x_2, the number of page faults when the core allocation is reduced for each task. This new number of page faults is used to determine a new number of CPU - I/O cycles before task termination and also to change the length of the average I/O interval since page faults in general require less time than other types of input or output. The average CPU time used between the first and last page fault is $(x_1 - 1)t_1$ milliseconds as there is one less CPU interval between page faults than there are page faults. Over this period of CPU time there will be a page fault every t_2 milliseconds on average when the core allocation for each task is reduced. Therefore

$$x_2 = 1 + \frac{(x_1 - 1)t_1}{t_2}$$

where a one is added to account for the fact that a page fault occurs at both ends of the CPU intervals between page faults. This expression should be a reasonable approximation for the mean number of page faults per task execution under conditions of reduced core allocation providing we do not change the core allocation by too large an extent.

As an example, consider Fortran when the core allocation is reduced. From Figure 4.7 it is seen that with the original core allocation of 25 pages, $t_1 = 215$ milliseconds elapse between page faults on the average. The mean number of page faults per execution is $x_1 = 14.4$ (see Table D.2). With 22 pages in core, $t_2 = 145$ milliseconds of CPU time are used between page faults. Thus

$$x_2 = 1 + \frac{215\ (14.4 - 1)}{145} = 20.9$$

is the average number of page faults per run when Fortran tasks have 22 pages in core. This leads to a new mean number of CPU - I/O cycles which is the sum of the number of I/O requests per run (63.1 in Table D.2) and the number of page faults per run (20.9). The reciprocal of this sum is the termination probability in Table 4.1.

One other problem remains in determining parameter values for the case of reduced core allocation, and this is the problem of estimating the time required for a page fault. The increased rate at which page faults occur for each task as well as increased multiprogramming

could cause mean paging delays to increase because of increased congestion at the paging drum. For MTS, however, congestion at the drum channel was very low and remained low even with the increased paging so no appreciable error is introduced by assuming the mean paging delays remain constant. More specifically, the maximum drum busy fraction was determined to be less than .2 during the data collection period, and this goes up to a maximum of .27 when core allocation for all tasks is reduced by the amount given in Table 4.1. The change in drum busy fraction was derived by assuming that the pages of core allocated tasks not included in the scheduling model were also reduced, and that this reduction causes an increase in paging activity for these tasks which is the same proportionately as the increase in paging activity for scheduled tasks. Another factor which influences mean paging delay is the fact that there are 18 queues for the two drums and this further tends to mitigate the effect of increased activity on mean delay.

A comparison of bars 5 and 8 in Figure 4.6 shows that with MTS scheduling and no information sharing, throughput is increased about 20% simply by decreasing the number of pages allocated each task in order to increase the degree of multiprogramming. These results were obtained by assuming central processors which run at full rate and since all tasks, and not just scheduled tasks, may be run with reduced core allocation, the results may be considered indicative of

the increase in throughput for MTS as a whole if there were three central processors, no increase in total main memory, and the increase in degree of multiprogramming made possible by the given decrease in core allocation for each task. It is clear that if more CPU power were added to the system, the degree of multiprogramming and thereby throughput could be raised with the real memory currently available and without danger of thrashing, although just how much the increase in degree of multiprogramming should be to maximize throughput is not given by these results.

4.11 Further Results

The results presented up to this point were obtained from the scheduling model by use of input parameters which were extracted from data obtained from a multiprogrammed computer system. Results are presented in this section for several sets of hypothetical data. These data were chosen to illustrate how throughput may be expected to vary over a range of values of certain parameters. Presented first is throughput in a system where parameters were chosen for which it was felt optimal scheduling would result in an especially large improvement in throughput. Next results are presented in which memory size is varied over a wide range, and finally results in which a relatively large fraction of the pages are sharable. For all these results it is assumed that each arriving task is placed in one of two possible classifications.

The first system to be discussed is one in which arriving tasks may be classified either as compute-bound or as I/O bound. Some of the arriving tasks use, on the average, ten times as much CPU time as I/O time, while the others require ten times as much I/O time as CPU time. There is a single CPU in the system, and core memory is of such a size relative to the arriving tasks that it is always possible to multiprogram two tasks. That is, the maximum degree of multiprogramming is always two. Figure 4.9 shows the effect on throughput as the fraction of arriving tasks that are I/O bound is varied, both for optimal scheduling and single queue scheduling. Table 4.2 gives parameter values in the model for the results shown. The figure shows a maximum gain in throughput of about 8% when optimal scheduling is used over single queue scheduling. It should be borne in mind that the single queue policy is itself a good heuristic scheduling algorithm since it gives tasks arriving at the CPU from I/O priority over any task using the CPU. This tends to keep I/O bound tasks from getting locked up in the CPU queue. Thus it is expected that the increase in throughput under optimal scheduling would be even greater in systems using other simple scheduling algorithms such as FCFS at all queues. The figure also shows the increase in throughput with optimal scheduling going to zero when only one type of task is arriving at the system, because in these cases there are no scheduling decisions to be made other than the trivial one of loading a task whenever there is room in core.

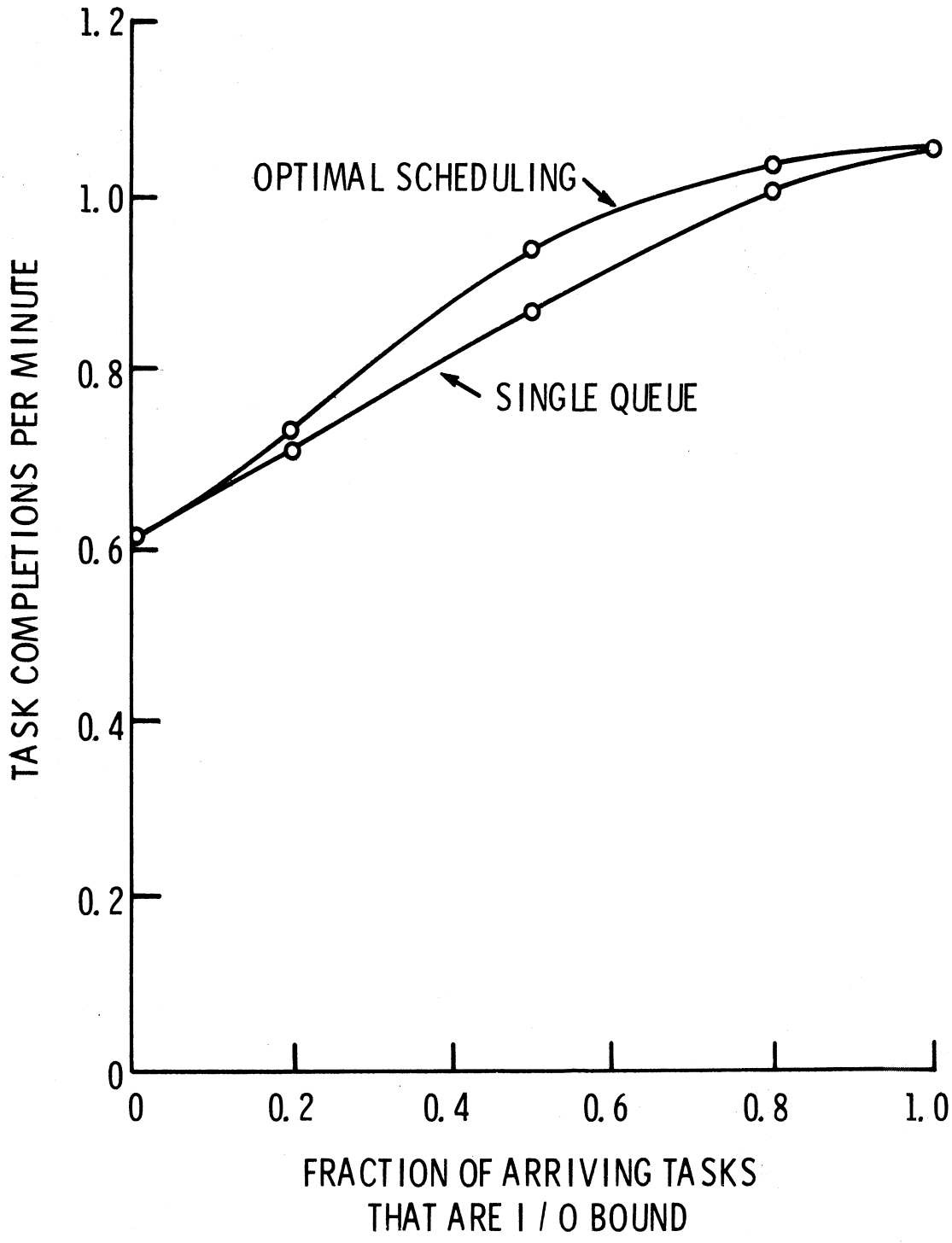

Figure 4.9. Variation of Throughput With Scheduling Policy and Relative Task Arrival Rates

Table 4.2

Parameters For Results in Figure 4.9

Task types	2	
Tasks	4	
Pages available	50	
CPU's	1	
Task Completion Rate Limited By	I/O	CPU
Arrival probability	Variable	
Termination probability	.01	.01
Mean CPU interval (sec)	0.1	1.0
Mean I/O interval	1.0	0.1
Non-sharable pages	25	25
Sharable pages	0	0
Loading time	1	1

Another parameter varied when the arrival probability for each task type was 0.5 was the time-slice. The plotted values are for a time slice of 100 seconds which is effectively no interruptions due to time slice runout since the average CPU interval is no longer than one second. A time slice of .01 second was also tried and the throughput increased by around 1%. Actual increase would be even less than this since the model does not account for the overhead involved in switching tasks at the CPU.

The optimal scheduling policy may be given as a table of scheduling decisions. This table shows the decision to be made in each

possible state of the system. For the results shown in Figure 4.9, an optimal policy may be stated in simpler terms. The optimal scheduling policy given below was abstracted from the decision table produced by the computer program which determined the optimal policy. This policy is

1) If core is empty, load a CPU bound task if any are in the memory queue. Otherwise load an I/O bound task.

2) Always give an I/O bound task preemptive priority at the CPU.

3) If there is one task in core and it is I/O bound, load a CPU bound task if possible. If there are no CPU bound tasks in the memory queue, load an I/O bound task.

4) If there is one task in core and it is CPU bound, load an I/O bound task if possible. If there are no I/O bound tasks in the memory queue, load another CPU bound task.

This is a good place to point out that sharing and optimal scheduling may operate at cross purposes to one another in the sense that the optimal scheduling policy with no sharing is to mix the task types in core while with sharing the degree of multiprogramming and thereby throughput may be increased by scheduling tasks of the same type together.

We now present results for a system inwhich the memory size was varied over a range wide enough to exhibit the maximum range in system throughput.

The values of the parameters for Figure 4.10 are given in Table 4.3, and the policies found to be optimal are presented in Appendix G. Here is a situation where all tasks being run on the system are CPU bound, and so throughput is limited by saturation of the CPU when the possible degree of multiprogramming as determined by the size of memory is still fairly low. The maximum throughput is shown and occurs at 100% CPU utilization. When the memory is so small that no multiprogramming is possible, throughput is the same for all cases. When memory is very large, throughput is limited by available

Table 4.3
Parameters For Results Of Figure 4.10

Task types	(M)	2
Tasks	(N)	5
CPUs		1

ITEM	TYPE 1	TYPE 2
Arrival probability	.5	.5
Termination probability	.01	.02
Mean CPU Interval (sec)	1.0	2.0
Mean I/O Interval (sec)	.111	.143
Non-sharable pages	20	15
Sharable pages	10	10
Loading time (sec)	1	1

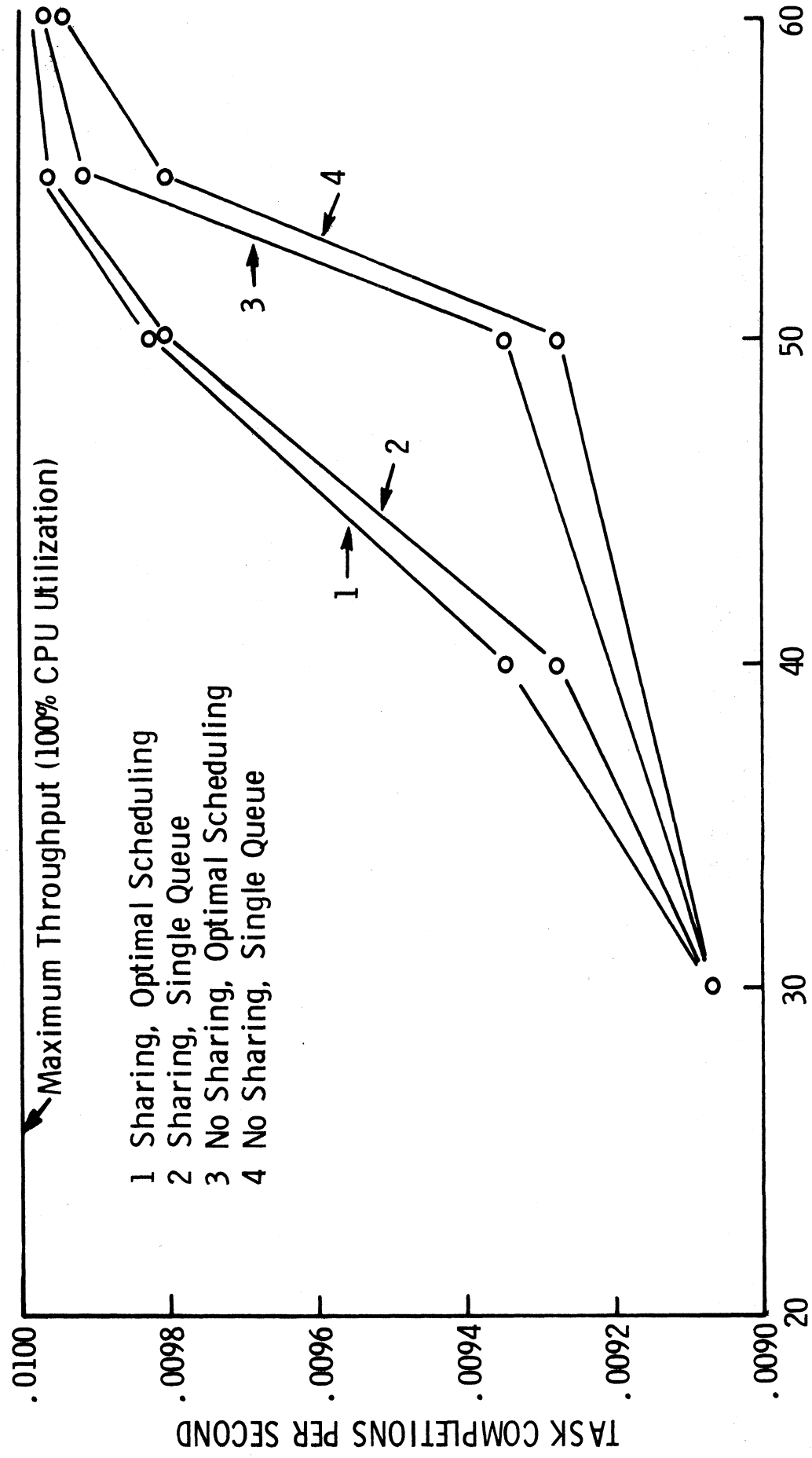

Figure 4.10. Comparison of Throughput

CPU time. It is only in the middle range where use of reentrant code and scheduling affect system throughput. Of course we are assuming here as elsewhere that the additional I/O delay experienced by tasks as the degree of multiprogramming increases is insignificant over the range of interest and thus throughput is ultimately limited by the CPU and not by bottlenecks at I/O devices. Since the CPU is generally the most expensive single system resource this assumption is reasonable for most properly designed general purpose computing systems. The behavior shown in Figure 4.10 would appear in Figures 4.5 and 4.11 if throughput had been computed for a wider range of memory sizes.

In Figure 4.11 throughput is shown for a system dedicated to running two types of tasks. Table 4.4 gives values of the parameters for these results and shows that the sharable fraction of the total information used by each type of task is substantial. The optimal scheduling policies are given in Appendix G. At 35 pages of core no multiprogramming is possible and so scheduling decisions and reentrant code do not affect throughput. At 60 pages the capability of sharing information is seen to have substantially more impact on throughput than optimal scheduling. This is due to the large fraction of sharable pages in the two types of tasks using the system. This means the degree of multiprogramming possible with sharing is substantially higher than that without sharing.

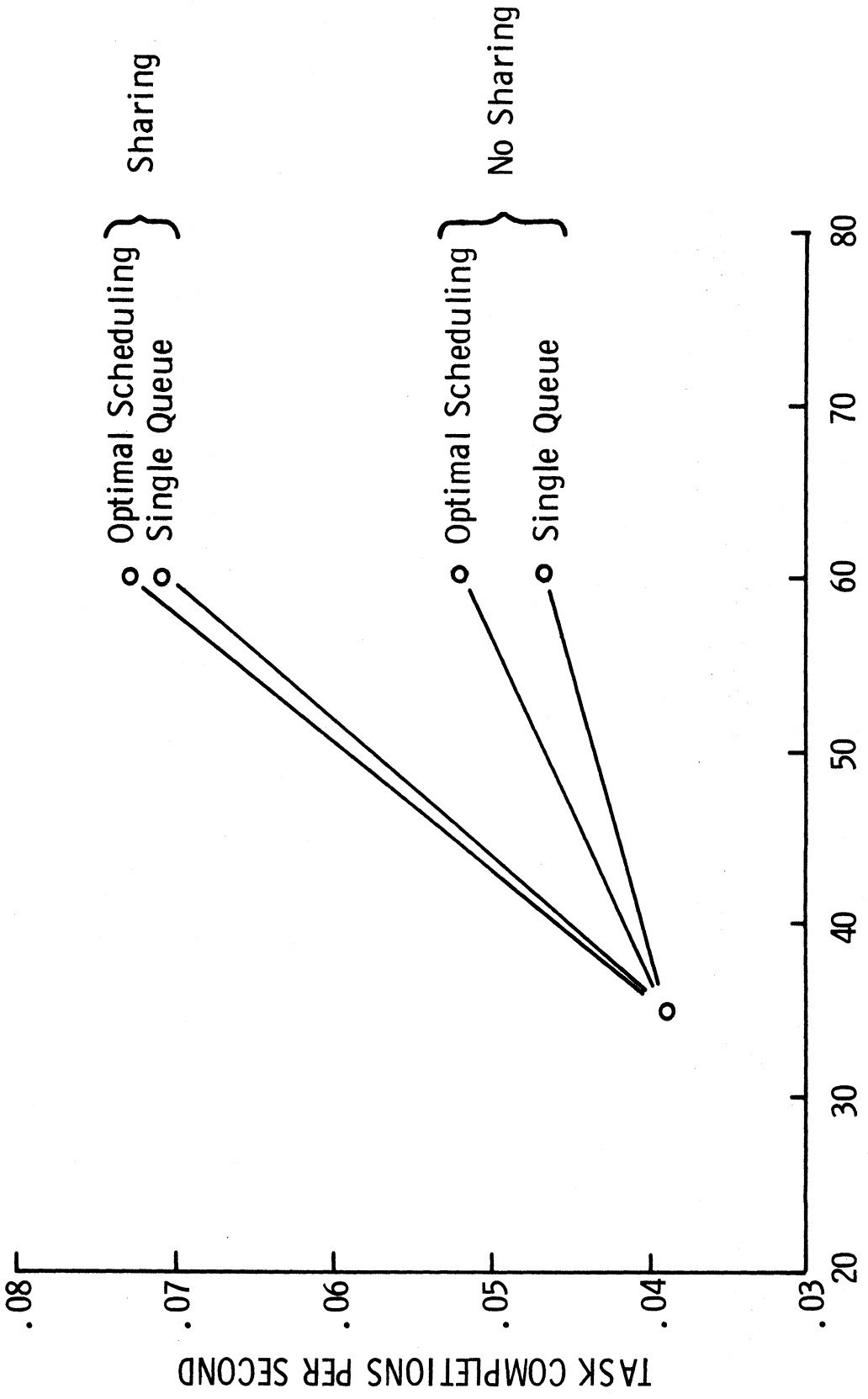

Figure 4.11. Comparison of Throughput

Table 4.4
Parameters For Results Of Figure 4.11

Task types	(M)	2
Tasks	(N)	5
Pages available	(P)	35,60
CPUs		1

ITEM	TYPE1	TYPE2
Arrival probability (b)	.5	.5
Termination probability (p)	.01	.01
Mean CPU Interval (sec)	.1	.08
Mean I/O Interval (sec)	.1	.2
Non-sharable pages	10	15
Sharable pages	25	15
Loading time (sec)	1	1

Parameters such as those given in Table 4.4 might arise in a small-to medium scale multiprogrammed computer system which is dedicated to querying sharable data bases and perhaps updating the data and carrying out some computation with the data. Users give commands which initiate execution of one of several possible programs which use one or the other of the two sharable data bases.

Chapter 5
CONCLUSION

The major efforts in this work have been directed toward derivation of a useful optimization method for Markov renewal decision processes (MRDP's), and toward development and application of a queuing model useful for determination of the effects of queue control rules on computer system efficiency. The queuing model is a Markov renewal process under any specific queue control rule and further may be formulated as a MRDP to find optimal control rules. Thus the MRDP optimization method derived in Chapter 2 is applicable to the computer system queueing model developed in Chapter 3, and results of this application are presented in Chapter 4. Ancillary to getting these results was collection and analysis of computer system data which provided concrete input parameters to the model and therefore results which have applicability to current multiprogrammed computer systems. Results of this data analysis are presented in both Chapter 4 and Appendix D.

The major theoretical development in this thesis is the optimization method for infinite time Markov renewal decision processes in Chapter 2. Both optimization per unit time and per stage are treated; and the method provides a practical computational technique for models with large numbers of states, especially when transition matrices are

sparse. The advantage of having to store and use only non-zero elements of these matrices can be very great. Furthermore, models of most physical systems do have sparse transition matrices, and so this method should have wide applicability both for models of computer systems and for models of other stochastic service systems which may be treated as MRDP's.

There are two endeavors which might provide interesting and useful extensions to the work presented in Chapter 2. The first has to do with the rate of convergence of the optimization algorithm. This could be an attempt to find out what factors determine or affect the rate of convergence and an attempt to influence these factors in order to speed convergence. No formal attempt was made to study this problem although experience with optimization of scheduling rules showed that seemingly small changes in some parameters affected the rate of convergence quite substantially in some cases. The second has to do with determination of conditions which assure convergence of the algorithm that are more general than those given. The conditions given are sufficient for convergence and probably hold for almost all models of physical systems, but these conditions are certainly not necessary. Of course, even if the conditions given are not met, the algorithm is still useful as long as convergence does in fact occur.

The model developed in Chapter 3 has rather general applicability to the problem of scheduling the two major resources of most computer

systems; main memory and the central processors. The goal is to schedule these resources in a way that maximizes system efficency. The complexity of this problem arises from having to schedule two separate resources which nonetheless are interdependent. Besides allowing comparison of scheduling rules for their effects on system efficiency, the model also allows investigation of the improvement in efficiency when information is sharable among tasks. In particular, in the development of this model, effort was concentrated on a formulation suitable for determining optimal scheduling policies and on formulation for a particular simple heuristic policy: FCFS at the memory and LCFS at the CPU's.

It should be pointed out that in addition to the effort required to develop and describe this scheduling model, a very substantial programming effort was required to obtain useful results. The programming required probably matches that in many simulation models of computer systems. The advantage of the model here over a simulation model is that optimization is possible and results are not dependent on the particular sequence of random numbers generated during the simulation. Of course, a disadvantage is the necessity of assuming certain "nice" probability distributions such as the exponential.

The results obtained from this mathematical model, in terms of system throughput, are presented as a function of memory size, scheduling rule, and whether or not sharing of information among

tasks is possible. Values for other parameters of the model were determined in some cases by analysis of data collected from a time-shared, multiprogrammed computer system. Effects of increasing the load on this system are shown as well as variation in throughput when the parameters given above are varied. Another result shows that scheduling and sharing of information are only important in an environment where there is enough memory to support some multiprogramming but not so much memory that full CPU utilization occurs simply due to the high degree of multiprogramming possible and independently of the scheduling policy used or sharing of information.

One conclusion to be drawn from the results shown and from experience with the model is that the gain to be derived from optimal scheduling and sharing information depends strongly on the computer system and on the tasks which use it. Potentially, this gain will be greatest in systems where the users are constrained to run a small number of programs which contain a large percentage of sharable procedure or data. An example might be a computer used to update and query some form of intelligence data. Users may be constrained to interrogate the data through a given set of programs, and the same data may be required by more than one user.

At this point it would be well to reiterate some assumptions made in the model which would not strictly hold in the system being modelled. Any additional supervisory overhead brought about by implementation of an optimal scheduling policy was ignored. It was not feasible to estimate this for purposes of the model, but should certainly be a factor in any decision to use such a scheduling policy.

In an environment where information is shared, an additional benefit of sharing may be the reduction in CPU time used by the loader due to the fact that only one copy of a task need be loaded for several users. On the other hand, this may be somewhat offset if there is additional supervisory overhead associated with the sharing of information. In the model CPU time used by the loader in moving a program into main memory in a form suitable for execution was ignored so that no cognizance was taken of the problem of scheduling the loader at the CPU or of savings in loader CPU time when information is shared. This assumption was considered reasonable in view of the fact that the loader is usually a program characterized by large amounts of input/output relative to the amount of CPU time required.

The results obtained by applying the optimization method developed in this work to an important computer system control problem should be useful to designers of such systems, while at the same time these results show that this optimization technique can profitably be used in the design of stochastic service systems.

Finally, a brief comment may be made on the data analysis carried out during this study. Computer system data are scarce and difficult to obtain, and so it is hoped that the data presented in this thesis will find applicablilty to computer system analysis efforts of others. For that reason, a substantial amount of data are presented which were not used directly in the analysis and optimization work in this thesis, but which it was felt would be of fairly general interest to other workers in the field.

Appendix A

EXTENT OF PAGE SHARING IN MAIN MEMORY

In this appendix we are concerned with a set of tasks in main memory which shares pages. For these tasks we assume a set of sharable pages common to all the tasks, but allow for the possibility that not all these pages are in main memory and further, that each of the pages which is in main memory is not in use by the entire set of tasks. Using simple assumptions, a functional relationship is obtained among the total number of sharable pages for a task, the number of those pages in core, and the number of those pages being used by each of the set of tasks in core.

Stated in another way, the goal is to obtain a reasonable approximation to the number of sharable pages each task in main memory is making use of where more than one task may be using the same page because of the possibility of tasks sharing pages. In the derivation of this approximation all tasks make potential use of the same set of sharable pages, some of which may not be in core. Define

w: the approximation to the number of sharable pages in use by each task.

q: the total number of sharable pages in memory for the tasks.

z: the number of tasks in memory which may share these pages.

c: the total number of sharable pages for these tasks.

W_m: the set of sharable pages in use by the mth task of the set of z tasks in memory.

$(m = 1, \ldots, z)$

$|W_m|$: the cardinality of W_m

We see that since sharing of pages among tasks is possible

$$q \leq zw$$

Also, of course, $q \leq c$

The assumptions we make in this derivation are that all tasks of the same type are statistically independent of one another and that all possible page sets of a given size have equal probability of being in use by a given task. These assumptions represent a worst case as far as the extent of page sharing among tasks is concerned. We also assume that

$$|W_m| = |W_n| \qquad \forall\ m, n$$

and of course it is just the cardinality of these sets for which we want an approximation. That is, our goal is a function w(q, z) to be used as an approximation to $|W_m|$ for all m.

The function will be determined separately for each value of z. To determine the function we begin by assuming a value for w. Under our assumptions we are then able to compute the probability that any given number of pages is used by m tasks (m= 1, . . . , z). Our approximation consists of determining the expectation of the amount of page sharing which occurs and using this expectation as the amount of actual

page sharing. This expectation gives the amount of overlap of the sets W_m and gives the desired relation between the total number of sharable pages in memory (q) and the number of pages each task is using (w).

The case z = 1 is trivial. Here

$$w(q, 1) = q; \qquad z = 1 .$$

Next suppose z = 2. Again we assume a value for w, the cardinality of W_1 and W_2. Choose any page, p, in W_1. Then

$$Pr\{p \in W_2 \mid p \in W_1\} = Pr\{p \in W_2\} = \frac{w}{c}$$

Let S_2 be the number of pages shared by both tasks. S_2 is a random variable which may take on values from 0 to w. The probability that the page p, chosen above is not shared is

$$Pr\{p \notin W_2 \mid p \in W_1\} = 1 - \frac{w}{c} .$$

From this and our prior assumptions:

$$Pr\{S_2 = 0\} = (1 - \frac{w}{c})^w$$

$$Pr\{S_2 = 1\} = w \frac{w}{c} (1 - \frac{w}{c})^{w - 1}$$

$$Pr\{S_2 = k\} = (\frac{w}{k}) (\frac{w}{c})^k (1 - \frac{w}{c})^{w-k}$$

The random variable S_2 thus has a binomial distribution and

$$E(S_2) = \frac{w^2}{c} .$$

Now $$q = 2w - S_2 .$$

We replace S_2 by its expectation to get

$$q = 2w - \frac{w^2}{c}$$

Then $$w(q,2) = c - \sqrt{c^2 - qc} \quad ; z = 2$$

Next let $z = 3$. Again we assume the cardinality of sets W_1, W_2, and W_3 is w. Again define S_2 as the number of pages shared by W_m and $W_n (m \neq n)$. Then as before we have

$$E(S_2) = \frac{w^2}{c}$$

It is also possible that all three tasks may share some pages. The number of pages so shared will be denoted by the random variable S_3. Then the relation

$$q = 3w - 3S_2 + S_3 \qquad \text{(A. 1)}$$

must hold. To get the approximation w in terms of q we replace S_2 and S_3 by their expected values in the same manner as before.

To get $E(S_3)$ we proceed as follows. Analogous to the case where $z = 2$ we pick a page $p \in W_1$ and determine the probability that this page is also in W_2 and W_3.

$$\Pr\{p \in W_2 \cap W_3 \mid p \in W_1\}$$

$$= \Pr\{p \in W_2, p \in W_3 \mid p \in W_1\}$$

$$= \Pr\{p \in W_3 \mid p \in W_1, p \in W_2\} \Pr\{p \in W_2 \mid p \in W_1\}$$

$$= \frac{w}{c} \times \frac{w}{c} = \frac{w^2}{c^2}$$

$$\Pr\{p \notin W_2 \cap W_3 \mid p \in W_1\} = 1 - \frac{w^2}{c^2}$$

From this we get

$$\Pr\{S_3 = 0\} = \left(1 - \frac{w^2}{c^2}\right)^w$$

$$\Pr\{S_3 = k\} = \binom{w}{k} \left(\frac{w^2}{c^2}\right)^k \left(1 - \frac{w^2}{c^2}\right)^{w-k}$$

so that

$$E(S_3) = w\frac{w^2}{c^2} = \frac{w^3}{c^2}$$

Replacing S_2 and S_3 by their expectations in Equation A.1 we get

$$q = 3w - 3\frac{w^2}{c} + \frac{w^3}{c^2}$$

$$w^3 - 3cw^2 + 3c^2 w - qc^2 = 0$$

This cubic equation in w may be solved to yield w(q, 3) for positive integer values of q. Because analytic solution for $z \geq 3$ is difficult, values of w as a function of q, z, and c are tabulated at the end of this appendix (Table A.I).

We may derive expressions for w for $z = 4, 5, \ldots$ individually for each z, but it seems more reasonable to try to get a general expression which holds for all z. To this end let us define a random variable S_n as the number of pages shared by all n of n tasks when each task is using w pages. We have already found S_2 and S_3 and of course $S_1 = w$.

Pick a page $p \in W_1$. Then

$$P\{p \in (W_2 \cap \ldots \cap W_n) \mid p \in W_1\}$$

$$= Pr\{p \in W_n \mid p \in (W_1 \cap \ldots \cap W_{n-1})\}\ P_r\{p \in W_{n-1} \mid p \in (W_1 \cap \ldots \cap W_{n-2})\} \ldots P_r\{p \in W_2 \mid p \in W_1\}$$

$$= \frac{w^{n-1}}{c^{n-1}}$$

$$Pr\{p \notin (W_2 \cap \ldots \cap W_n) \mid p \in W_1\} = 1 - \frac{w^{n-1}}{c^{n-1}}$$

$$Pr\{S_n = k\} = \binom{w}{k} \left(\frac{w^{n-1}}{c^{n-1}}\right)^k \left(1 - \frac{w^{n-1}}{c^{n-1}}\right)^{w-k}$$

$$E(S_n) = \frac{w^n}{c^{n-1}}$$

In what follows we replace S_n by its expectation so that each of two different groups of n tasks may be assumed to share $E(S_n)$ pages among all n of the tasks. We now obtain a relation among w, q, and z.

This will be a polynomial of degree z in w. The polynomial may be solved for w and is tabulated in Table A.I as mentioned earlier.

We build up f(w, z) iteratively. First let

$$q = zw + f_2(w, z)$$

Here we have added the w pages of each of the z tasks. f_2 accounts for the possible intersections. Next take all possible groups of two tasks and note that for each of these we have added in a number of pages equal to $E(S_2)$ twice. Thus

$$q = zw - \binom{z}{2} E(S_2) + f_3(w, z)$$

Now take all possible groups of three tasks and we see that in each case we have added the pages shared by all three tasks three times and then subtracted these pages three times. Therefore

$$q = zw - \binom{z}{2} E(S_2) + \binom{z}{3} E(S_3) + f_4(w, z)$$

Next take all possible groups of four and note that for each we have added the pages shared by all four, four times by adding the pages of each task individually. We then subtracted these same pages six, or $\binom{4}{2}$ times when subtracting all possible pairs; and finally added the pages again four times, or $\binom{4}{3}$ times when adding the pages shared by three tasks. Then in order to add these pages in exactly once, we must subtract them from each group of four tasks. So

$$q = zw - \binom{z}{2} E(S_2) + \binom{z}{3} E(S_3) - \binom{z}{4} E(S_4) + f_5(w, z).$$

We continue forming larger groups until we have included all z tasks in a single group. Thus

$$q = zw - \binom{z}{2} E(S_2) + \ldots - (-1)^n \binom{z}{n} E(S_n) + \ldots - (-1)^z E(S_z)$$

$$q = zw - \binom{z}{2} \frac{w^2}{c} + \ldots - (-1)^n \binom{z}{n} \frac{w^n}{c^{n-1}} + \ldots - (-1)^z \frac{w^z}{c^{z-1}}$$

As stated above, this is a polynomial equation of degree z in w and may be solved numerically. Solutions are tabulated for $1 \leq z \leq 10$ and for $1 \leq c \leq 10$ below.

Table A. 1
Pages Used By Single Task

		I	Z=									
C	Q	I	1	2	3	4	5	6	7	8	9	10
1	1	I	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1	I	1.00	0.59	0.41	0.32	0.26	0.22	0.19	0.17	0.15	0.13
	2	I	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
3	1	I	1.00	0.55	0.38	0.29	0.23	0.20	0.17	0.15	0.13	0.12
	2	I	2.00	1.27	0.92	0.72	0.59	0.50	0.44	0.38	0.34	0.31
	3	I	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00
4	1	I	1.00	0.54	0.37	0.28	**0.23**	0.19	0.16	0.14	0.13	0.11
	2	I	2.00	1.17	0.83	0.64	0.52	0.44	0.38	0.33	0.30	0.27
	3	I	3.00	2.00	1.48	1.17	0.97	0.83	0.72	0.64	0.57	0.52
	4	I	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
5	1	I	1.00	0.53	0.36	0.27	0.22	0.18	0.16	0.14	0.12	0.11
	2	I	2.00	1.13	0.78	0.60	0.49	0.41	0.35	0.31	0.28	0.25
	3	I	3.00	1.84	1.32	1.02	0.84	0.71	0.61	0.54	0.48	0.44
	4	I	4.00	2.76	2.08	1.66	1.38	1.18	1.03	0.91	0.82	0.74
	5	I	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
6	1	I	1.00	0.52	0.35	0.27	0.21	0.18	0.15	0.14	0.12	0.11
	2	I	2.00	1.10	0.76	0.58	0.47	0.39	0.34	0.30	0.26	0.24
	3	I	3.00	1.76	1.24	0.95	0.78	0.65	0.57	0.50	0.44	0.40
	4	I	4.00	2.54	1.84	1.44	1.18	1.00	0.87	0.77	0.69	0.62
	5	I	5.00	3.55	2.70	2.17	1.81	1.55	1.35	1.20	1.08	0.98
	6	I	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00

Table A. 1 Continued

7	1	I	1.00	0.52	0.35	0.26	0.21	0.18	0.15	0.13	0.12	0.11
	2	I	2.00	1.08	0.74	0.56	0.46	0.38	0.33	0.29	0.26	0.23
	3	I	3.00	1.71	1.19	0.91	0.74	0.62	0.54	0.47	0.42	0.38
	4	I	4.00	2.42	1.72	1.34	1.09	0.92	0.80	0.70	0.63	0.57
	5	I	5.00	3.26	2.39	1.88	1.55	1.32	1.15	1.01	0.91	0.82
	6	I	6.00	4.35	3.34	2.70	2.26	1.94	1.70	1.51	1.36	1.24
	7	I	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00
8	1	I	1.00	0.52	0.35	0.26	0.21	0.18	0.15	0.13	0.12	0.11
	2	I	2.00	1.07	0.73	0.56	0.45	0.37	0.32	0.28	0.25	0.23
	3	I	3.00	1.68	1.16	0.89	0.72	0.60	0.52	0.46	0.41	0.37
	4	I	4.00	2.34	1.65	1.27	1.04	0.87	0.75	0.66	0.59	0.54
	5	I	5.00	3.10	2.23	1.74	1.42	1.21	1.05	0.92	0.83	0.75
	6	I	6.00	4.00	2.96	2.34	1.94	1.65	1.44	1.27	1.14	1.04
	7	I	7.00	5.17	4.00	3.24	2.72	2.34	2.05	1.83	1.65	1.50
	8	I	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00
9	1	I	1.00	0.51	0.35	0.26	0.21	0.17	0.15	0.13	0.12	0.11
	2	I	2.00	1.06	0.72	0.55	0.44	0.37	0.32	0.28	0.25	0.22
	3	I	3.00	1.65	1.14	0.87	0.70	0.59	0.51	0.44	0.40	0.36
	4	I	4.00	2.29	1.60	1.23	1.00	0.84	0.72	0.64	0.57	0.51
	5	I	5.00	3.00	2.13	1.65	1.35	1.14	0.98	0.87	0.78	0.70
	6	I	6.00	3.80	2.76	2.16	1.78	1.51	1.31	1.15	1.03	0.94
	7	I	7.00	4.76	3.55	2.82	2.34	2.00	1.74	1.54	1.39	1.26
	8	I	8.00	6.00	4.67	3.80	3.20	2.76	2.42	2.16	1.95	1.78
	9	I	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00

Table A. 1 Continued

10	1	I	1.00	0.51	0.35	0.26	0.21	0.17	0.15	0.13	0.12	0.10
	2	I	2.00	1.06	0.72	0.54	0.44	0.37	0.31	0.28	0.24	0.22
	3	I	3.00	1.63	1.12	0.85	0.69	0.58	0.50	0.44	0.39	0.35
	4	I	4.00	2.25	1.57	1.20	0.97	0.82	0.70	0.62	0.55	0.50
	5	I	5.00	2.93	2.06	1.59	1.29	1.09	0.94	0.83	0.74	0.67
	6	I	6.00	3.68	2.63	2.05	1.67	1.42	1.23	1.08	0.97	0.88
	7	I	7.00	4.52	3.31	2.60	2.14	1.82	1.58	1.40	1.25	1.13
	8	I	8.00	5.53	4.15	3.31	2.75	2.35	2.05	1.82	1.64	1.49
	9	I	9.00	6.84	5.36	4.38	3.69	3.19	2.80	2.50	2.26	2.06
	10	I	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00

Appendix B

COMPUTATION OF e^A

We consider here the problem of computing the matrix e^A in an efficient manner when the N x N matrix A is given. Much of what follows holds for any matrix A although the particular problem of interest is for A a transition intensity matrix.

An efficient method of computation of e^A is very useful because this matrix provides a solution to the general system of homogeneous linear differential equations of first order with constant coefficients. These equations are

$$\dot{z}(t) = z(t)\, C$$

where z(t) is an N-vector and C an N x N matrix. Then the solution to these equations may be shown to be

$$z(t) = z(0)\, e^{Ct}$$

In what follows, Ct may be substituted for matrix A without difficulty.

We may write

$$S = e^A = M + R$$

where

$$M = \sum_{k=0}^{K} \frac{A^k}{k!}$$

$$R = \sum_{k=K+1}^{\infty} \frac{A^k}{k!}$$

We use M as the approximating matrix for e^A and therefore are interested in determining how large K must be to give us an acceptably small error. Liou [33] proves

Theorem B. 1

If $||A|| < K + 2$, then for all elements r_{ij} of R,

$$|r_{ij}| \leq \epsilon = \frac{||A||^{K+1}}{(K+1)!} \cdot \frac{1}{1 - \frac{||A||}{K+2}} \qquad \text{(B. 1)}$$

where $||A|| = \sum_{i,j=1}^{N} |a_{ij}|$

This theorem gives an easily computed upper bound on all elements of the remainder matrix.

Proof

It can be shown that

$$||A^k|| \leq ||A||^k, \quad k = 1, 2, \ldots$$

Hence each element of the matrix A^k is less than or equal to $||A||^k$. It follows that

$$|r_{ij}| \leq \sum_{k=K+1}^{\infty} \frac{||A||^k}{k!}$$

$$|r_{ij}| \leq \frac{||A||^{K+1}}{(K+1)!} \left[1 + \frac{||A||}{K+2} + \frac{||A||^2}{(K+3)(K+2)} + \ldots\right]$$

$$\leq \frac{||A||^{K+1}}{(K+1)!} \left[1 + \frac{||A||}{K+2} + \frac{||A||^2}{(K+2)^2} + \ldots\right]$$

If we choose K so that $||A|| < K + 2$, then

$$|r_{ij}| \leq \frac{||A||^{K+1}}{(K+1)!} \cdot \frac{1}{1 - \frac{||A||}{K+2}} = \epsilon$$

We may now choose a maximum acceptable error, ϵ_o, and use this in Equation (B. 1) to determine the number of terms, K, needed in an approximating series for e^A. This in itself may be satisfactory, but an N x N matrix multiplication is required for every additional term in the series and so for large K the computation required may be excessive. Figure B. 1 shows the variation in ϵ with K for several values of $||A||$. When A is a transition intensity matrix, the matrix S is a stochastic matrix with all row sums equal to one and so we certainly want $\epsilon < 1$. It is clear from this figure that the number of matrix multiplications necessary to achieve a given ϵ is strongly dependent on $||A||$.

The strong dependence of the number of required terms on $||A||$ suggests we try to use a related matrix with smaller norm. In particular, define the matrix

$$B = \frac{A}{2^b} \quad \text{for some } b = 1, 2, \ldots$$

We have $$||B|| = \frac{1}{2^b} \cdot ||A||$$

and define $$Q = e^B = X + Y$$

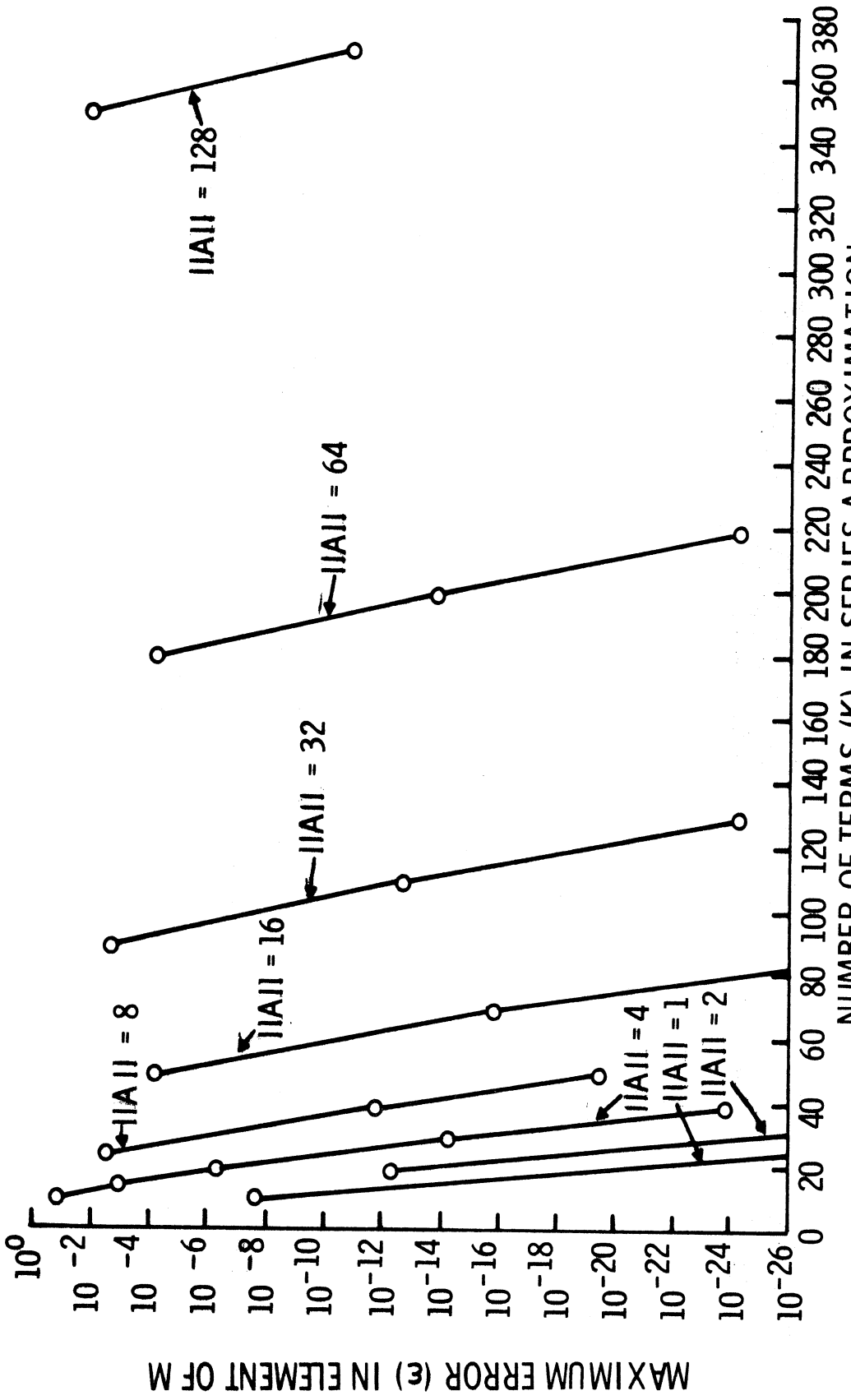

Figure B. 1 Maximum Possible Error in any Element of Series Approximation to e^A

where X is the approximating series and Y is the remainder series as defined earlier for A using M and R. With matrix B we may compute X and then with b additional matrix multiplications recover an approximation to S. That is,

$$S \approx X^{2b}$$

Suppose ϵ_o is the maximum acceptable error in each element of the approximation to the matrix S. If Q were computed exactly then S could be determined with no error (except round-off error). But we use the approximation X for Q and so there is an error which we may call ϵ_b associated with this computation. This error ϵ_b may be propagated and grow as we take successive powers of X. Therefore ϵ_b must be chosen to insure that this accumulated error is less than ϵ_o when X is raised to the 2^b power.

In order to find an ϵ_b (in terms of ϵ_o) which assures that the allowable error ϵ_o is not exceeded, assume that the maximum possible error occurs with the same sign in all elements of X. That is,

$$Q = X + [1]\epsilon_b \text{ where } [1] \text{ is an NxN matrix of ones.}$$

Then

$$Q^2 = (X + [1]\epsilon_b)^2$$

$$q_{ik}^{(2)} = \sum_{j=1}^{N} (x_{ij} + \epsilon_b)(x_{jk} + \epsilon_b)$$

$$= \sum_{j=1}^{N} x_{ij}x_{jk} + (x_{ij} + x_{jk})\epsilon_b + \epsilon_b^2$$

Now $$\sum_{j=1}^{N} x_{ij} = 1$$

$$\sum_{j=1}^{N} x_{jk} \leq N$$

Therefore the maximum error in Q^2 is

$$\epsilon_{b-1} \approx (N+1)\,\epsilon_b$$

where we have assumed $\epsilon_b << N + 1$.

This error may actually be shown to be smaller by noting that the row sums of the actual error terms are equal to zero because row sums of any power of a transition intensity matrix are zero. Note also that this means the row sums of X are always equal to one.

We may now square the approximation to Q^2 and again assume the maximum error, ϵ_{b-1}, occurs in every element of the approximating matrix. We get

$$\epsilon_{b-2} = (N+1)\,\epsilon_{b-1} = (N+1)^2\,\epsilon_b$$

Continuing this process we get

$$\epsilon_o = (N+1)^b\,\epsilon_b$$

Therefore if we choose

$$\epsilon_b \leq \frac{\epsilon_o}{(N+1)^b}, \qquad \text{(B. 2)}$$

we are assured that no element of $S - X^{2b}$ will exceed ϵ_o in absolute value.

Using Equations (B. 1) and (B. 2) it is possible to choose a value of b which results in the minimum number of matrix multiplications to obtain an approximation to S with maximum error ϵ_o. In actual practice however, a rough estimate is sufficient. As an example suppose

$$||A|| = 128$$

$$\epsilon_o = 10^{-6}$$

$$N = 99$$

The number of matrix multiplications required as a function of $||B||$ is shown in Figure B. 2. This number includes the b multiplications needed to get e^A from e^B. An actual minimum does not appear in Figure B. 2 although it is clear that we are very close to the minimum and that little additional computational savings can be realized by increasing the value of b further. A very pertinent fact about this figure is that it is not very sensitive to changes in N or ϵ_o. This suggests using a rule of thumb where we choose b to get a matrix B with norm somewhere around 1.

One final note may be made about the computational cost as we increase the number of states in the transition matrix. Usually as the number of states increases, $||A||$ increases also since there are more terms in the sum which determine this quantity. And of course, the number of actual multiplications for a matrix multiplication goes up as N^3.

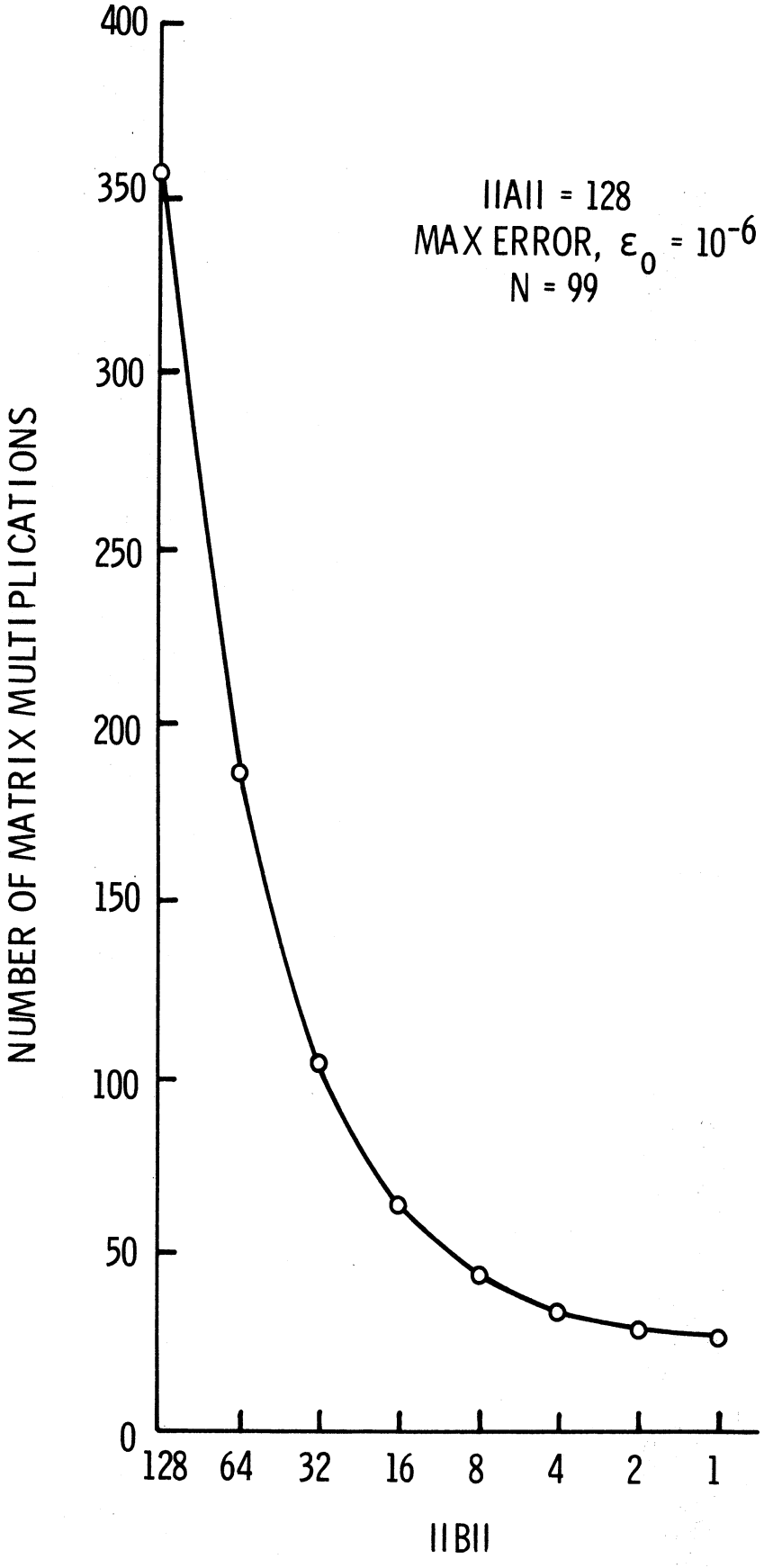

Figure B. 2 Matrix Multiplications to Compute e^A Using $\sum B^k/k!$

Appendix C

DISPERSION OF MEMORY REQUIREMENTS WITH DEGREE OF MULTIPROGRAMMING

This appendix contains a brief discussion of the advantage which may accrue in terms of more efficient use of memory as the memory size is increased to allow more programs to share core simultaneously. No consideration is given to changes in other facets of the system needed to maintain balance as the memory size is increased (such as increased CPU power). Changes in other system functions are assumed to keep step with changes in the memory size.

In a monoprogrammed computer system we may want assurance that some fraction $x(x > \frac{1}{2})$ of all the jobs which use the system will fit entirely within core memory. This requirement implies that we must supply some excess memory, z_1, over the average memory requirement, m_1, for a job. If we assume that system cost and capacity are proportional to the memory size then the fractional increase in cost over the average memory needed due to provision of memory z_1 is

$$f_1 = \frac{z_1}{m_1}$$

If we increase the memory size in order to be able to multiprogram two programs we must provide excess memory z_2 in order to be assured that the same fraction x of the sets of programs we wish to multiprogram

will fit. Our fractional cost increase for memory in excess of our mean memory requirement is now

$$f_2 = \frac{z_2}{m_2} = \frac{z_2}{2m_1}$$

We wish to show that $f_2 < f_1$ and in general that $f_{n+1} < f_n$ which says that as memory size is increased to allow a greater degree of multiprogramming, the excess memory which must be provided per task to assure that an acceptable fraction of the sets of programs we wish to multiprogram will fit decreases and thus we use memory more efficiently.

One way to see this is to look at the coefficient of variation of memory requirements as we increase the degree of multiprogramming. If tasks have mean size m_1 and standard deviation s_1, then the coefficient of variation of memory requirements in a monoprogrammed system is

$$c_1 = \frac{s_1}{m_1} \quad .$$

This is really an expression of dispersion of memory required as a fraction of mean memory requirements of tasks. Assuming that tasks are statistically independent the mean (m_n) and standard deviation (s_n) of memory required when n tasks are multiprogrammed is just

$$m_n = nm_1$$

$$s_n = \sqrt{n}\ s_1$$

Thus

$$c_n = \frac{s_n}{m_n} = \frac{s_1}{\sqrt{n}\ m_1}$$

and we see that the coefficient of variation decreases as the degree of multiprogramming increases.

Let us take a concrete example. Suppose memory size for tasks is normally distributed such that

$$c_1 = \frac{s_1}{m_1} = \frac{1}{3}$$

We wish to determine the fraction of memory in excess of the mean requirement as we vary the degree of multiprogramming. Figure C. 1 shows this for the cases where we want assurance that 75% and 90% of the sets of tasks we multiprogram will fit in core.

As another example, suppose the mean task size is 20 pages and task sizes are uniformly distributed from 16 to 24 pages. We decide to provide memory 10% in excess of our mean requirements in order to handle some of the programs entirely within core which are larger than the mean program size. Then if we desire to multiprogram 1, 2, 3, or 4 tasks; we provide 22, 44, 66 and 88 pages respectively. Figure C. 2 gives the probability that a set of tasks will not fit in core as a function of the degree of multiprogramming. It is seen that the probability that tasks will not fit is substantially smaller with four tasks multiprogrammed than in a monoprogrammed configuration. Looked at in another way, this is a strong argument for having flexible partitions in a multiprogrammed computer system rather than partitioning core into fixed, equal size blocks.

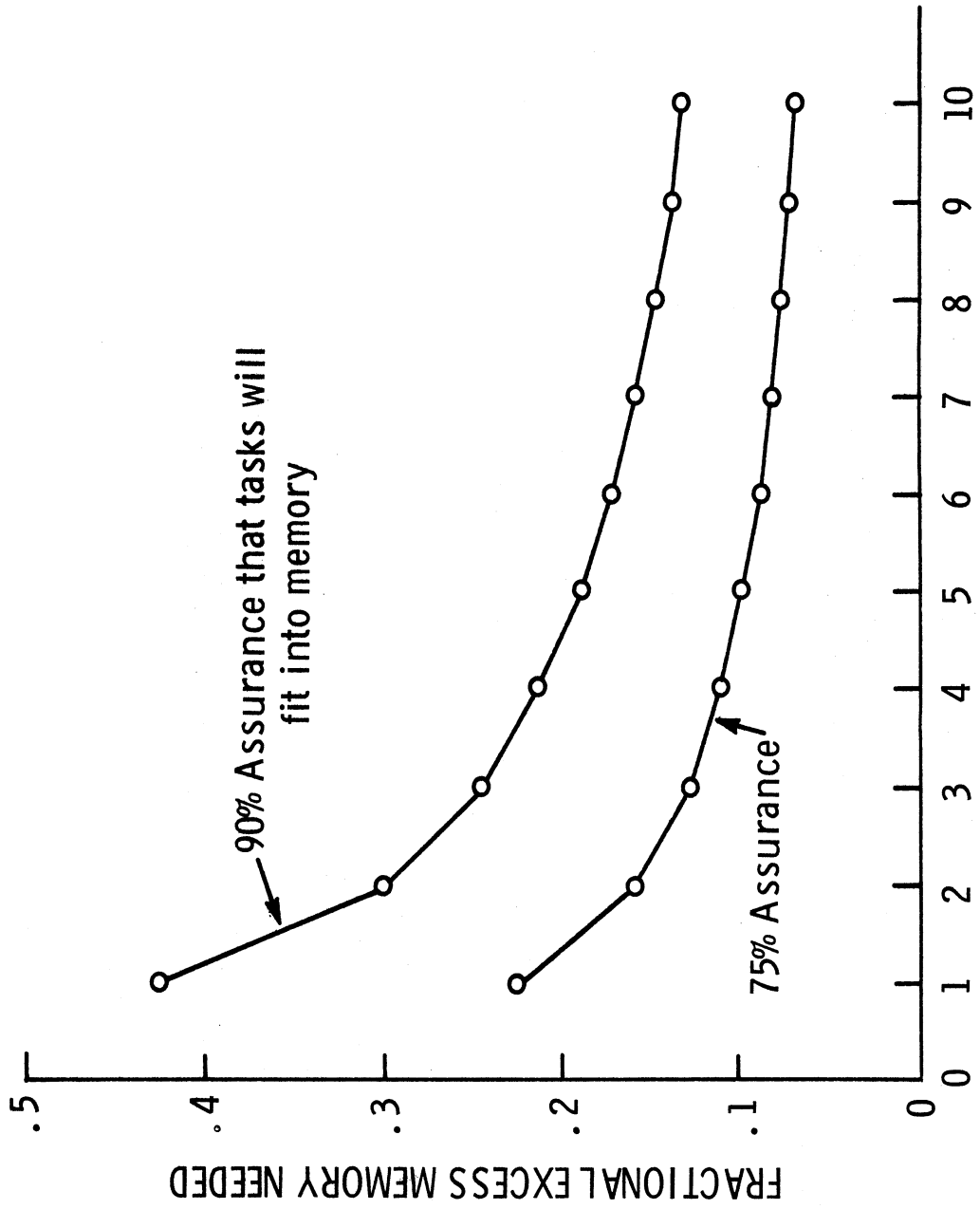

Figure C.1 Fraction of Memory Required in Excess of Mean Requirements

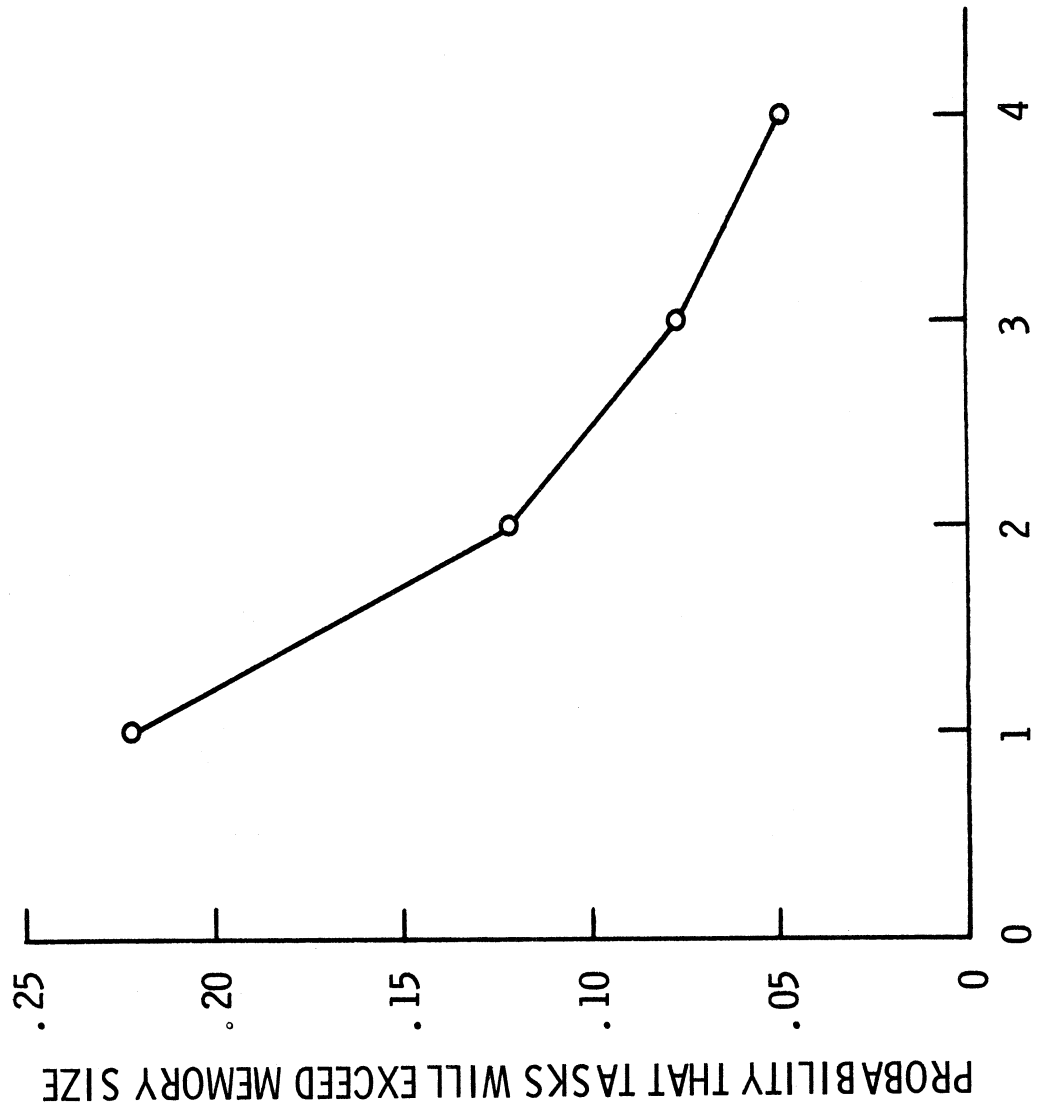

Figure C.2 Memory 10% in Excess of Mean Requirement

Appendix D

DATA COLLECTION AND ANALYSIS

This appendix gives the background for the data collection and analysis carried out on the multiprogrammed, time-shared system at The University of Michigan and results from that effort.

D.1 System Description

The computing system consists of an IBM System/360 Model 67 with two central processing units and 384 pages of core storage (1024 words per page). This is a virtual memory machine operated in a page-on-demand mode with paging to two drums (and possibly disks). The system runs under UMMPS (University of Michigan Multi-Programming System) which is described in more detail by Alexander [1] and Pinkerton [41]. In addition to the supervisor there exists a reentrant program, MTS (Michigan Terminal System) [54], which controls remote terminal users and the batch streams.

Code has been added to the system which allows data collection to be initiated at the operator's console. The specific items which may be collected are described by Pinkerton [39,41]. Events which occur along with the time they occur are placed in buffers which are written onto tape when they are full. For example, each time the CPU changes hands, the time is recorded along with the identification

of the job relinquishing the CPU and the job receiving the CPU. Pinkerton has shown that the process of collecting data disturbs the system operation a negligible amount so no cognizance need be taken of this when interpreting results.

D.2 General Data

The data presented here were collected October 10, 1970 about 3:00 p.m. This time was chosen in order to observe the system with as heavy a load as possible. During the data collection period (approximately 22 minutes) there were, on the average, about 50 terminal users and two batch streams active. The CPU time used may be placed in four broad categories. MTS batch and MTS terminal CPU time consists of time used by jobs in the batch stream and terminal users, respectively. Non-MTS CPU time is time used by system routines which cannot reasonably be charged to a particular job. Finally a CPU may be idle. The number of seconds spent by the central processors in each of these four categories is shown in Table D.1.

Table D.1

CPU Use During Data Collection (seconds)

Non-MTS	601
MTS batch	341
MTS Terminal	1571
Idle	129

During the data collection period the two CPU's were switched from one job to another a total of 470321 times. Thus the average length of time a job held a CPU was 5.7 ms.

D.3 Analysis for Specific Types of Tasks

In this and following sections results of analysis of the collected data are presented. The data collection facility records a large amount of detailed information on the status of each job in the system, but a substantial amount of effort is required for analysis of these data in order to obtain results which are useful and interesting. The main results presented here consist of presentation of the running characteristics of each of several types of tasks in the system (e.g. an assembler).

Figure D.1 shows the cumulative distribution for the elapsed time between occurrence of a page fault and satisfaction of the page request. The particular curve shown here is for page faults during runs of a Fortran compiler, but the distribution is independent of the type of task.

In the following sections we present results for each of several types of tasks using the system. By task we mean a program which is loaded and run from a terminal or batch stream. We give the elapsed time required to load the task. The remainder of the data presented give characteristics of the tasks from the

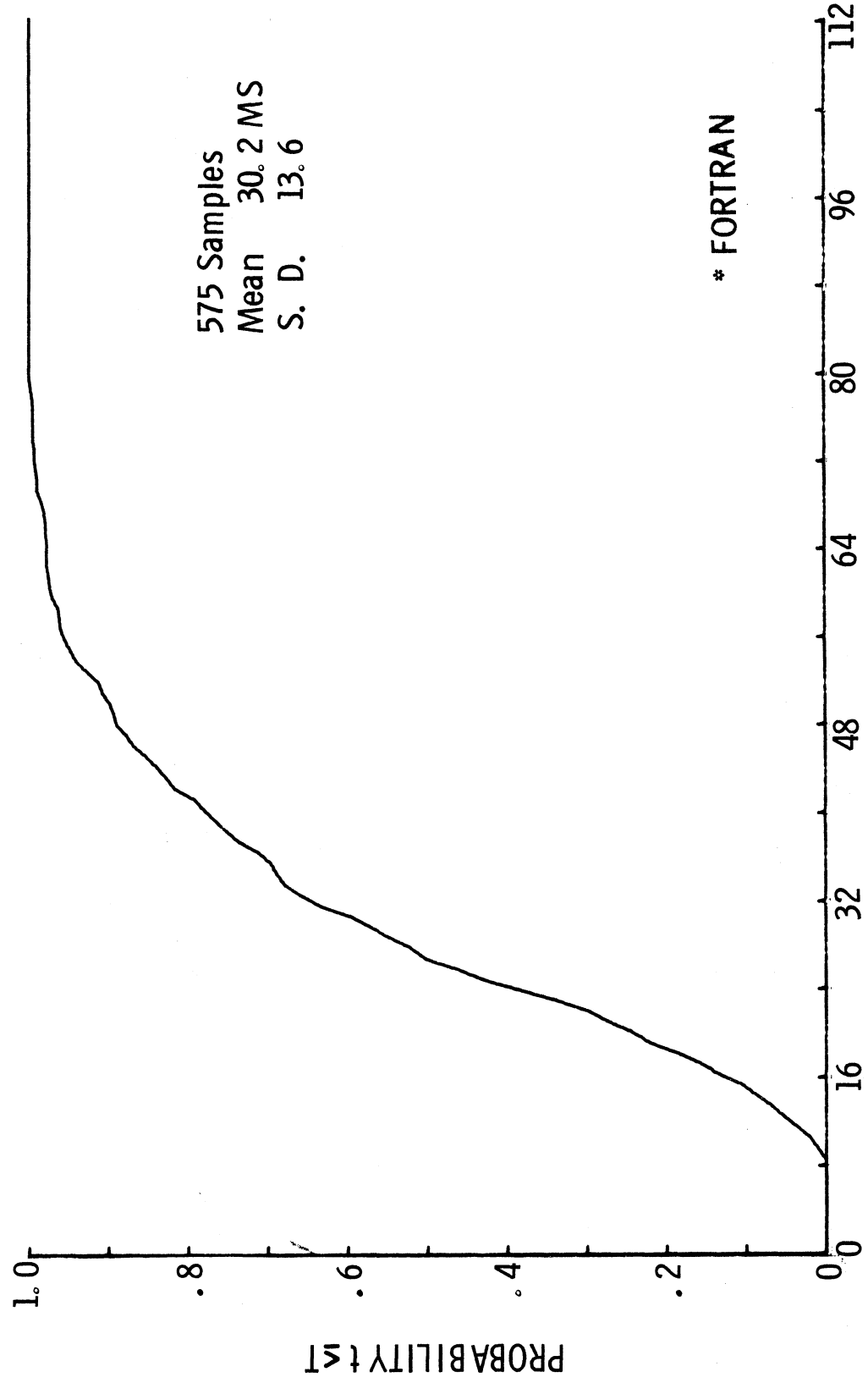

Figure D. 1. Cumulative Distribution of Time (t) From Page Demand Until Page Available

time loading is completed until execution of the task terminates. For each type of task a table presents means and standard deviations for elapsed execution time, CPU time, I/O time, and page wait time. The time the tasks are eligible to use the CPU (ready) may be obtained by subtracting CPU time, I/O time and page wait time from the elapsed running time. The mean number of I/O operations and the mean number of page waits per run are also given in this table.

In addition the table shows virtual memory and real memory assigned to tasks during execution. The virtual memory consists of the memory assigned the executing task plus three pages of tables used by the system to keep track of the job status.

Cumulative distributions are given for the length of time the task type uses the CPU between I/O requests and for the length of time the task type uses the CPU between page faults. Also distributions are given for the time required to carry out requested I/O operations (time from I/O request until request satisfied and task again eligible to use a CPU).

In MTS, programs intended for public use (such as compilers and assemblers) are placed in files and given names preceded by an asterisk. Data analysis was carried out for each such identifiable public task run during the data collection period. In addition all non-public tasks were grouped together for analysis purposes and

are called user generated tasks. Many tasks other than the ones presented in this appendix were run during the data collection period. The ones chosen for presentation were picked because it was felt there would be general interest in the results and because of the high number of requests for these tasks.

D.4 Fortran Compiler (IBM Lelvel G)

This compiler is by far the most heavily used program run by users of MTS. It is the IBM Fortran IV (G) compiler with a modified interface to allow it to run with the operating system (UMMPS) used at The University of Michigan. There were 41 requests (loads) for this compiler during the 22 minute data collection period. The mean elapsed loading time for those 41 samples was 5.06 seconds with a standard deviation of 1.75. Data obtained during completed executions of this compiler are presented in Table D.2.

In Figure D.2 we show the distribution of lengths of CPU intervals between input-output requests for the compiler. In addition two theoretical distributions with the same mean as the empirical distribution are shown. It is clear that the hyperexponential distribution fits the data far better than the exponential distribution. Figure D.3 gives the distribution of lengths of I/O operations during Fortran compilations and again we see that a hyperexponential distribution fits the data far better than an exponential distribution.

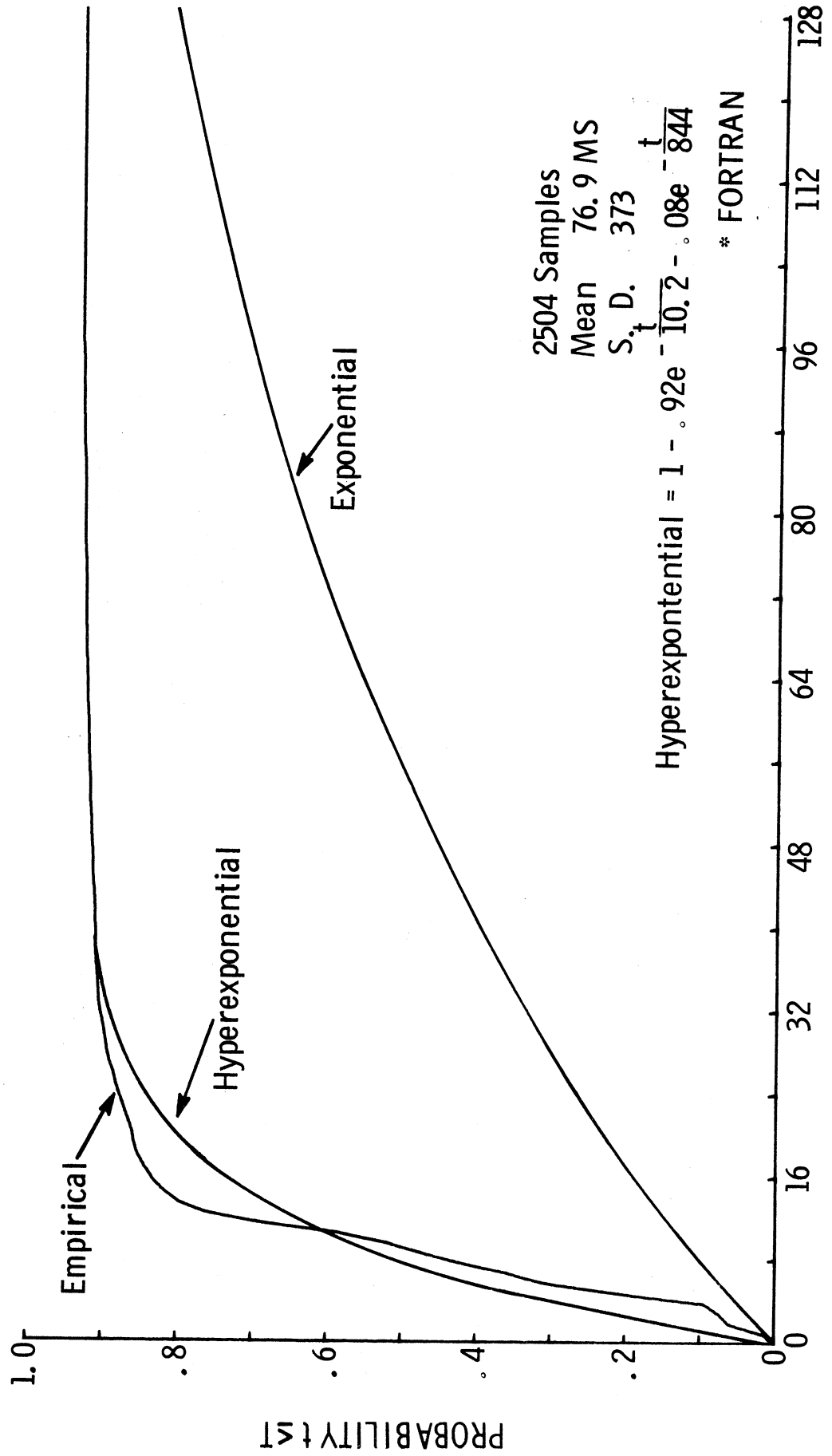

Figure D.2. Cumulative Distribution of CPU Time (t) Used Between I/O Operations During Fortran Compilations

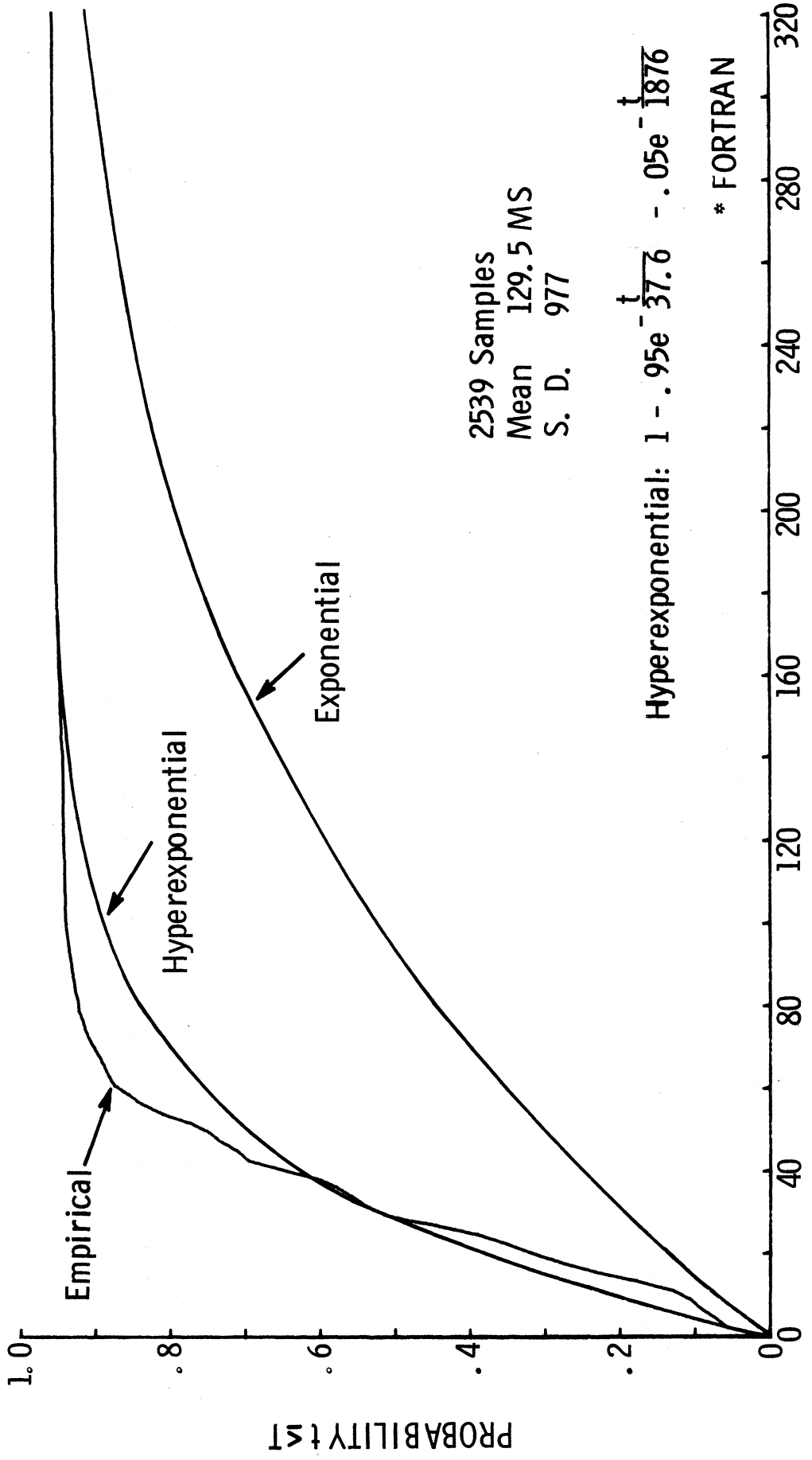

Figure D. 3. Cumulative Distribution of Time (t) Used Per I/O Request During Fortran Compilations

Table D.2

Fortran Execution Data

Item	Samples	Mean	S.D.
Elapsed Time Per Run (sec)	40	19.1	27.0
CPU Time Per Run	40	4.90	5.37
I/O Time Per Run	40	8.13	16.3
Page Wait Time Per Run	40	.434	.758
Number I/O Requests Per Run	40	63.1	71.9
Number Page Waits Per Run	40	14.4	24.1
Virtual Pages Per CPU Interval	2504	29	7
Real Pages Per CPU Interval	2504	24	7
Virtual Pages Per I/O Interval	2539	30	9
Real Pages Per I/O Interval	2539	25	8

Figures D.4 and D.5 give distributions on the CPU time used between page faults. Figure D.5 is a breakdown of the data in Figure D.4 by mean number of pages in core during the period of CPU use between page faults. Pertinent data for this figure are given in Table D.3.

Table D.3

Data for Figure D.5

Pages in Core	Samples	Mean	S.D.
0 - 9	29	7.54	13.6
10 - 19	127	43.6	136
20 - 29	325	205	775
30 - 39	57	620	1179
40 - 49	6	415	492

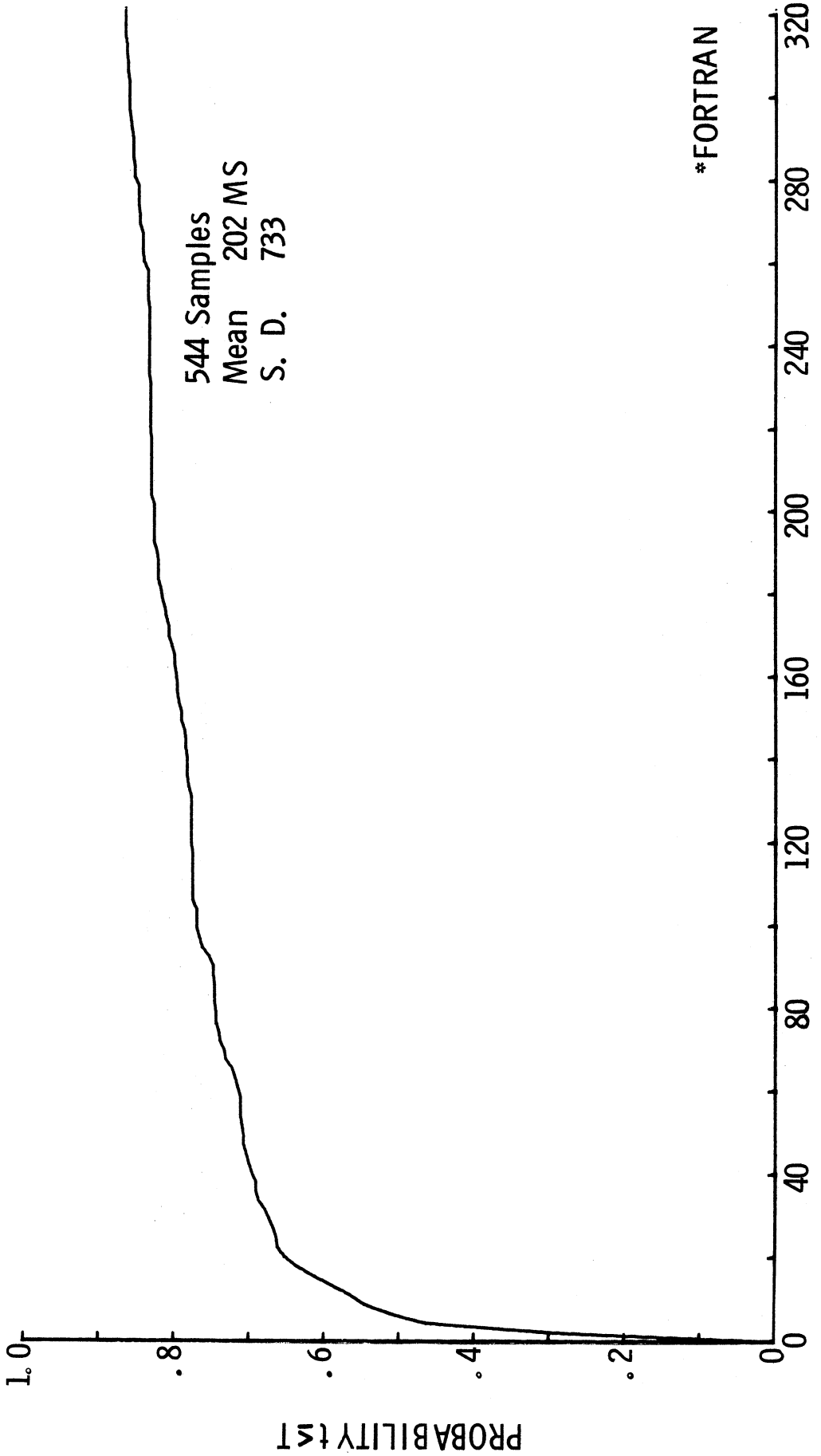

Figure D. 4. Cumulative Distribution of CPU Time (t) Used Between Page Faults During Fortran Compilations

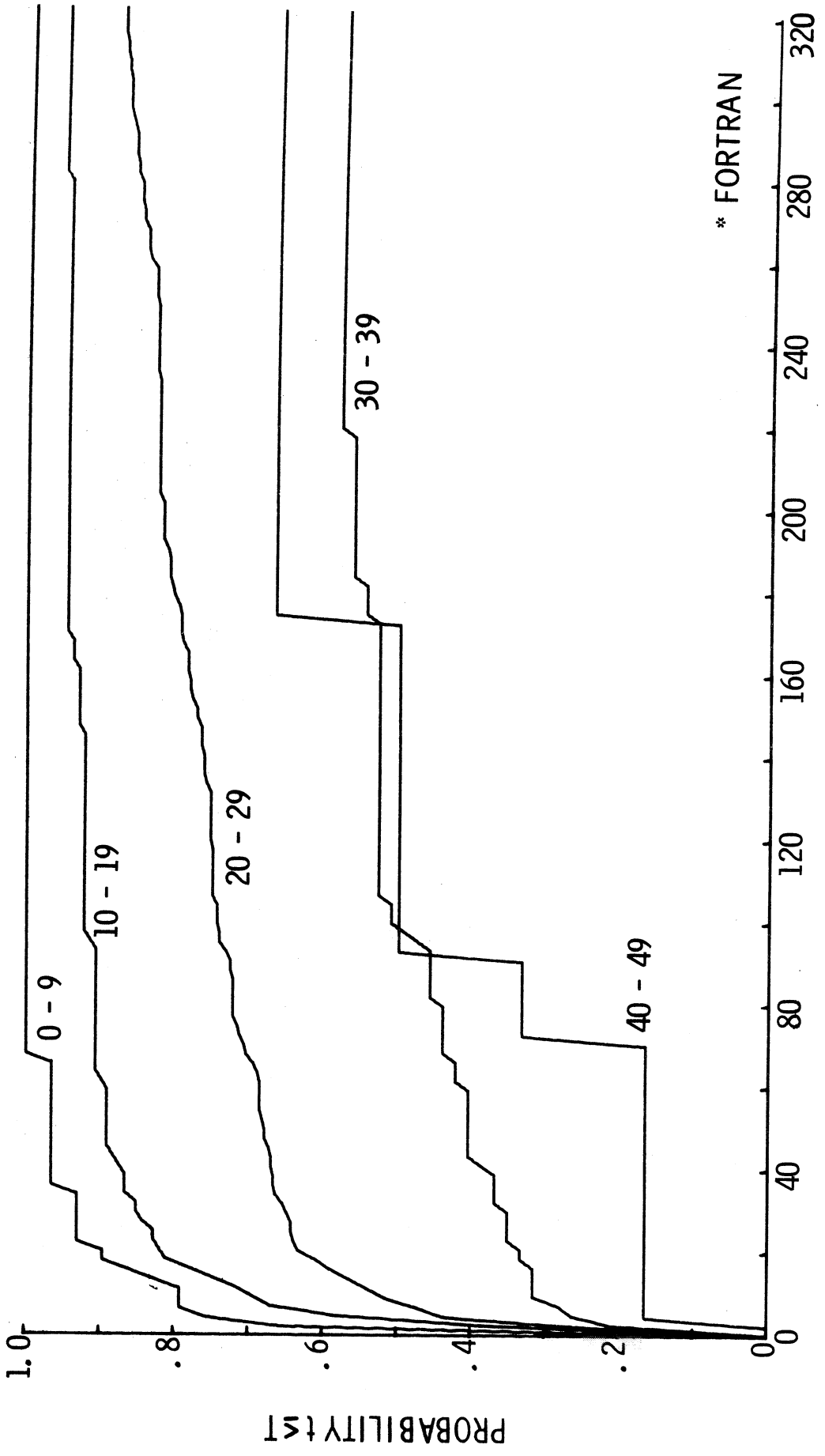

Figure D. 5. Cumulative Distribution of CPU Time (t) Used Between Page Faults as a Function of the Number of Pages in Core For Fortran Compilations

D.5 Fortran compiler (University of Waterloo)

This is the so-called Watfor compiler developed at The University of Waterloo. Invoking this compiler causes the user's source program to be both compiled and executed. There are two versions available on MTS; the version reported here has the smaller storage requirements of the two and is most often used by students. There were 13 requests for this compiler during the data collection period. Mean elapsed loading time was 8.17 seconds with a standard deviation of 3.01. Execution time data are given in Table D.4 for this compiler and include both compile time and execution of the compiler output.

Table D.4

Watfor Execution Data

Item	Samples	Mean	S.D.
Elapsed Time Per Run (sec)	12	41.0	68.8
CPU Time Per Run	12	4.85	1.45
I/O Time Per Run	12	31.0	66.4
Page Wait Time Per Run	12	.900	1.33
Number I/O Requests Per Run	12	92.3	36.8
Number Page Waits Per Run	12	31.6	44.7
Virtual Pages Per CPU Interval	1185	133	34
Real Pages Per CPU Interval	1185	26	13
Virtual Pages Per I/O Interval	1186	135	22
Real Pages Per I/O Interval	1186	26	10

Figures D. 6 - D. 9 present empirical cumulative distributions for lengths of CPU intervals between I/O operations and between page faults and for lengths of I/O operations similar to Figures D. 2 - D. 5.

D. 6 Assembler

The assembler used by MTS is The University of Waterloo G-Assembler. The entire assembler is not loaded when initially called. Rather, modules are loaded selectively depending on options specified by the user. For example the user may specify the model 67 instruction set (default) or the instruction set for some other model of the 360. Default options were used for all 8 of the runs during the data collection period. Mean elapsed loading time for all the modules for each run was 9. 63 seconds. Table D. 5 gives pertinent execution time data for the assembler.

Table D. 5

Assembler Execution Data

Item	Samples	Mean	S. D.
Elapsed Time Per Run (sec)	8	47. 4	46. 9
CPU Time Per Run	8	7. 58	5. 50
I/O Time Per Run	8	34. 9	50. 9
Page Wait Time Per Run	8	. 561	. 495
Number I/O Requests Per Run	8	103	64. 5
Number Page Waits Per Run	8	19. 7	17. 7
Virtual Pages Per CPU Interval	791	78	34
Real Pages Per CPU Interval	791	31	13
Virtual Pages Per I/O Interval	822	81	37
Real Pages Per I/O Interval	822	33	13

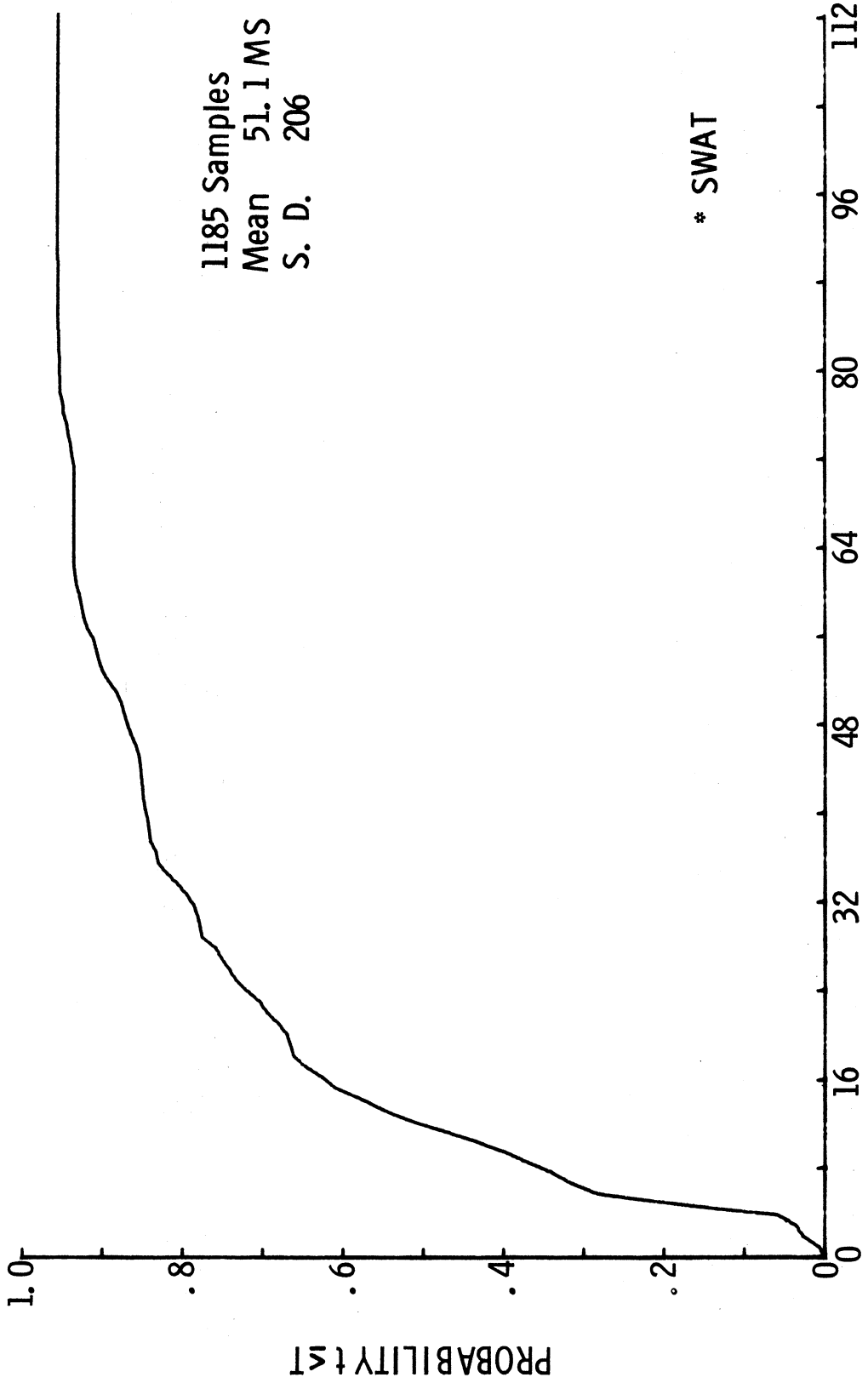

Figure D. 6. Cumulative Distribution of CPU Time (t) Used Between I/O Operations During Watfor Executions

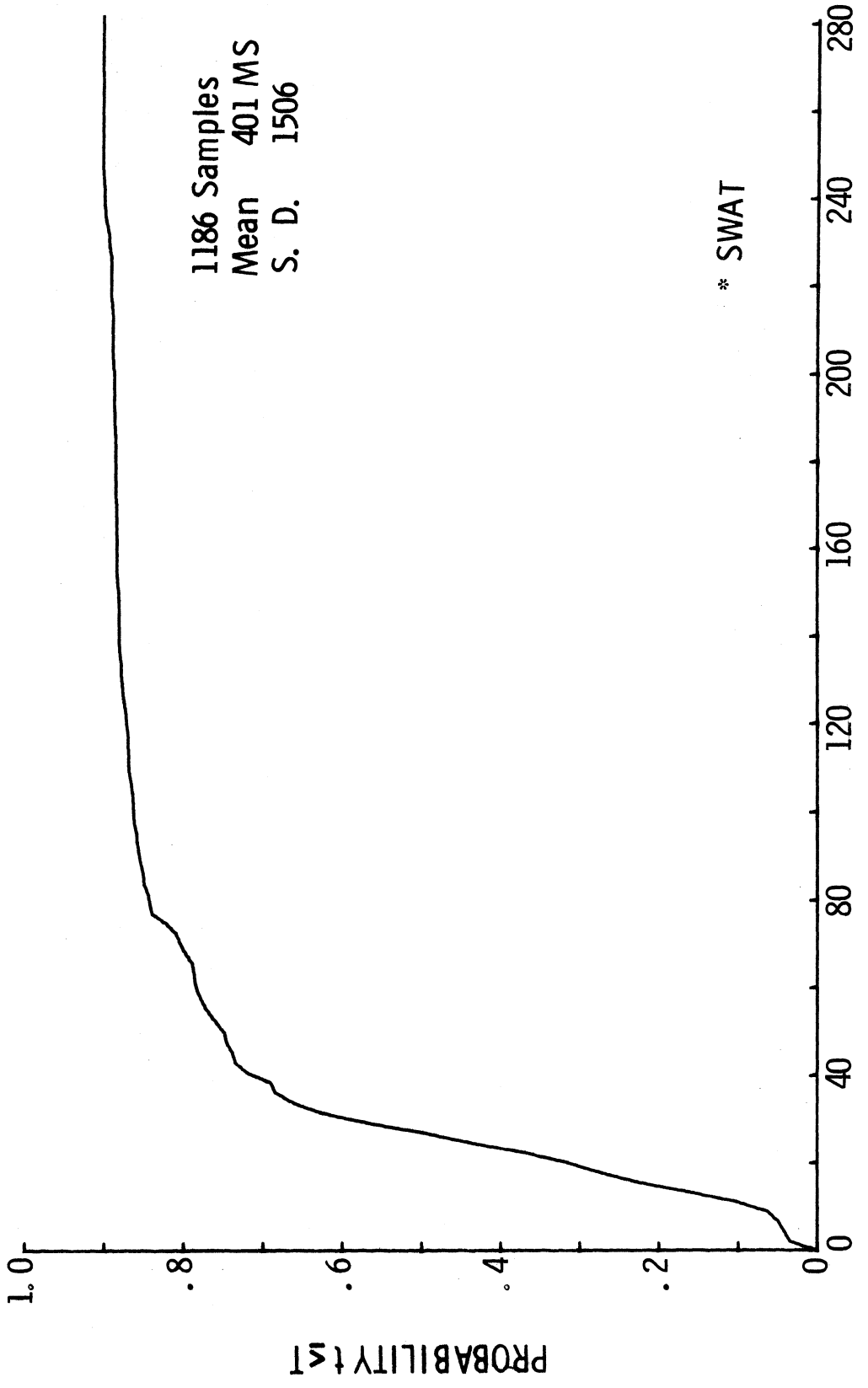

Figure D. 7. Cumulative Distribution of Time (t) Used Per I/O Request During Watfor Executions

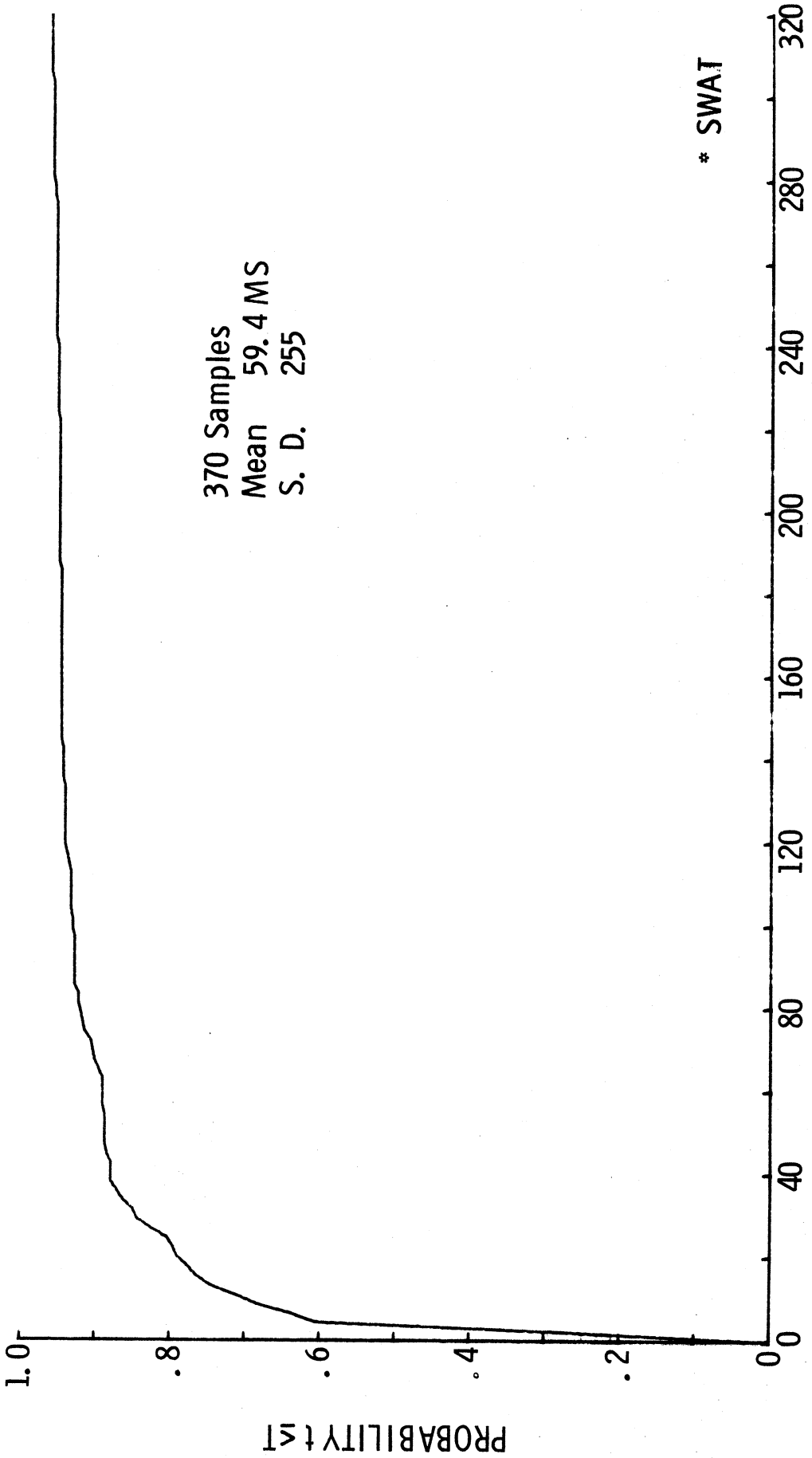

Figure D. 8. Cumulative Distribution of CPU Time (t) Used Between Page Faults During Watfor Executions

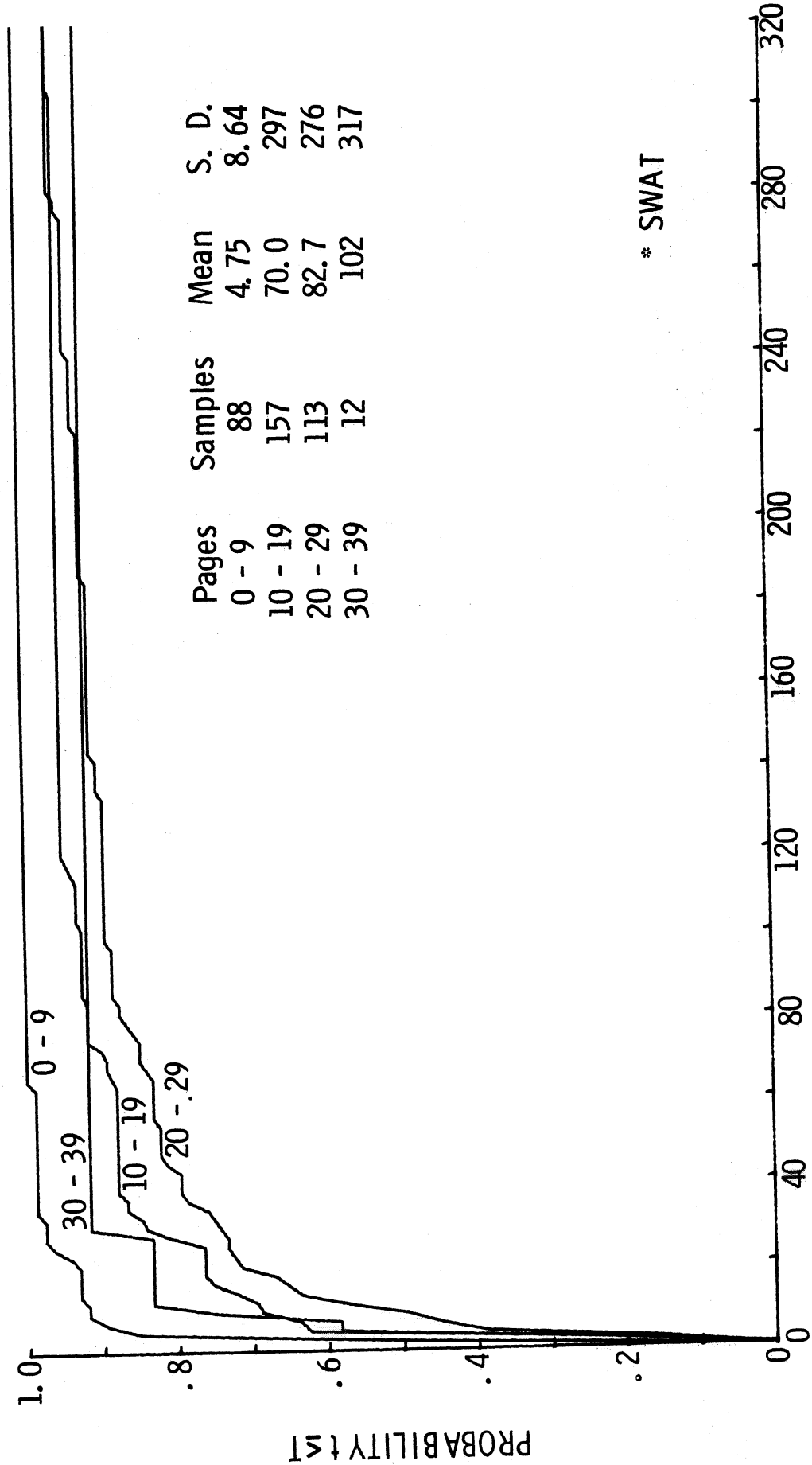

Figure D. 9. Cumulative Distribution of CPU Time (t) Used Between Page Faults As Function Number Pages in Core During Watfor Executions

Figures D. 10 - D. 13 show cumulative distributions for the same quantities shown in Figures D. 2 - D. 5. Table D. 6 gives pertinent data for Figure D. 13.

Table D. 6

Data for Figure D. 13

Pages in Core	Samples	Mean	S. D.
0 - 9	27	5. 38	6. 88
10 - 19	21	11. 4	14. 3
20 - 29	20	209	565
30 - 39	39	253	479
40 - 49	10	222	364
50 or more	18	201	276

D. 7 Line File Editor

The editor is an interactive program which provides terminal users with a powerful facility for making desired changes in their line files. There were 10 requests for the editor during the data collection period and 7 complete executions. Because the editor is likely to be used for relatively long periods of time, three of the ten users who began execution during the data collection period were still in execution when the period ended. Mean elapsed loading time for the editor was 4. 41 seconds with a standard deviation of 1. 32. Table D. 7 gives execution data for the editor. Note that the

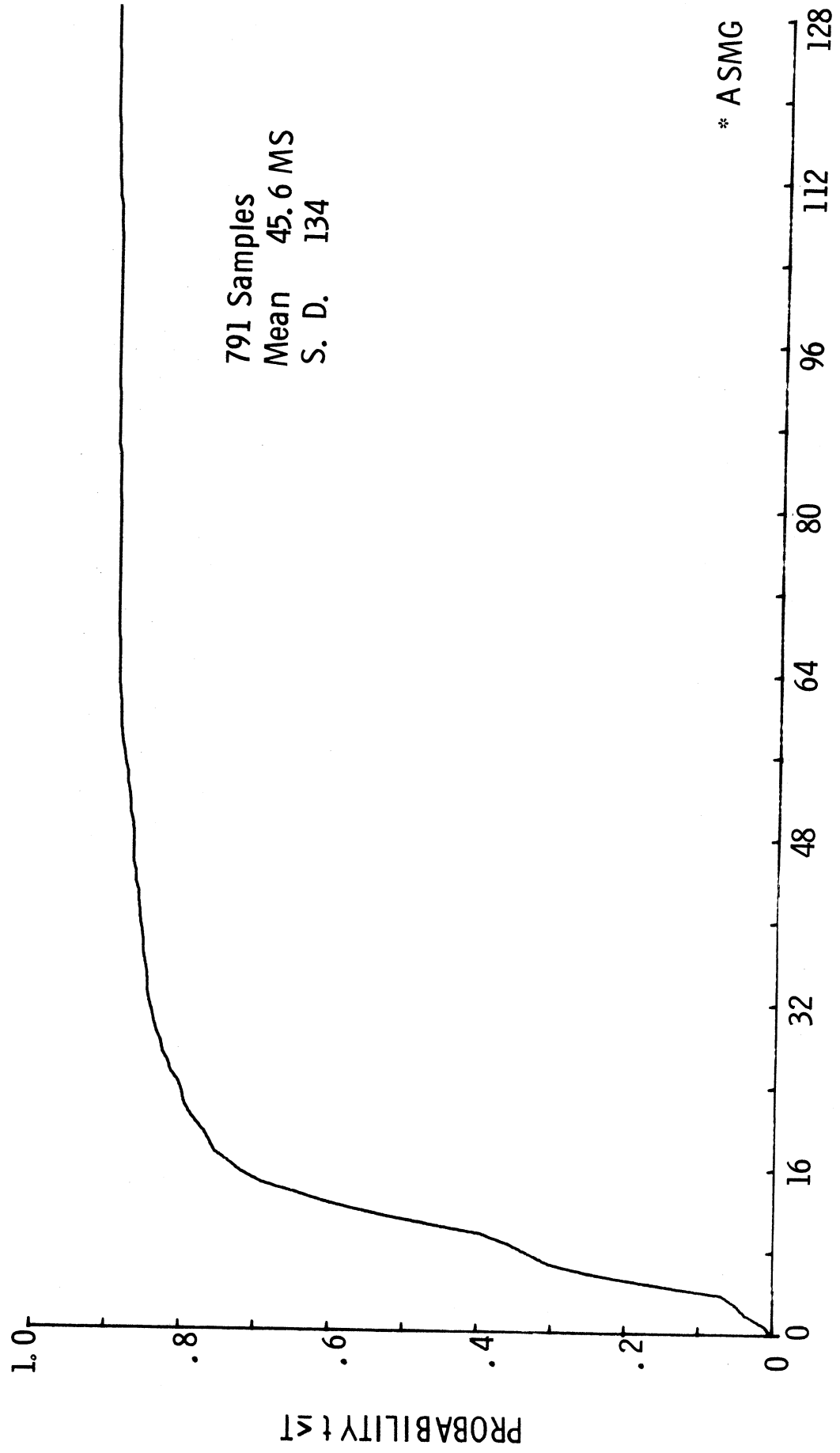

Figure D. 10. Cumulative Distribution of CPU Time (t) Used Between I/O Operations By Assembler

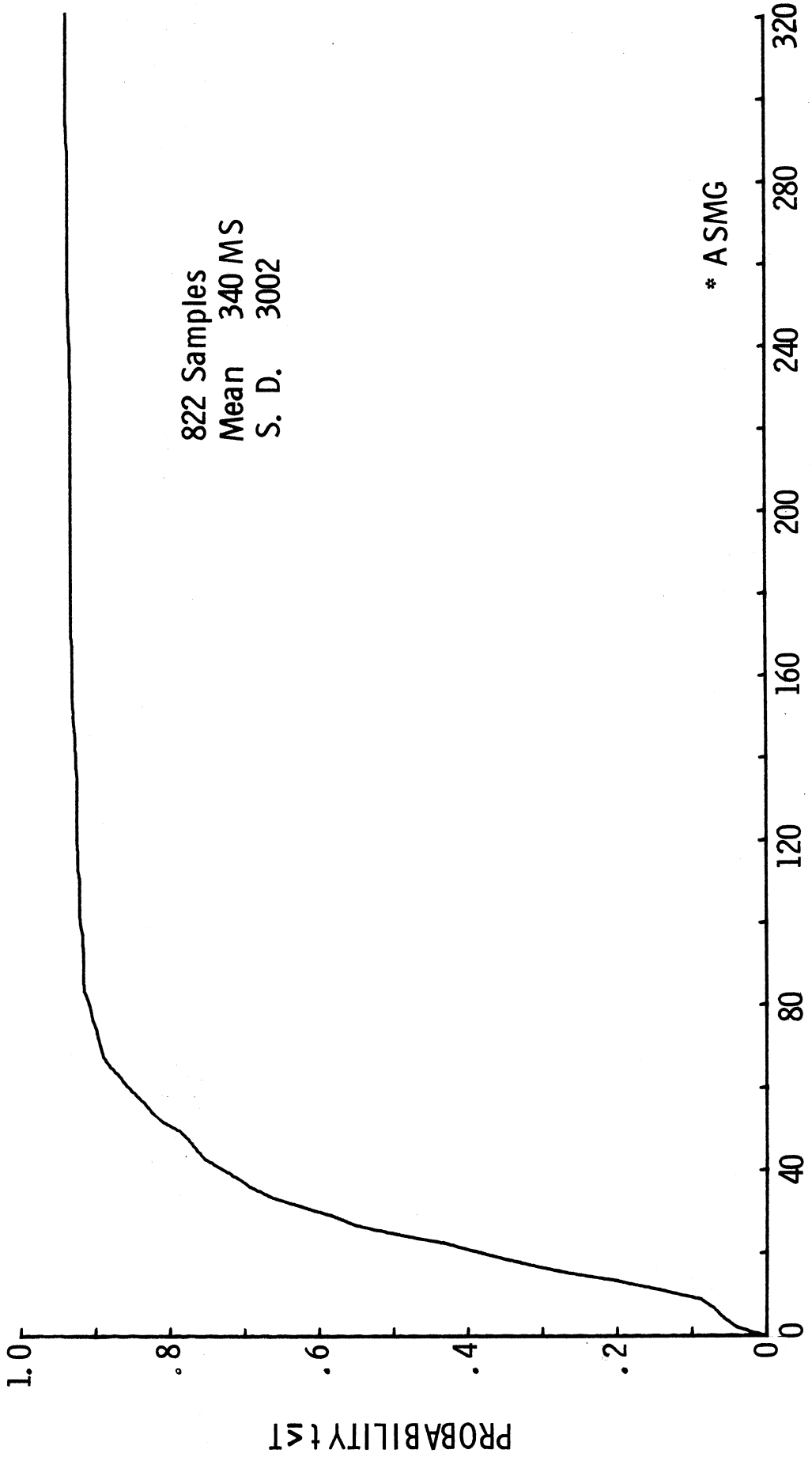

Figure D. 11. Cumulative Distribution of Time (t) Used Per I/O Request by Assembler

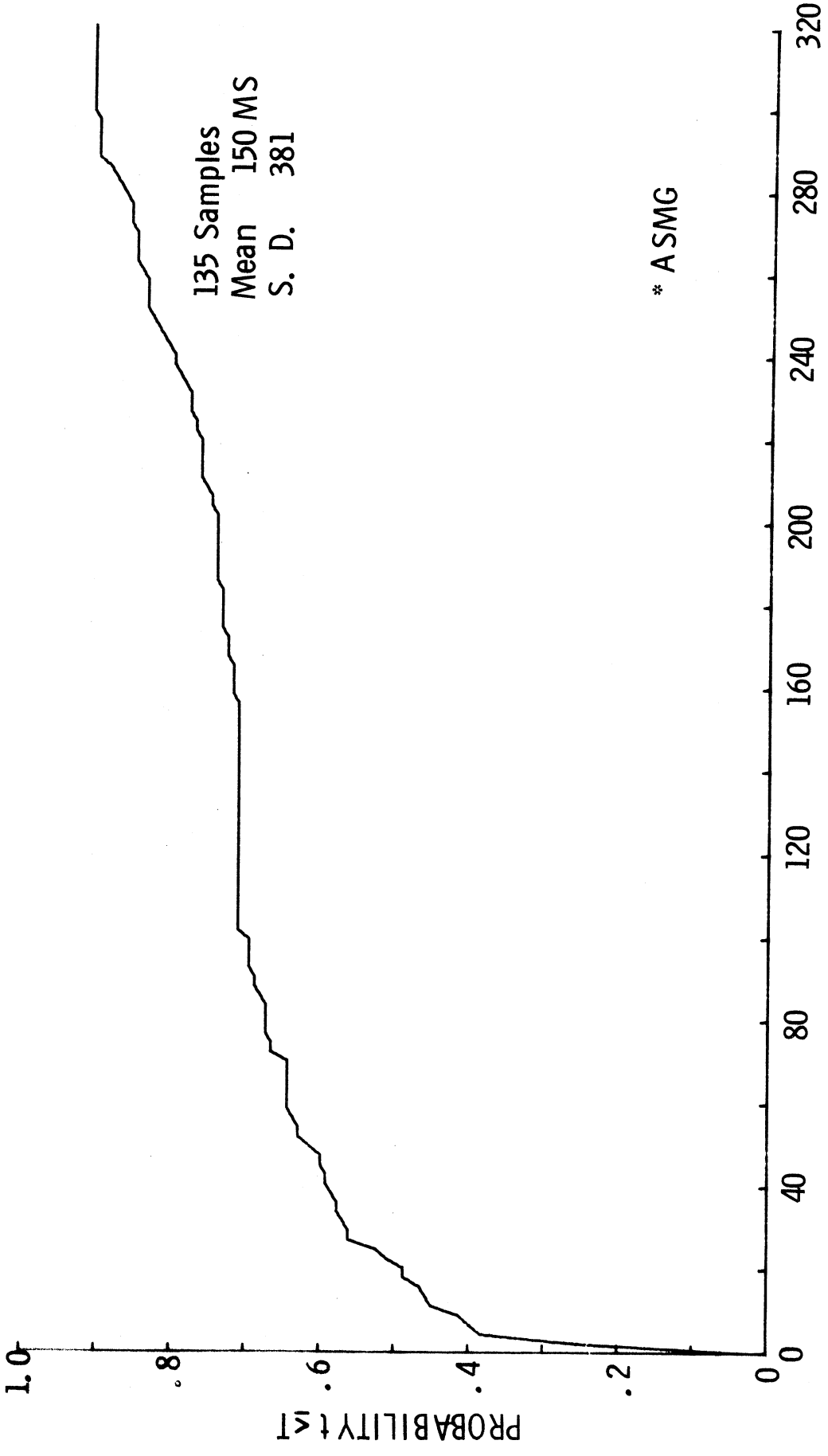

Figure D.12. Cumulative Distribution of CPU Time (t) Used Between Page Faults by Assembler

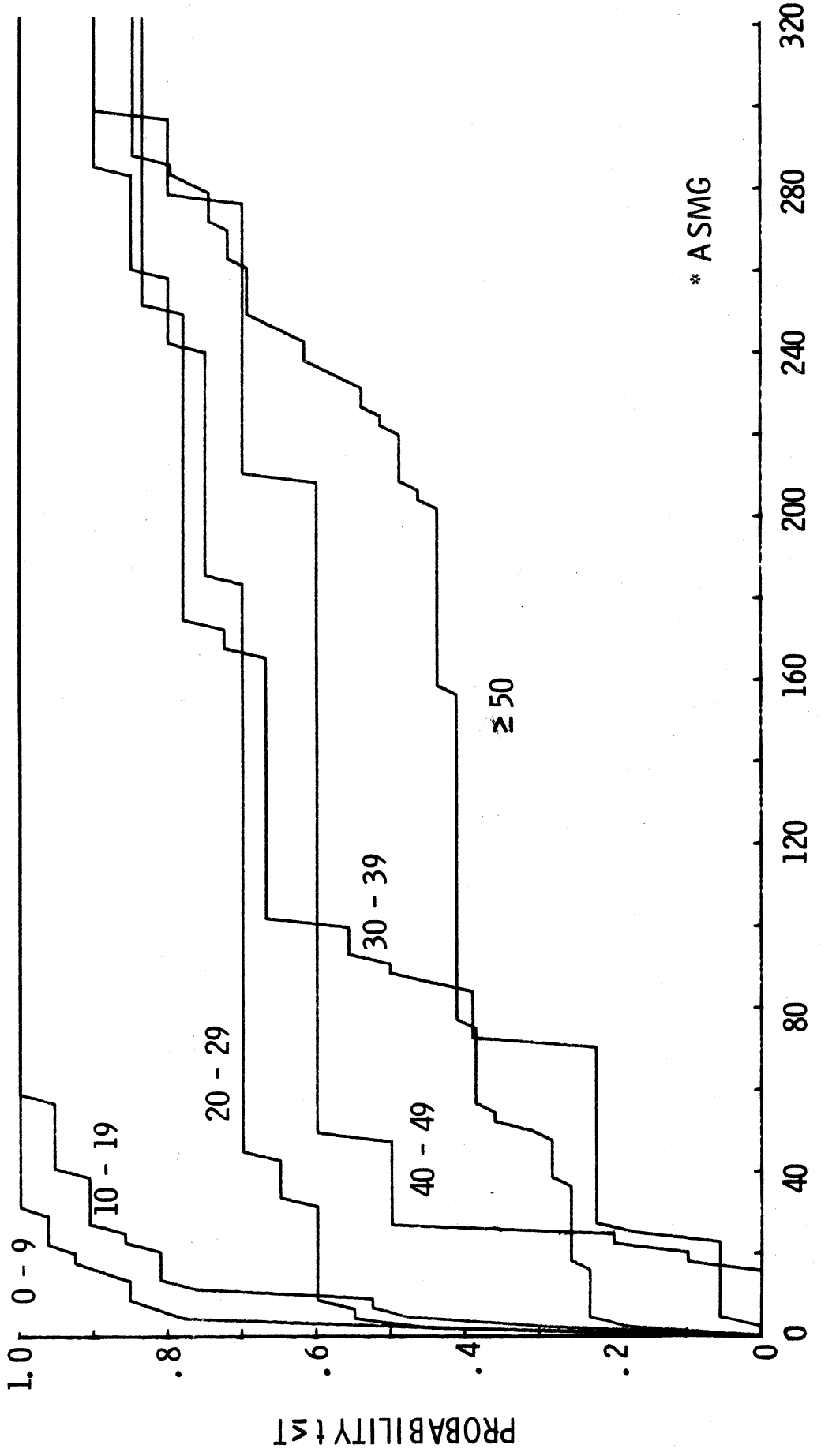

Figure D.13. Cumulative Distribution of CPU Time (t) Used Between Page Faults As Function Number Pages in Core For Assembler

items given "per run" may be somewhat biased toward shorter runs since data do not appear for the runs begun but not completed during the time data were being collected.

Table D.7

Editor Execution Data

Item	Samples	Mean	S.D.
Elapsed Time Per Run (sec)	7	219	365
CPU Time Per Run	7	2.07	2.75
I/O Time Per Run	7	215	357
Page Wait Time Per Run	7	2.16	3.22
Number I/O Requests Per Run	7	141	210
Number Page Waits Per Run	7	67.4	115
Virtual Pages Per CPU Interval	1214	13	6
Real Pages Per CPU Interval	1214	8	5
Virtual Pages Per I/O Interval	1219	14	4
Real Pages Per I/O Interval	1219	9	3

Figures D.14 - D.17 show cumulative distributions for the same quantities plotted in Figures D.2 - D.5.

D.8 User-generated Tasks

The data in this section were obtained from all tasks run from files with names not preceded by an asterisk. These are in general object modules of programs written by users. There were 83 loads and 61 executions during the data collection period. In this case, in contrast to the line file editor, the disparity between

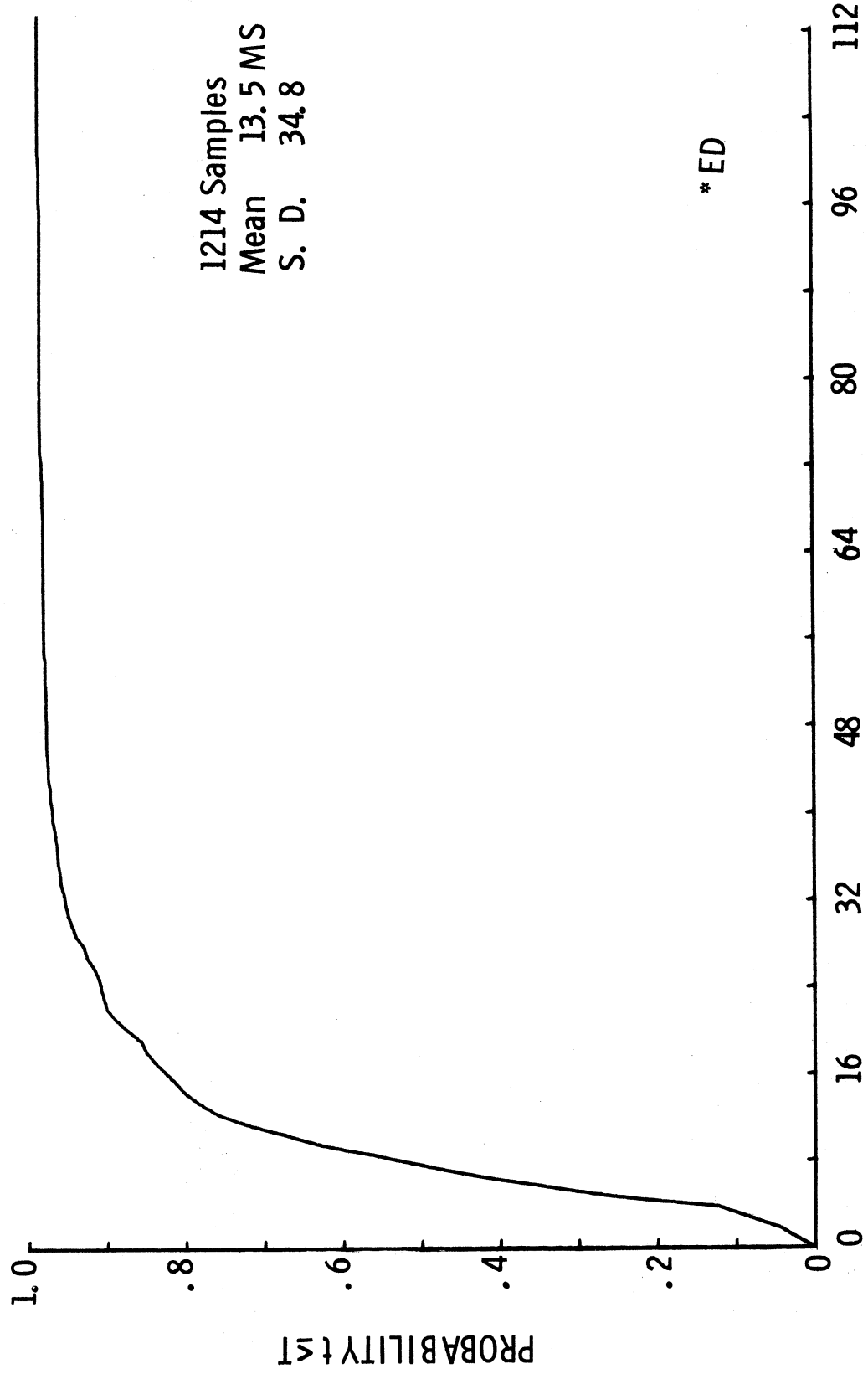

Figure D. 14. Cumulative Distribution of CPU Time (t) Used Between I/O Operations By Editor

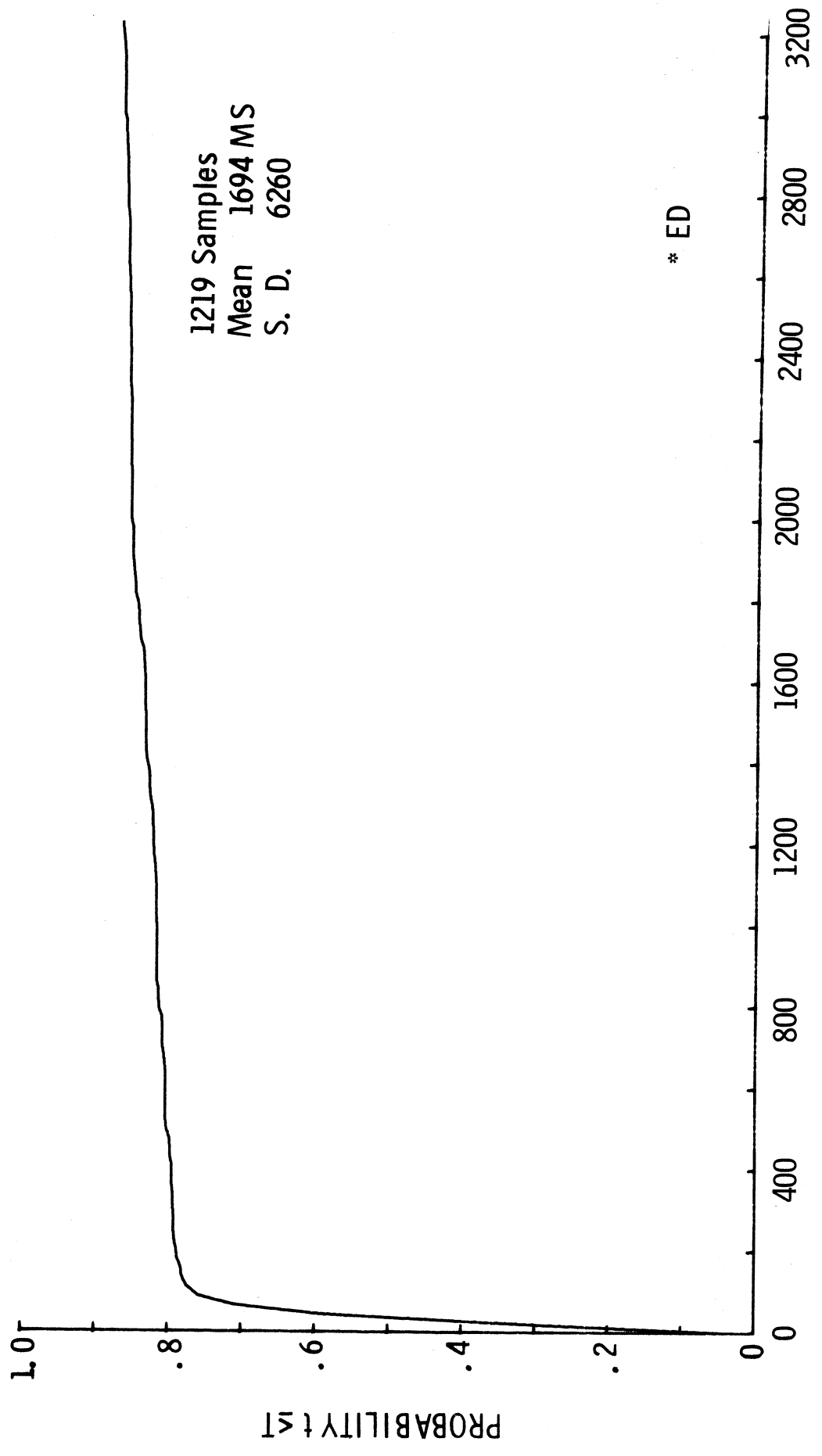

Figure D. 15. Cumulative Distribution of Time (t) Used Per I/O Request By Editor

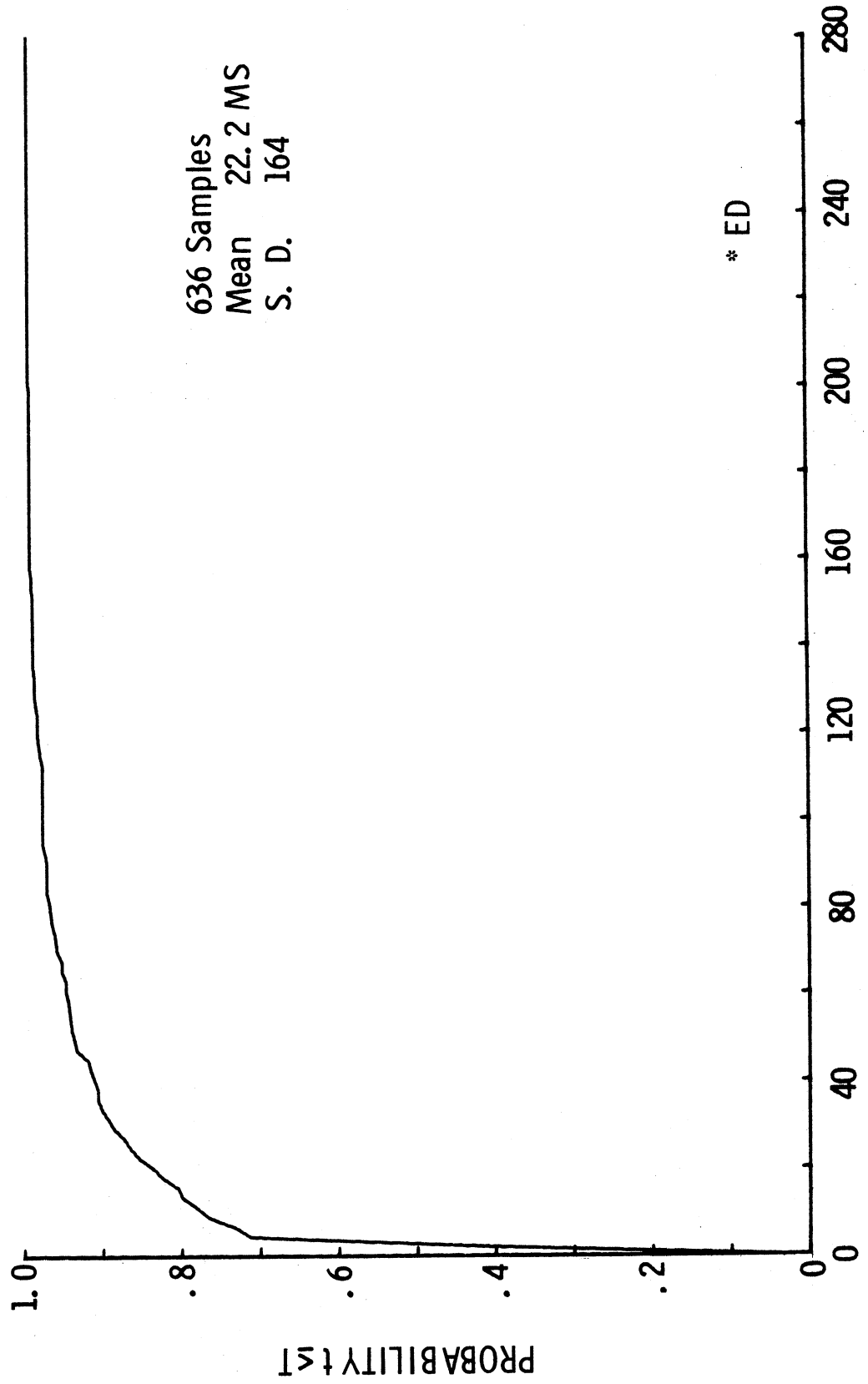

Figure D.16. Cumulative Distribution of CPU Time (t) Used Between Page Faults By Editor

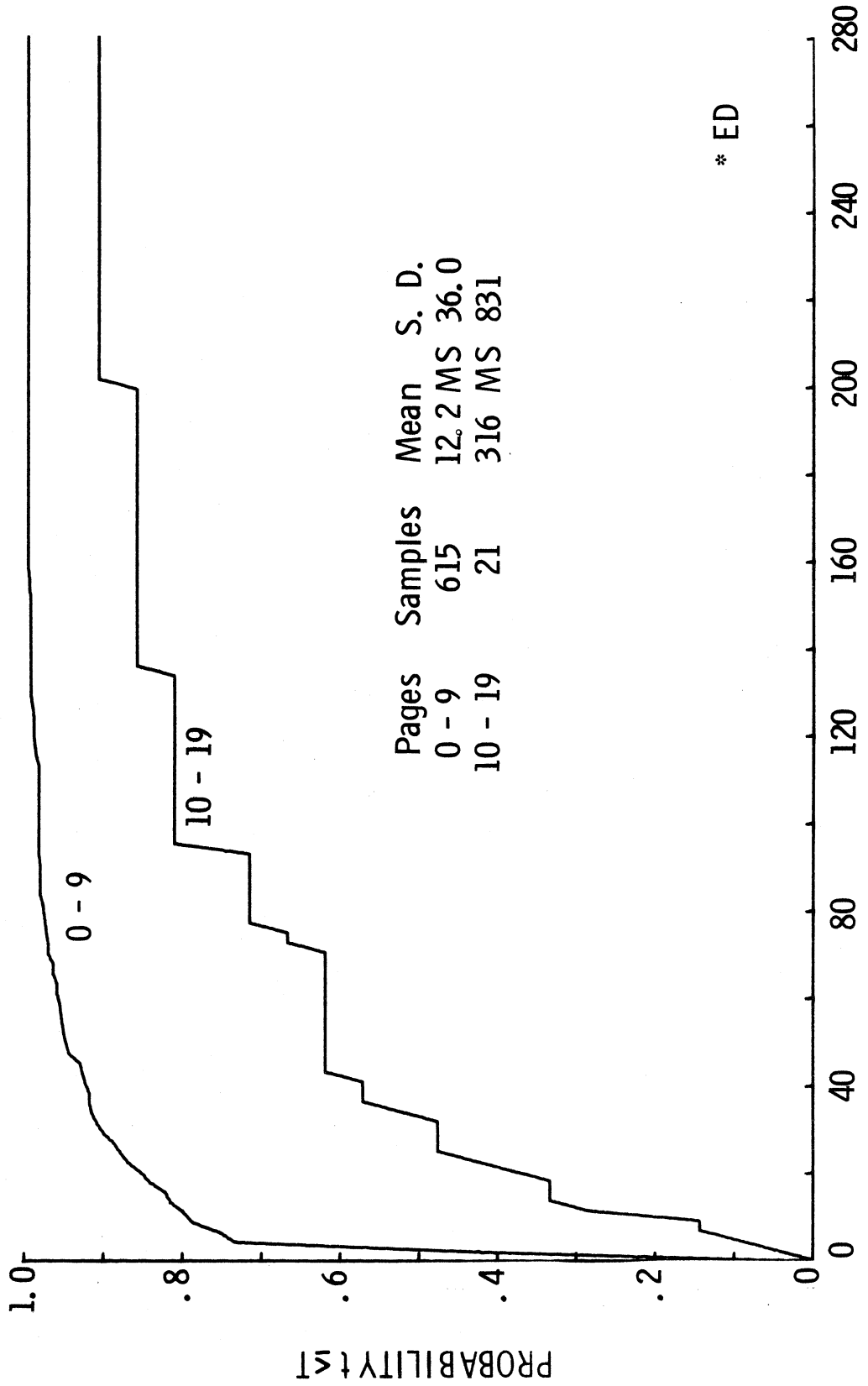

Figure D.17. Cumulative Distribution of CPU Time (t) Between Page Faults As Function Number Pages In Core For Editor

the number of loads and number of executions is probably due in part to a number of unsuccessful loading attempts. The mean elapsed load time was 13.4 seconds with a standard deviation of 15.1. Some of the long load times may have occurred because conversational users initially provided the loader with insufficient data and were thus prompted at the terminal to provide more.

Execution data are given in Table D.8 and distributions in Figures D.18 - D.21. Table D.9 gives parameters for Figures D.21. The long I/O times shown in Table D.8 and Figure D.19 may be due to terminal interactions built in by the user but in many cases are probably caused by undebugged programs being interrupted during execution. If the user is at a terminal the program remains loaded after an interruption and he may display or alter the contents of chosen memory locations and restart the program. These operations appear in the execution data.

Table D.8

User Program Execution Data

Item	Samples	Mean	S.D.
Elapsed Time Per Run (sec)	61	102	138
CPU Time Per Run	61	6.24	22.1
I/O Time Per Run	61	88.6	114
Page Wait Time Per Run	61	.853	1.09
Number I/O Requests Per Run	61	93.0	136
Number Page Waits Per Run	61	29.1	37.5
Virtual Pages Per CPU Interval	9296	23	16
Real Pages Per CPU Interval	9296	12	9
Virtual Pages Per I/O Interval	9303	24	16
Real Pages Per I/O Interval	9303	13	9

Table D.9

Parameters for Figure D.21

Pages in Core	Samples	Mean (MS)	S.D.
0 - 9	1952	132	2097
10 - 19	1068	52.3	224
20 - 29	186	297	773
30 - 39	143	253	656
40 - 49	8	76.9	65.1
50 or More	1	3.70	0

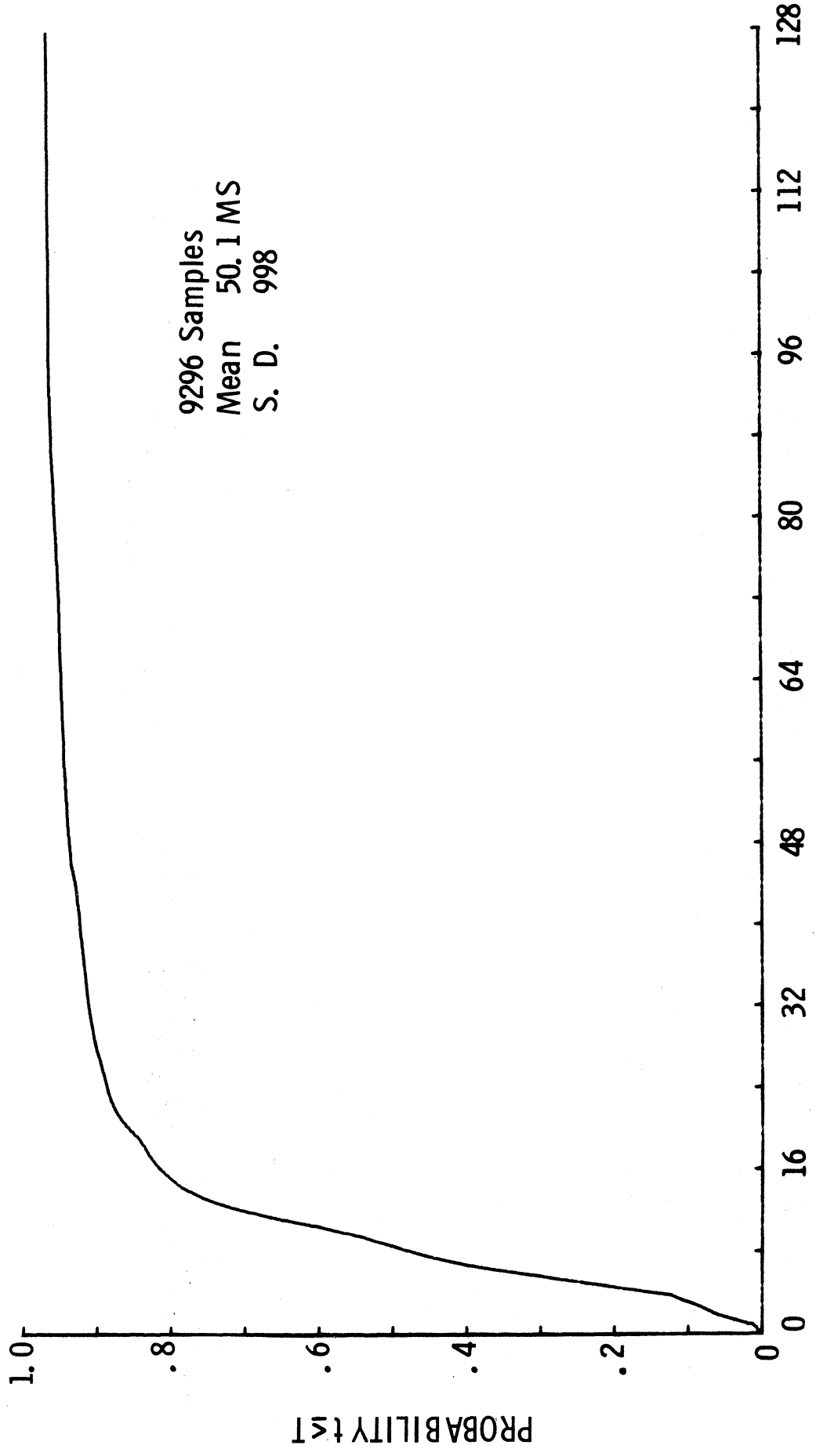

Figure D. 18. Cumulative Distribution of CPU Time (t) Used Between I/O Requests By User-generated Tasks

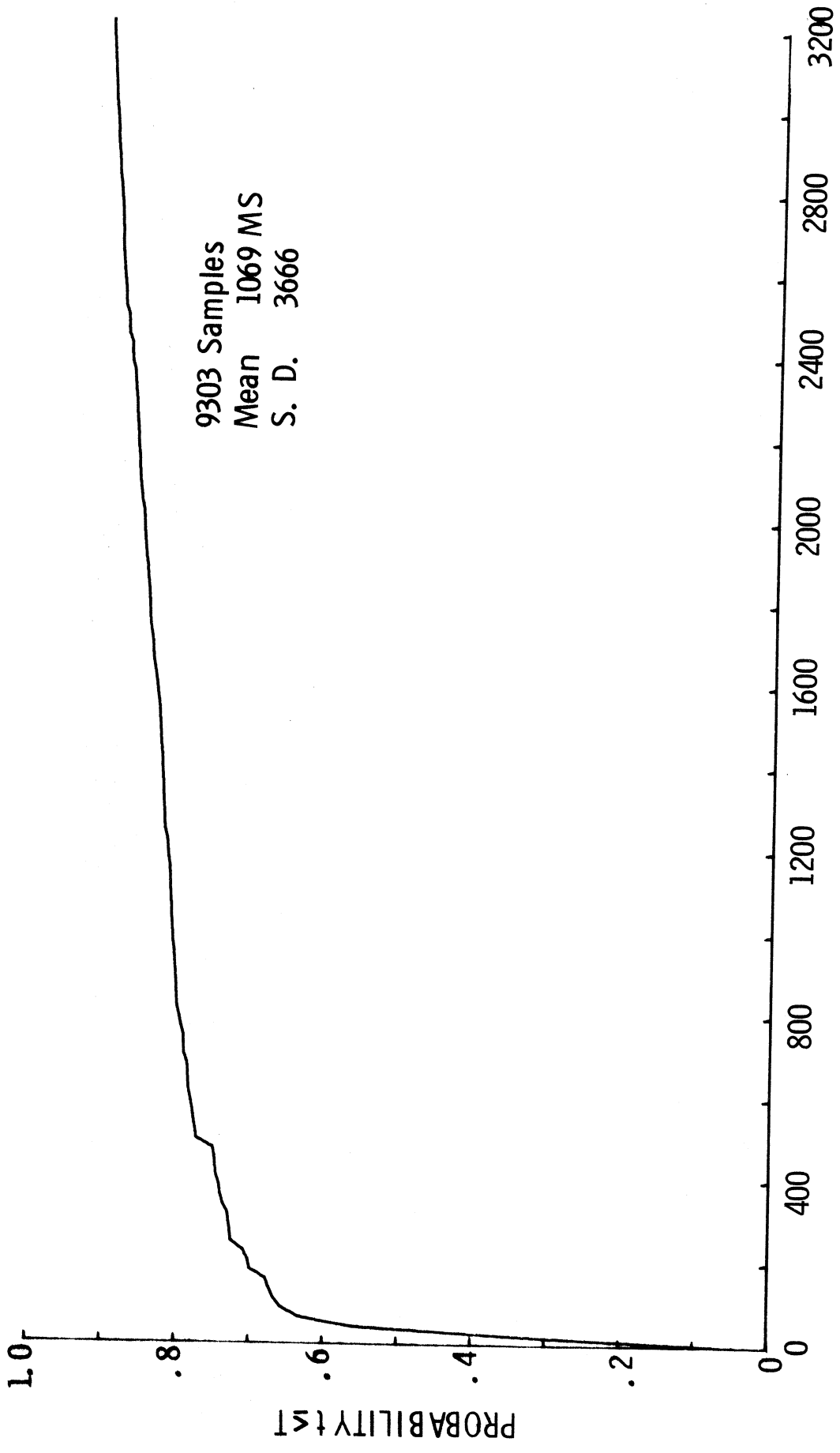

Figure D.19. Cumulative Distribution of Time (t) Used Per I/O Request By User-generated Tasks

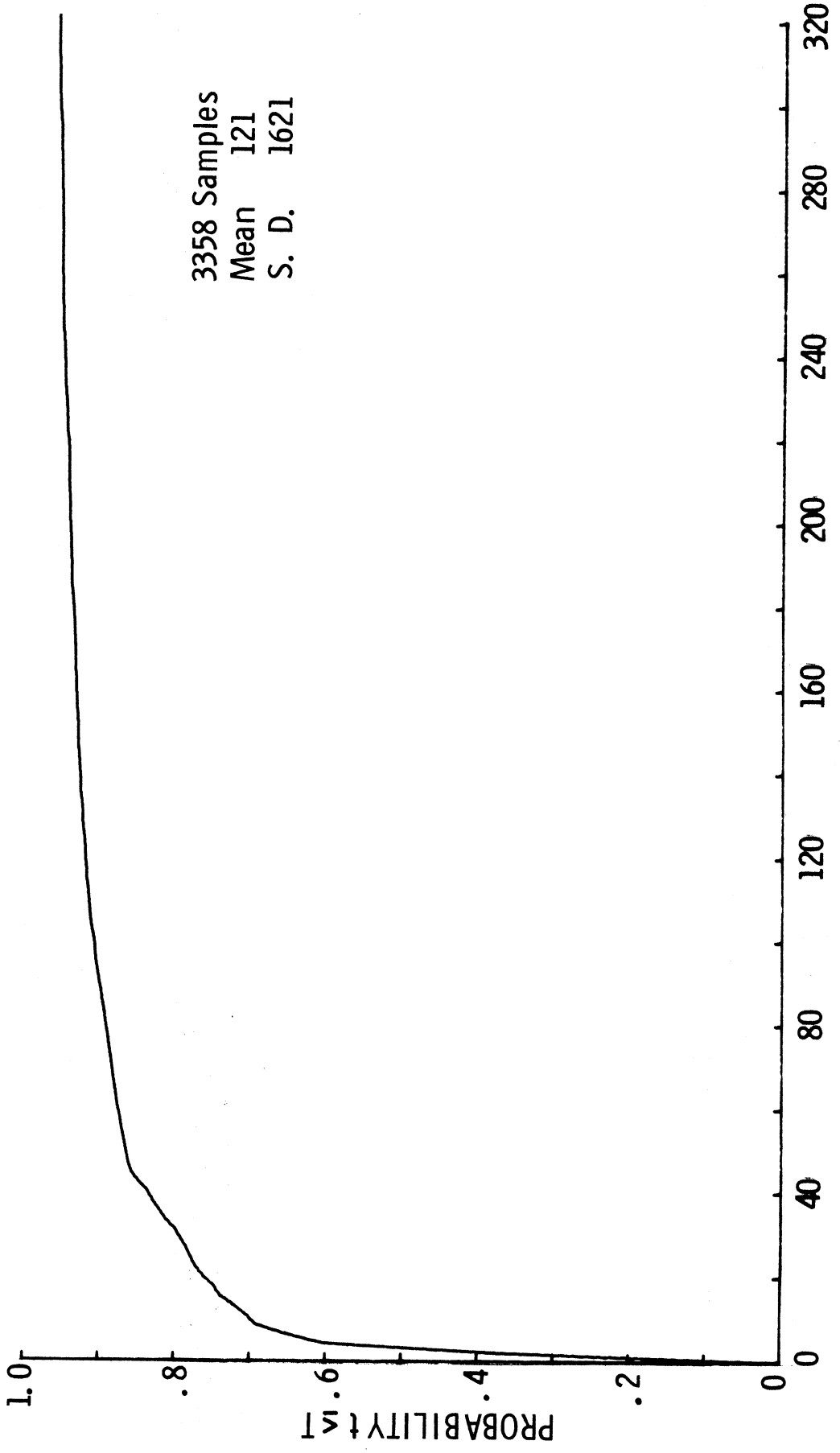

Figure D. 20. Cumulative Distribution of CPU Time (t) Used Between Page Faults By User-generated Tasks

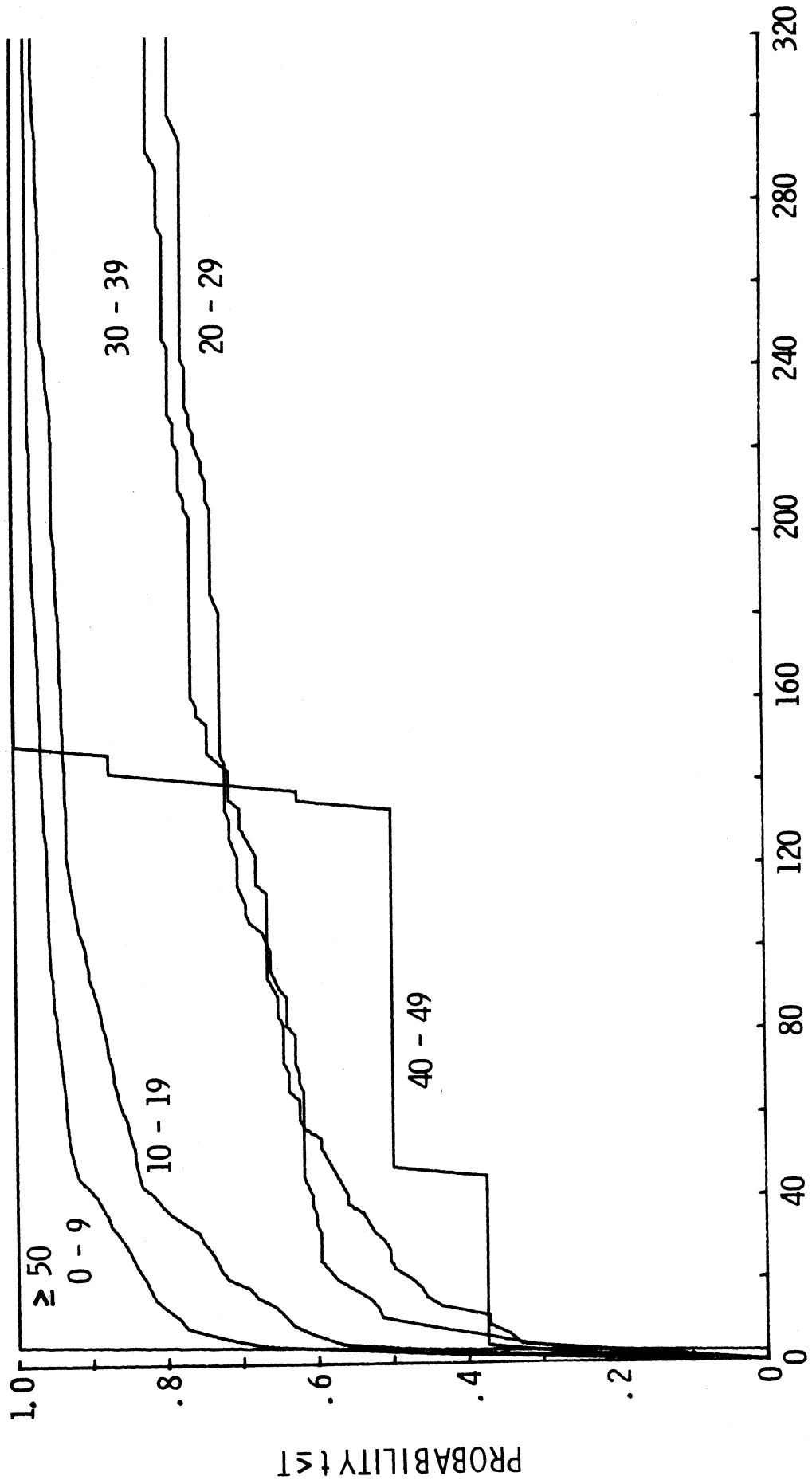

Figure D. 21. Cumulative Distribution of CPU Time (t) Used Between Page Faults By User-generated Tasks As Function Number Pages in Core

Appendix E

REACHABILITY OF STATE m+(L-1)N

This appendix is devoted to showing that state m+(L-1)N is occupied with probability greater than zero after N or less transitions of the U-process regardless of the initial state when $d^k_{mm}(1) > 0$ for all k. The U-process is the Markov chain induced by using stochastic matrices U^{A_1}, U^{A_2},... as transition probability matrices for successive state transitions where A_1, A_2,... are arbitrary L-stage policy sequences. These matrices are defined in section 2.3 and one is shown in Figure 2.3. Reference to this figure will make the arguments in this appendix easier to follow. The underlying Markov chain of the MRDP is called the P-process in this appendix and is restricted to be such that no matter what the sequence of policies used to determine the probabilities of successive transitions, state m may be occupied after N-1 or less transitions regardless of the initial state. For some particular policy, α, a P-matrix, $P_\alpha = (p_{ij}^{a(i)})$, is defined, and similarly for any L-stage policy sequence, A, a U-matrix U^A is defined. Transitions in the N state process defined by a sequence of P-matrices will be called P-transitions while those in the LN state process defined by a sequence of U-matrices will be called U-transitions. The remainder of this appendix is given in the form of a theorem and its proof.

Theorem E.1

Hypothesize a MRDP with a state m which may be occupied with probability greater than zero after N-1 or less transitions regardless of the sequence of policies used. Hypothesize also that $d^k_{mm}(1)>0$ for all k, and let arbitrary L-stage policy sequences; $A_1, A_2, \ldots$; be used to define stochastic matrices $U^{A_1}, U^{A_2}, \ldots$ in the manner discussed in Chapter 2. Then regardless of the $A_1, A_2, \ldots$ chosen and regardless of the initial state occupied, state m+(L-1)N is occupied with probability greater than zero after N or less transitions in the Markov process induced by using $U^{A_1}, U^{A_2}, \ldots$ as transition probability matrices for successive transitions.

Proof

First it is argued that transition is possible from any one of states m, m+N,..., m+(L-1)N in the U-process to state m+(L-1)N. Inspection of the last column of submatrices; $U_{1L}, \ldots, U_{LL}$; shows that element (m,m) is greater than zero in each. This is easily seen by noting that it is true in the first submatrix row and that in submatrix row I+1 we have a term $D_I(1)\, U_{IL}$ where by hypothesis $[D_I(1)]_{mm}>0$. Thus if $(U_{IL})_{mm}>0$, then $(U_{I+1,L})_{mm}>0$ and since $(U_{1L})_{mm}>0$ this part of the proof is complete. Therefore, if transition to any of states m, m+N,..., m+(L-1)N can be shown to be possible in N-1 or less transitions, we are done.

At this point it is useful to define a generic state, i, called generic i in the U-process where i is an integer such that $1 \leq i \leq N$. This term will be used to refer to any one or more than one of the states i, i+N,...,

i+(L-1)N in the U-process. Thus if transition is possible (occurs with probability >0) to generic i in the U-process, then transition is possible to at least one of states i, i+N,..., i+(L-1)N. In view of what was proved in the preceding paragraph, it remains to show that transition occurs with probability greater than zero to generic m in N-1 or less transitions.

Another useful notion in what follows is that of a transition tree. A transition tree is a tree structure which shows all transitions possible out of some initial state in a Markov chain, then all transitions possible out of some or all of those states and so on up to some maximum number of transitions. Thus at any possible node in the tree, the structure may terminate or all possible transitions out of this node may be shown. An example of such a tree is shown in Figure E.1 in which the state represented by each node is given. The minimum path length in this tree is one because no transitions occur out of state two at level one. It is clear that if these are transitions in a P-process where the possible transitions at each level in the tree are determined by the policy chosen at that level; then if the minimum path length in the tree is at least N-1, state m must appear in at least one terminal node of the tree. The tree shown in Figure E.2 has a more complex structure. Here it is assumed that the possible transitions out of any state at any level are all those which could occur under one or more of the possible alternatives in that state. Recall that transitions from state i under alternative k are possible to those states j for which $p_{ij}^{k} > 0$. In Figure E.2, transition from a given

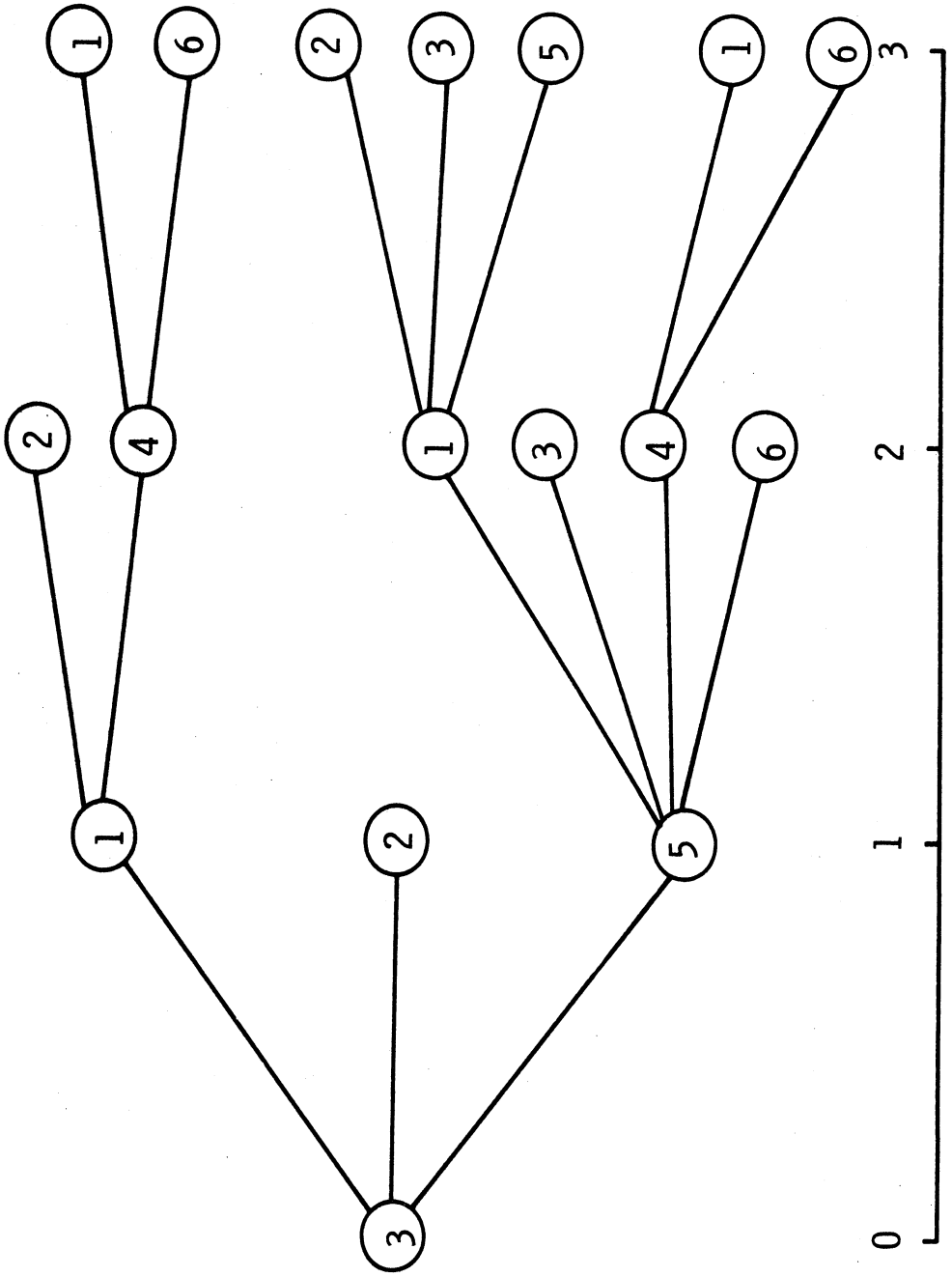

Figure E.1. Transition Tree

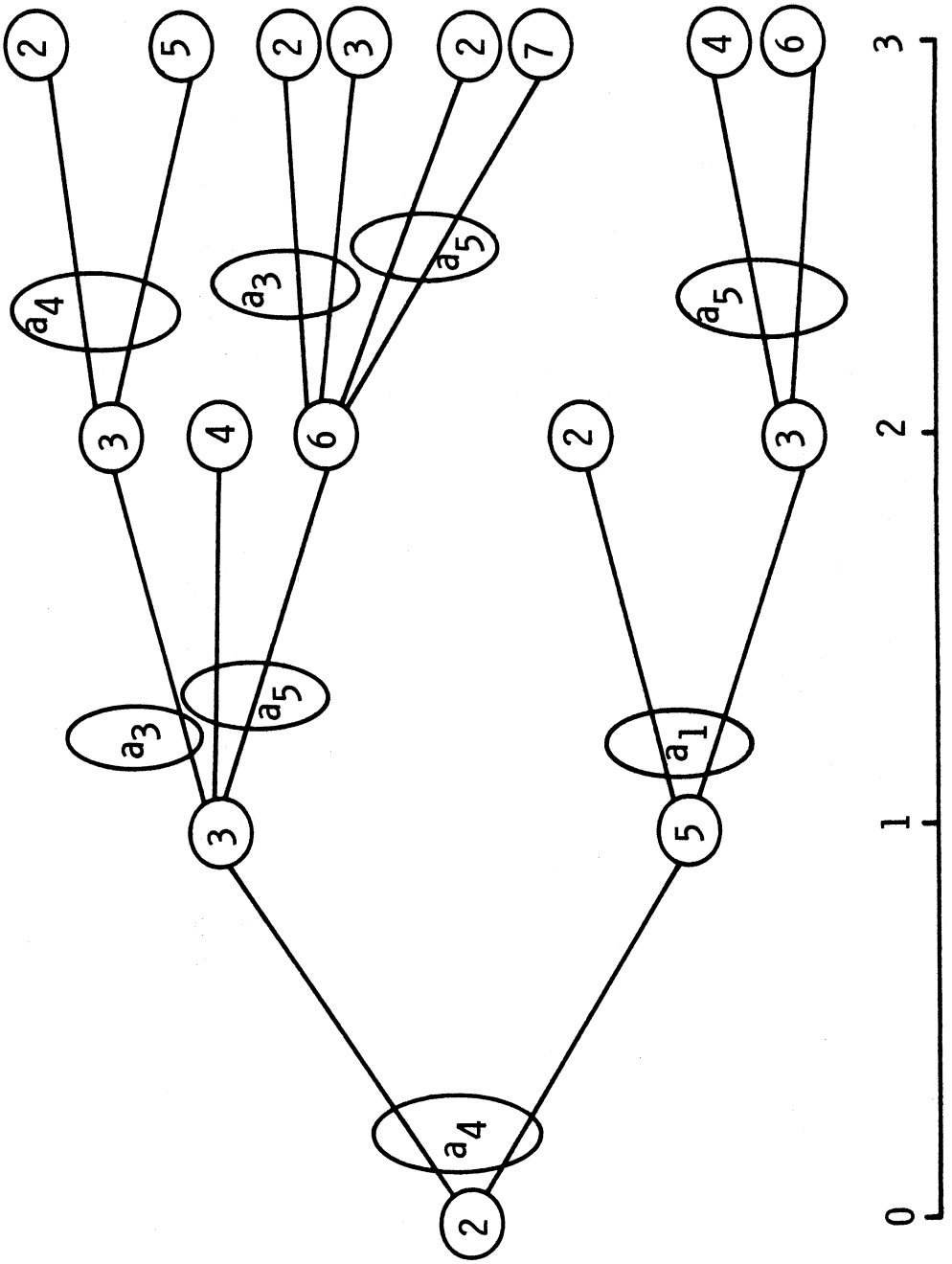

Figure E. 2. Transition Tree Showing Alternatives Which Determine Possible Transitions

state at any level of the tree may be possible to all states reachable under more than a single alternative, k, and thus for the transitions and alternatives possible in the P-matrices it is obviously still true that if the minimum path length in this tree is N-1 or more, state m will appear in a terminal node.

It will be shown in what follows that given a transition out of state $i+xN$ in the U-process for some $0 \leq x \leq L-1$ and for any $1 \leq i \leq N$, there is a corresponding tree of the above form with root node i, with possible transitions determined by the possible P-transitions under one or more alternatives, and with minimum path length one which determines which generic states may be reached as a result of the U-transition. Thus if the tree shows j reachable from i, then in the U-process one or more of states $j, j+N, \ldots, j+(L-1)N$ are reachable from $i+xN$. Then for the next U-transition, the possible initial states include one or more of $j, j+N, \ldots, j+(L-1)N$, and since i and x were arbitrary in the discussion above there is a corresponding tree for each of these initial states.

Therefore, the transition tree corresponding to two successive U-transitions consists of the transition tree for the first transition extended by other trees of the same form with root nodes which are the terminal nodes of the first tree. Thus in N-1 U-transitions, there is a corresponding transition tree with minimum path length N-1 and with possible transitions to generic states determined by possible P-transitions under one or more alternatives which means that generic m is reachable in

N-1 transitions of the U-process regardless of the intitial state. As stated above this is sufficient to show that m+(L-1)N may be occupied after N transitions because it has already been shown that a direct transition from generic m to m+(L-1)N is possible. It remains to show that there exists a transition tree for any U-transition which shows possible transitions to generic U-states which correspond to states to which transitions can be made under specific alternatives in the P-matrices.

The transition tree is exhibited by using an inductive proof on the U-matrix. Such a transition tree is shown to exist for transitions determined by submatrix row one; i.e. transitions out of states i,..., N; and then it is shown that if such a tree exists for submatrix rows 1,...,I, it must exist for submatrix row I+1.

In submatrix row 1 we have submatrices $D_{a_0(1)}(1),\ldots,D_{a_0(L)}(L)$, where an element of one of these matrices is $d_{ij}^{a_0(i)}(\ell) = \phi_{ij}^{a_0(i)}(\ell)p_{ij}^{a_0(i)}$. Thus if $p_{ij}^{a_0(i)} > 0$, then $d_{ij}^{a_0(i)}(\ell) > 0$ for at least one ℓ; $1 \leq \ell \leq L$; so that if transition is possible from i to j under alternative $a_0(i)$, transition is possible from i to generic j when this U-matrix is the transition probability matrix. Note that the minimum and maximum path lengths in this transition tree are one and that the transitions possible out of state i are those determined by alternative $a_0(i)$.

Now suppose that a transition tree of the required form exists for U-transitions determined by submatrix rows 1,...,I, i.e., out of states 1,...,IN. It is now shown that such a tree exists for transitions out of

any state i+IN; i=1,..,,N. First the general submatrix in row I+1 may be written as

$$U_{I+1,J} = \begin{cases} D_I(I)U_{1J}+D_I(I-1)U_{2J}+\cdots+D_I(1)U_{IJ} & J\leq I \\ D_I(L-J+I+1)+D_I(I)U_{1J}+D_I(I-1)U_{2J}+\cdots+D_I(1)U_{IJ}; & J>I. \end{cases}$$

Note that $D_I(K)$; $1\leq K\leq L$; appears either as a term by itself in some submatrix of the I+ 1 st row, or it appears in each submatrix of the row post-multiplied by a submatrix in the same column and in some specific previous row. Thus if $p_{ij}^{a_I(i)} > 0$ and $\phi_{ij}^{a_I(i)}(\ell) > 0$ for some $\ell > I$, then it is clear that transition is possible from i +IN to generic j. Otherwise $\phi_{ij}^{a_I(i)}(\ell) > 0$ for some $\ell \leq I$ and so there is a transition at level zero in the tree to j and then additional transitions from j determined by the sub-ma trices which multipy $D_I(\ell)$: $U_{I-\ell+1,1},\cdots, U_{I-\ell+1,L}$. This row of submatrices specifies a transition tree for the initial state j and so this tree extends the branch which already goes from i to j. Thus for any initial state i+IN, there is a corresponding transition tree of minimum length one, and so in N-1 or less U-transitions, generic m may be occupied which shows that m+(L-1)N may be occupied after N transitions.

Appendix F

CONVERGENCE OF GAIN RATE BOUNDS IN SCHEDULING MODEL

In Chapter 2 conditions were given which are sufficient to insure that the upper and lower bounds on the gain or gain rate converge. These bounds provide an upper limit to the gain or gain rate of the optimal policy and a lower limit to the gain or gain rate of the current policy in the MRDP. Recall that it was pointed out that the conditions given for convergence were sufficient but not necessary, and thus that the optimization method may be useful even when these conditions are not met. As a matter of fact, the conditions given in Chapter 2 which assure convergence of the bounds do not hold in general for the model developed in Chapter 3. This offers no difficulty, however, since the bounds did in fact converge in all cases. It is nonetheless of interest to show one set of restrictions on the model which assure convergence, and this is now done.

The first restriction is to remove from the model all alternatives which would allow no task to be loaded when in fact there is room in main memory for a task which is in the memory queue. Such alternatives may be removed from consideration without loss of generality because in the model the loading of the additional task must increase throughput and so the choice not to load is sub-optimal.

Next restrict the number of task types to two and the size of tasks relative to the memory size to be such that at least one half of the fixed

number of tasks in the model may occupy main memory simultaneously. Also restrict the normalized load times for all tasks to be one. With this last restriction, all transitions require the same amount of time, and so the problem of maximizing the gain rate is the same as the problem of maximizing gain. Therefore it is only necessary to exhibit a state reachable from all other states in some fixed number of transitions regardless of the sequence of policies chosen. Call this state m. Furthermore define N_0 as the maximum number of tasks which may occupy memory. The restriction above states that $N_0 \geq N/2$. Then the state m will be shown to be that state in which there are N_0 type 1 tasks in the memory queue and $N-N_0$ type 2 tasks in the memory queue.

Recall that between regeneration points any number (including zero) of the tasks in core may terminate and that this same number of tasks enter the memory queue. The type of task which enters the memory queue may, however, differ from the type of task which terminates. Then it is obvious that with N_0 tasks in core, state m is reachable in any finite number of transitions. That is, with probability greater than zero, no tasks terminate for n-1 transitions and then all N_0 tasks terminate at transition n for $n=1, 2, \ldots$ $(n<\infty)$. Furthermore with N_0 tasks in core, the number of type one tasks in the memory queue is less than or equal to N_0. Then after termination of all N_0 tasks in core, there will be N_0 type one tasks in the memory queue with probability greater than zero.

Next it is noted that after N_o or less transitions there may be N_o tasks in main memory because a task is always loaded when there is room in main memory for one. At worst there may be no tasks in memory in the initial state and then N_o transitions fill the memory with probability greater than zero because it is possible for no tasks to terminate between regeneration points.

From the preceding discussion it should now be clear that state m may be occupied after $N_o + 1$ transitions regardless of the initial state and regardless of the sequence of policies used. This is sufficient to assure convergence of the bounds on the gain rate.

Appendix G

SCHEDULING POLICIES

In this appendix, scheduling policies are given which maximized throughput for the scheduling model parameters of Chapter 4. Each table in this appendix contains one scheduling policy and refers back to the result in Chapter 4 obtained by using this policy. Each table presents all states of the model and the memory scheduling and CPU scheduling decision to be used in each state. The state is specified by giving the number of type one and type two tasks in the memory queue, at the CPU, and in I/O. The scheduling decision is denoted by A1 and A4 where A1 denotes the memory scheduling decision.

$$A1 = \begin{cases} 0 \text{ means no load} \\ 1 \text{ means load a type 1 task} \\ 2 \text{ means load a type 2 task} \end{cases}$$

A4 denotes the CPU scheduling decision as follows.

$$A4 = \begin{cases} 12 \text{ means type 1 tasks have preemptive priority at the CPU} \\ 21 \text{ means type 2 tasks have preemptive priority at the CPU} \end{cases}$$

Note that in some states there actually is no choice to be made; for example, only one decision is possible when memory is filled with a single type of task.

Table G. 1

Scheduling Policy For Figure 4. 5 (70 Pages, No Sharing)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
6	0	0	0	0	0	1	12
5	1	0	0	0	0	1	12
4	2	0	0	0	0	1	12
3	3	0	0	0	0	1	12
2	4	0	0	0	0	1	12
1	5	0	0	0	0	1	12
0	6	0	0	0	0	2	12
5	0	0	0	0	1	1	21
4	1	0	0	0	1	1	21
3	2	0	0	0	1	1	21
2	3	0	0	0	1	1	21
1	4	0	0	0	1	1	21
0	5	0	0	0	1	2	21
4	0	0	0	0	2	1	21
3	1	0	0	0	2	1	21
2	2	0	0	0	2	1	21
1	3	0	0	0	2	1	21
0	4	0	0	0	2	2	21
3	0	0	0	0	3	0	21
2	1	0	0	0	3	0	21
1	2	0	0	0	3	0	21
0	3	0	0	0	3	0	21
5	0	0	0	1	0	1	12
4	1	0	0	1	0	1	12
3	2	0	0	1	0	1	12
2	3	0	0	1	0	2	12
1	4	0	0	1	0	2	12
0	5	0	0	1	0	2	12
4	0	0	0	1	1	0	12
3	1	0	0	1	1	2	12
2	2	0	0	1	1	2	12
1	3	0	0	1	1	2	12
0	4	0	0	1	1	2	12
3	0	0	0	1	2	0	12
2	1	0	0	1	2	0	12
1	2	0	0	1	2	0	21
0	3	0	0	1	2	0	21
4	0	0	0	2	0	0	12
3	1	0	0	2	0	0	12
2	2	0	0	2	0	0	12

Table G. 1 (continued)

1	3	0	0	2	0	0	12
0	4	0	0	2	0	0	12
5	0	0	1	0	0	1	21
4	1	0	1	0	0	1	21
3	2	0	1	0	0	1	21
2	3	0	1	0	0	1	21
1	4	0	1	0	0	1	21
0	5	0	1	0	0	2	21
4	0	0	1	0	1	1	21
3	1	0	1	0	1	1	21
2	2	0	1	0	1	1	21
1	3	0	1	0	1	1	21
0	4	0	1	0	1	2	21
3	0	0	1	0	2	0	21
2	1	0	1	0	2	0	21
1	2	0	1	0	2	0	21
0	3	0	1	0	2	0	21
4	0	0	1	1	0	0	12
3	1	0	1	1	0	2	12
2	2	0	1	1	0	2	12
1	3	0	1	1	0	2	12
0	4	0	1	1	0	2	12
3	0	0	1	1	1	0	12
2	1	0	1	1	1	0	12
1	2	0	1	1	1	0	21
0	3	0	1	1	1	0	21
4	0	0	2	0	0	1	21
3	1	0	2	0	0	1	21
2	2	0	2	0	0	1	21
1	3	0	2	0	0	1	21
0	4	0	2	0	0	2	21
3	0	0	2	0	1	0	21
2	1	0	2	0	1	0	21
1	2	0	2	0	1	0	21
0	3	0	2	0	1	0	21
3	0	0	2	1	0	0	12
2	1	0	2	1	0	0	12
1	2	0	2	1	0	0	21
0	3	0	2	1	0	0	21
3	0	0	3	0	0	0	21
2	1	0	3	0	0	0	21
1	2	0	3	0	0	0	21
0	3	0	3	0	0	0	21
5	0	1	0	0	0	1	12
4	1	1	0	0	0	1	12
3	2	1	0	0	0	1	12

Table G. 1 (continued)

2	3	1	0	0	0	2	12
1	4	1	0	0	0	2	12
0	5	1	0	0	0	2	12
4	0	1	0	0	1	0	12
3	1	1	0	0	1	2	12
2	2	1	0	0	1	2	12
1	3	1	0	0	1	2	12
0	4	1	0	0	1	2	12
3	0	1	0	0	2	0	12
2	1	1	0	0	2	0	12
1	2	1	0	0	2	0	21
0	3	1	0	0	2	0	21
4	0	1	0	1	0	0	12
3	1	1	0	1	0	0	12
2	2	1	0	1	0	0	12
1	3	1	0	1	0	0	12
0	4	1	0	1	0	0	12
4	0	1	1	0	0	0	12
3	1	1	1	0	0	2	12
2	2	1	1	0	0	2	12
1	3	1	1	0	0	2	12
0	4	1	1	0	0	2	12
3	0	1	1	0	1	0	12
2	1	1	1	0	1	0	12
1	2	1	1	0	1	0	21
0	3	1	1	0	1	0	21
3	0	1	2	0	0	0	12
2	1	1	2	0	0	0	12
1	2	1	2	0	0	0	21
0	3	1	2	0	0	0	21
4	0	2	0	0	0	0	12
3	1	2	0	0	0	0	12
2	2	2	0	0	0	0	12
1	3	2	0	0	0	0	12
0	4	2	0	0	0	0	12

Table G. 2

Scheduling Policy For Figure 4. 5 (70 Pages, Sharing)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
6	0	0	0	0	0	1	12
5	1	0	0	0	0	1	12
4	2	0	0	0	0	1	12
3	3	0	0	0	0	1	12
2	4	0	0	0	0	1	12
1	5	0	0	0	0	1	12
0	6	0	0	0	0	2	12
5	0	0	0	0	1	1	21
4	1	0	0	0	1	1	21
3	2	0	0	0	1	1	21
2	3	0	0	0	1	1	21
1	4	0	0	0	1	1	21
0	5	0	0	0	1	2	21
4	0	0	0	0	2	1	21
3	1	0	0	0	2	1	21
2	2	0	0	0	2	1	21
1	3	0	0	0	2	2	21
0	4	0	0	0	2	2	21
3	0	0	0	0	3	0	21
2	1	0	0	0	3	0	21
1	2	0	0	0	3	0	21
0	3	0	0	0	3	0	21
5	0	0	0	1	0	1	12
4	1	0	0	1	0	1	12
3	2	0	0	1	0	1	12
2	3	0	0	1	0	1	12
1	4	0	0	1	0	1	12
0	5	0	0	1	0	2	12
4	0	0	0	1	1	1	12
3	1	0	0	1	1	1	12
2	2	0	0	1	1	1	12
1	3	0	0	1	1	1	12
0	4	0	0	1	1	2	12
3	0	0	0	1	2	0	21
2	1	0	0	1	2	0	21
1	2	0	0	1	2	0	21
0	3	0	0	1	2	0	21
4	0	0	0	2	0	1	12
3	1	0	0	2	0	2	12

Table G. 2 (continued)

2	2	0	0	2	0	2	12
1	3	0	0	2	0	2	12
0	4	0	0	2	0	2	12
3	0	0	0	2	1	0	21
2	1	0	0	2	1	0	21
1	2	0	0	2	1	0	21
0	3	0	0	2	1	0	21
3	0	0	0	3	0	0	12
2	1	0	0	3	0	0	12
1	2	0	0	3	0	0	12
0	3	0	0	3	0	0	12
5	0	0	1	0	0	1	21
4	1	0	1	0	0	1	21
3	2	0	1	0	0	1	21
2	3	0	1	0	0	1	21
1	4	0	1	0	0	1	21
0	5	0	1	0	0	2	21
4	0	0	1	0	1	1	21
3	1	0	1	0	1	1	21
2	2	0	1	0	1	1	21
1	3	0	1	0	1	2	21
0	4	0	1	0	1	2	21
3	0	0	1	0	2	0	21
2	1	0	1	0	2	0	21
1	2	0	1	0	2	0	21
0	3	0	1	0	2	0	21
4	0	0	1	1	0	1	12
3	1	0	1	1	0	1	12
2	2	0	1	1	0	1	12
1	3	0	1	1	0	1	12
0	4	0	1	1	0	2	12
3	0	0	1	1	1	0	21
2	1	0	1	1	1	0	21
1	2	0	1	1	1	0	21
0	3	0	1	1	1	0	21
3	0	0	1	2	0	0	21
2	1	0	1	2	0	0	21
1	2	0	1	2	0	0	21
0	3	0	1	2	0	0	21
4	0	0	2	0	0	1	21
3	1	0	2	0	0	1	21
2	2	0	2	0	0	1	21
1	3	0	2	0	0	2	21
0	4	0	2	0	0	2	21
3	0	0	2	0	1	0	21

Table G. 2 (continued)

2	1	0	2	0	1	0	21
1	2	0	2	0	1	0	21
0	3	0	2	0	1	0	21
3	0	0	2	1	0	0	21
2	1	0	2	1	0	0	21
1	2	0	2	1	0	0	21
0	3	0	2	1	0	0	21
3	0	0	3	0	0	0	21
2	1	0	3	0	0	0	21
1	2	0	3	0	0	0	21
0	3	0	3	0	0	0	21
5	0	1	0	0	0	1	12
4	1	1	0	0	0	1	12
3	2	1	0	0	0	1	12
2	3	1	0	0	0	1	12
1	4	1	0	0	0	1	12
0	5	1	0	0	0	2	12
4	0	1	0	0	1	1	12
3	1	1	0	0	1	1	12
2	2	1	0	0	1	1	12
1	3	1	0	0	1	1	12
0	4	1	0	0	1	2	12
3	0	1	0	0	2	0	21
2	1	1	0	0	2	0	21
1	2	1	0	0	2	0	21
0	3	1	0	0	2	0	21
4	0	1	0	1	0	1	12
3	1	1	0	1	0	2	12
2	2	1	0	1	0	2	12
1	3	1	0	1	0	2	12
0	4	1	0	1	0	2	12
3	0	1	0	1	1	0	21
2	1	1	0	1	1	0	21
1	2	1	0	1	1	0	21
0	3	1	0	1	1	0	21
3	0	1	0	2	0	0	12
2	1	1	0	2	0	0	12
1	2	1	0	2	0	0	12
0	3	1	0	2	0	0	12
4	0	1	1	0	0	1	12
3	1	1	1	0	0	1	12
2	2	1	1	0	0	1	12
1	3	1	1	0	0	1	12
0	4	1	1	0	0	2	12
3	0	1	1	0	1	0	21

Table G. 2 (continued)

2	1	1	1	0	1	0	21
1	2	1	1	0	1	0	21
0	3	1	1	0	1	0	21
3	0	1	1	1	0	0	21
2	1	1	1	1	0	0	21
1	2	1	1	1	0	0	21
0	3	1	1	1	0	0	21
3	0	1	2	0	0	0	21
2	1	1	2	0	0	0	21
1	2	1	2	0	0	0	21
0	3	1	2	0	0	0	21
4	0	2	0	0	0	1	12
3	1	2	0	0	0	2	12
2	2	2	0	0	0	2	12
1	3	2	0	0	0	2	12
0	4	2	0	0	0	2	12
3	0	2	0	0	1	0	21
2	1	2	0	0	1	0	21
1	2	2	0	0	1	0	21
0	3	2	0	0	1	0	21
3	0	2	0	1	0	0	12
2	1	2	0	1	0	0	12
1	2	2	0	1	0	0	12
0	3	2	0	1	0	0	12
3	0	2	1	0	0	0	21
2	1	2	1	0	0	0	21
1	2	2	1	0	0	0	21
0	3	2	1	0	0	0	21
3	0	3	0	0	0	0	12
2	1	3	0	0	0	0	12
1	2	3	0	0	0	0	12
0	3	3	0	0	0	0	12

Table G. 3

Scheduling Policy For Figure 4. 5 (90 Pages, No Sharing)

and Figure 4. 6 (2)

STATE				DECISION	
MEM	CPU	I/O			
1 2	1 2	1 2		A1	A4
6 0	0 0	0 0		1	12
5 1	0 0	0 0		1	12
4 2	0 0	0 0		1	12
3 3	0 0	0 0		1	12
2 4	0 0	0 0		1	12
1 5	0 0	0 0		1	12
0 6	0 0	0 0		2	12
5 0	0 0	0 1		1	21
4 1	0 0	0 1		1	21
3 2	0 0	0 1		1	21
2 3	0 0	0 1		1	21
1 4	0 0	0 1		1	21
0 5	0 0	0 1		2	21
4 0	0 0	0 2		1	21
3 1	0 0	0 2		1	21
2 2	0 0	0 2		1	21
1 3	0 0	0 2		1	21
0 4	0 0	0 2		2	21
3 0	0 0	0 3		0	21
2 1	0 0	0 3		2	21
1 2	0 0	0 3		2	21
0 3	0 0	0 3		2	21
2 0	0 0	0 4		0	21
1 1	0 0	0 4		0	21
0 2	0 0	0 4		0	21
5 0	0 0	1 0		1	12
4 1	0 0	1 0		1	12
3 2	0 0	1 0		1	12
2 3	0 0	1 0		1	12
1 4	0 0	1 0		1	12
0 5	0 0	1 0		2	12
4 0	0 0	1 1		1	12
3 1	0 0	1 1		1	12
2 2	0 0	1 1		1	12
1 3	0 0	1 1		1	12
0 4	0 0	1 1		2	12
3 0	0 0	1 2		0	21
2 1	0 0	1 2		0	21
1 2	0 0	1 2		0	21

Table G. 3 (continued)

0	3	0	0	1	2	0	12
4	0	0	0	2	0	1	12
3	1	0	0	2	0	1	12
2	2	0	0	2	0	1	12
1	3	0	0	2	0	2	12
0	4	0	0	2	0	2	12
3	0	0	0	2	1	0	21
2	1	0	0	2	1	0	21
1	2	0	0	2	1	0	21
0	3	0	0	2	1	0	21
3	0	0	0	3	0	0	12
2	1	0	0	3	0	0	12
1	2	0	0	3	0	0	12
0	3	0	0	3	0	0	12
5	0	0	1	0	0	1	21
4	1	0	1	0	0	1	21
3	2	0	1	0	0	1	21
2	3	0	1	0	0	1	21
1	4	0	1	0	0	1	21
0	5	0	1	0	0	2	21
4	0	0	1	0	1	1	21
3	1	0	1	0	1	1	21
2	2	0	1	0	1	1	21
1	3	0	1	0	1	1	21
0	4	0	1	0	1	2	21
3	0	0	1	0	2	0	21
2	1	0	1	0	2	2	21
1	2	0	1	0	2	2	21
0	3	0	1	0	2	2	21
2	0	0	1	0	3	0	21
1	1	0	1	0	3	0	21
0	2	0	1	0	3	0	21
4	0	0	1	1	0	1	12
3	1	0	1	1	0	1	12
2	2	0	1	1	0	1	12
1	3	0	1	1	0	1	12
0	4	0	1	1	0	2	12
3	0	0	1	1	1	0	21
2	1	0	1	1	1	0	21
1	2	0	1	1	1	0	21
0	3	0	1	1	1	0	12
3	0	0	1	2	0	0	21
2	1	0	1	2	0	0	21
1	2	0	1	2	0	0	21
0	3	0	1	2	0	0	21
4	0	0	2	0	0	1	21

Table G. 3 (continued)

3	1	0	2	0	0	1	21
2	2	0	2	0	0	1	21
1	3	0	2	0	0	1	21
0	4	0	2	0	0	2	21
3	0	0	2	0	1	0	21
2	1	0	2	0	1	2	21
1	2	0	2	0	1	2	21
0	3	0	2	0	1	2	21
2	0	0	2	0	2	0	21
1	1	0	2	0	2	0	21
0	2	0	2	0	2	0	21
3	0	0	2	1	0	0	21
2	1	0	2	1	0	0	21
1	2	0	2	1	0	0	21
0	3	0	2	1	0	0	12
3	0	0	3	0	0	0	21
2	1	0	3	0	0	2	21
1	2	0	3	0	0	2	21
0	3	0	3	0	0	2	21
2	0	0	3	0	1	0	21
1	1	0	3	0	1	0	21
0	2	0	3	0	1	0	21
2	0	0	4	0	0	0	21
1	1	0	4	0	0	0	21
0	2	0	4	0	0	0	21
5	0	1	0	0	0	1	12
4	1	1	0	0	0	1	12
3	2	1	0	0	0	1	12
2	3	1	0	0	0	1	12
1	4	1	0	0	0	1	12
0	5	1	0	0	0	2	12
4	0	1	0	0	1	1	12
3	1	1	0	0	1	1	12
2	2	1	0	0	1	1	12
1	3	1	0	0	1	1	12
0	4	1	0	0	1	2	12
3	0	1	0	0	2	0	21
2	1	1	0	0	2	0	21
1	2	1	0	0	2	0	21
0	3	1	0	0	2	0	12
4	0	1	0	1	0	1	12
3	1	1	0	1	0	1	12
2	2	1	0	1	0	1	12
1	3	1	0	1	0	2	12
0	4	1	0	1	0	2	12
3	0	1	0	1	1	0	21

Table G. 3 (continued)

2	1	1	0	1	1	0	21
1	2	1	0	1	1	0	21
0	3	1	0	1	1	0	21
3	0	1	0	2	0	0	12
2	1	1	0	2	0	0	12
1	2	1	0	2	0	0	12
0	3	1	0	2	0	0	12
4	0	1	1	0	0	1	12
3	1	1	1	0	0	1	12
2	2	1	1	0	0	1	12
1	3	1	1	0	0	1	12
0	4	1	1	0	0	2	12
3	0	1	1	0	1	0	21
2	1	1	1	0	1	0	21
1	2	1	1	0	1	0	21
0	3	1	1	0	1	0	12
3	0	1	1	1	0	0	21
2	1	1	1	1	0	0	21
1	2	1	1	1	0	0	21
0	3	1	1	1	0	0	21
3	0	1	2	0	0	0	21
2	1	1	2	0	0	0	21
1	2	1	2	0	0	0	21
0	3	1	2	0	0	0	12
4	0	2	0	0	0	1	12
3	1	2	0	0	0	1	12
2	2	2	0	0	0	1	12
1	3	2	0	0	0	2	12
0	4	2	0	0	0	2	12
3	0	2	0	0	1	0	21
2	1	2	0	0	1	0	21
1	2	2	0	0	1	0	21
0	3	2	0	0	1	0	21
3	0	2	0	1	0	0	12
2	1	2	0	1	0	0	12
1	2	2	0	1	0	0	12
0	3	2	0	1	0	0	12
3	0	2	1	0	0	0	21
2	1	2	1	0	0	0	21
1	2	2	1	0	0	0	21
0	3	2	1	0	0	0	21
3	0	3	0	0	0	0	12
2	1	3	0	0	0	0	12
1	2	3	0	0	0	0	12
0	3	3	0	0	0	0	12

Table G. 4

Scheduling Policy For Figure 4. 5 (90 Pages, Sharing)

and Figure 4. 6 (4)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
6	0	0	0	0	0	1	12
5	1	0	0	0	0	1	12
4	2	0	0	0	0	1	12
3	3	0	0	0	0	1	12
2	4	0	0	0	0	1	12
1	5	0	0	0	0	2	12
0	6	0	0	0	0	2	12
5	0	0	0	0	1	1	21
4	1	0	0	0	1	1	21
3	2	0	0	0	1	1	21
2	3	0	0	0	1	1	21
1	4	0	0	0	1	2	21
0	5	0	0	0	1	2	21
4	0	0	0	0	2	1	21
3	1	0	0	0	2	1	21
2	2	0	0	0	2	1	21
1	3	0	0	0	2	2	21
0	4	0	0	0	2	2	21
3	0	0	0	0	3	0	21
2	1	0	0	0	3	2	21
1	2	0	0	0	3	2	21
0	3	0	0	0	3	2	21
2	0	0	0	0	4	0	21
1	1	0	0	0	4	0	21
0	2	0	0	0	4	0	21
5	0	0	0	1	0	1	12
4	1	0	0	1	0	1	12
3	2	0	0	1	0	1	12
2	3	0	0	1	0	1	12
1	4	0	0	1	0	1	12
0	5	0	0	1	0	2	12
4	0	0	0	1	1	1	12
3	1	0	0	1	1	1	12
2	2	0	0	1	1	1	12
1	3	0	0	1	1	1	12
0	4	0	0	1	1	2	12
3	0	0	0	1	2	1	21
2	1	0	0	1	2	1	21
1	2	0	0	1	2	1	21

Table G. 4 (continued)

0	3	0	0	1	2	0	12
4	0	0	0	2	0	1	12
3	1	0	0	2	0	1	12
2	2	0	0	2	0	2	12
1	3	0	0	2	0	2	12
0	4	0	0	2	0	2	12
3	0	0	0	2	1	1	21
2	1	0	0	2	1	2	21
1	2	0	0	2	1	2	21
0	3	0	0	2	1	2	21
2	0	0	0	2	2	0	21
1	1	0	0	2	2	0	21
0	2	0	0	2	2	0	21
3	0	0	0	3	0	1	12
2	1	0	0	3	0	2	12
1	2	0	0	3	0	2	12
0	3	0	0	3	0	2	12
2	0	0	0	3	1	0	21
1	1	0	0	3	1	0	21
0	2	0	0	3	1	0	21
2	0	0	0	4	0	0	12
1	1	0	0	4	0	0	12
0	2	0	0	4	0	0	12
5	0	0	1	0	0	1	21
4	1	0	1	0	0	1	21
3	2	0	1	0	0	1	21
2	3	0	1	0	0	1	21
1	4	0	1	0	0	2	21
0	5	0	1	0	0	2	21
4	0	0	1	0	1	1	21
3	1	0	1	0	1	1	21
2	2	0	1	0	1	1	21
1	3	0	1	0	1	2	21
0	4	0	1	0	1	2	21
3	0	0	1	0	2	0	21
2	1	0	1	0	2	2	21
1	2	0	1	0	2	2	21
0	3	0	1	0	2	2	21
2	0	0	1	0	3	0	21
1	1	0	1	0	3	0	21
0	2	0	1	0	3	0	21
4	0	0	1	1	0	1	12
3	1	0	1	1	0	1	12
2	2	0	1	1	0	1	12
1	3	0	1	1	0	1	12

Table G. 4 (continued)

0	4	0	1	1	0	2	12
3	0	0	1	1	1	1	21
2	1	0	1	1	1	1	21
1	2	0	1	1	1	1	21
0	3	0	1	1	1	0	12
3	0	0	1	2	0	1	21
2	1	0	1	2	0	2	21
1	2	0	1	2	0	2	21
0	3	0	1	2	0	2	21
2	0	0	1	2	1	0	21
1	1	0	1	2	1	0	21
0	2	0	1	2	1	0	21
2	0	0	1	3	0	0	21
1	1	0	1	3	0	0	21
0	2	0	1	3	0	0	21
4	0	0	2	0	0	1	21
3	1	0	2	0	0	1	21
2	2	0	2	0	0	1	21
1	3	0	2	0	0	2	21
0	4	0	2	0	0	2	21
3	0	0	2	0	1	0	21
2	1	0	2	0	1	2	21
1	2	0	2	0	1	2	21
0	3	0	2	0	1	2	21
2	0	0	2	0	2	0	21
1	1	0	2	0	2	0	21
0	2	0	2	0	2	0	21
3	0	0	2	1	0	1	21
2	1	0	2	1	0	1	21
1	2	0	2	1	0	1	21
0	3	0	2	1	0	0	12
2	0	0	2	2	0	0	21
1	1	0	2	2	0	0	21
0	2	0	2	2	0	0	21
3	0	0	3	0	0	0	21
2	1	0	3	0	0	2	21
1	2	0	3	0	0	2	21
0	3	0	3	0	0	2	21
2	0	0	3	0	1	0	21
1	1	0	3	0	1	0	21
0	2	0	3	0	1	0	21
2	0	0	4	0	0	0	21
1	1	0	4	0	0	0	21
0	2	0	4	0	0	0	21
5	0	1	0	0	0	1	12

Table G. 4 (continued)

4	1	1	0	0	0	1	12
3	2	1	0	0	0	1	12
2	3	1	0	0	0	1	12
1	4	1	0	0	0	1	12
0	5	1	0	0	0	2	12
4	0	1	0	0	1	1	12
3	1	1	0	0	1	1	12
2	2	1	0	0	1	1	12
1	3	1	0	0	1	1	12
0	4	1	0	0	1	2	12
3	0	1	0	0	2	1	21
2	1	1	0	0	2	1	21
1	2	1	0	0	2	1	21
0	3	1	0	0	2	0	12
4	0	1	0	1	0	1	12
3	1	1	0	1	0	1	12
2	2	1	0	1	0	2	12
1	3	1	0	1	0	2	12
0	4	1	0	1	0	2	12
3	0	1	0	1	1	1	21
2	1	1	0	1	1	2	21
1	2	1	0	1	1	2	21
0	3	1	0	1	1	2	21
2	0	1	0	1	2	0	21
1	1	1	0	1	2	0	21
0	2	1	0	1	2	0	21
3	0	1	0	2	0	1	12
2	1	1	0	2	0	2	12
1	2	1	0	2	0	2	12
0	3	1	0	2	0	2	12
2	0	1	0	2	1	0	21
1	1	1	0	2	1	0	21
0	2	1	0	2	1	0	21
2	0	1	0	3	0	0	12
1	1	1	0	3	0	0	12
0	2	1	0	3	0	0	12
4	0	1	1	0	0	1	12
3	1	1	1	0	0	1	12
2	2	1	1	0	0	1	12
1	3	1	1	0	0	1	12
0	4	1	1	0	0	2	12
3	0	1	1	0	1	1	21
2	1	1	1	0	1	1	21
1	2	1	1	0	1	1	21
Q	3	1	1	0	1	0	12

Table G. 4 (continued)

3	0	1	1	1	0	1	21
2	1	1	1	1	0	2	21
1	2	1	1	1	0	2	21
0	3	1	1	1	0	2	21
2	0	1	1	1	1	0	21
1	1	1	1	1	1	0	21
0	2	1	1	1	1	0	21
2	0	1	1	2	0	0	21
1	1	1	1	2	0	0	21
0	2	1	1	2	0	0	21
3	0	1	2	0	0	1	21
2	1	1	2	0	0	1	21
1	2	1	2	0	0	1	21
0	3	1	2	0	0	0	12
2	0	1	2	1	0	0	21
1	1	1	2	1	0	0	21
0	2	1	2	1	0	0	21
4	0	2	0	0	0	1	12
3	1	2	0	0	0	1	12
2	2	2	0	0	0	2	12
1	3	2	0	0	0	2	12
0	4	2	0	0	0	2	12
3	0	2	0	0	1	1	21
2	1	2	0	0	1	2	21
1	2	2	0	0	1	2	21
0	3	2	0	0	1	2	21
2	0	2	0	0	2	0	21
1	1	2	0	0	2	0	21
0	2	2	0	0	2	0	21
3	0	2	0	1	0	1	12
2	1	2	0	1	0	2	12
1	2	2	0	1	0	2	12
0	3	2	0	1	0	2	12
2	0	2	0	1	1	0	21
1	1	2	0	1	1	0	21
0	2	2	0	1	1	0	21
2	0	2	0	2	0	0	12
1	1	2	0	2	0	0	12
0	2	2	0	2	0	0	12
3	0	2	1	0	0	1	21
2	1	2	1	0	0	2	21
1	2	2	1	0	0	2	21
0	3	2	1	0	0	2	21
2	0	2	1	0	1	0	21
1	1	2	1	0	1	0	21

Table G. 4 (continued)

0 2	2 1	0 1	0 21
2 0	2 1	1 0	0 21
1 1	2 1	1 0	0 21
0 2	2 1	1 0	0 21
2 0	2 2	0 0	0 21
1 1	2 2	0 0	0 21
0 2	2 2	0 0	0 21
3 0	3 0	0 0	1 12
2 1	3 0	0 0	2 12
1 2	3 0	0 0	2 12
0 3	3 0	0 0	2 12
2 0	3 0	0 1	0 21
1 1	3 0	0 1	0 21
0 2	3 0	0 1	0 21
2 0	3 0	1 0	0 12
1 1	3 0	1 0	0 12
0 2	3 0	1 0	0 12
2 0	3 1	0 0	0 21
1 1	3 1	0 0	0 21
0 2	3 1	0 0	0 21
2 0	4 0	0 0	0 12
1 1	4 0	0 0	0 12
0 2	4 0	0 0	0 12

Table G. 5

Scheduling Policy For Figure 4. 6 (6)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
6	0	0	0	0	0	1	12
5	1	0	0	0	0	1	12
4	2	0	0	0	0	1	12
3	3	0	0	0	0	1	12
2	4	0	0	0	0	1	12
1	5	0	0	0	0	1	12
0	6	0	0	0	0	2	12
5	0	0	0	0	1	1	21
4	1	0	0	0	1	1	21
3	2	0	0	0	1	1	21
2	3	0	0	0	1	1	21
1	4	0	0	0	1	1	21
0	5	0	0	0	1	2	21
4	0	0	0	0	2	1	21
3	1	0	0	0	2	1	21
2	2	0	0	0	2	1	21
1	3	0	0	0	2	1	21
0	4	0	0	0	2	2	21
3	0	0	0	0	3	0	21
2	1	0	0	0	3	2	21
1	2	0	0	0	3	2	21
0	3	0	0	0	3	2	21
2	0	0	0	0	4	0	21
1	1	0	0	0	4	0	21
0	2	0	0	0	4	0	21
5	0	0	0	1	0	1	12
4	1	0	0	1	0	1	12
3	2	0	0	1	0	1	12
2	3	0	0	1	0	1	12
1	4	0	0	1	0	1	12
0	5	0	0	1	0	2	12
4	0	0	0	1	1	1	12
3	1	0	0	1	1	1	12
2	2	0	0	1	1	1	12
1	3	0	0	1	1	1	12
0	4	0	0	1	1	2	12
3	0	0	0	1	2	0	12
2	1	0	0	1	2	0	12
1	2	0	0	1	2	0	12
0	3	0	0	1	2	0	12

Table G. 5 (continued)

4	0	0	0	2	0	1	12
3	1	0	0	2	0	1	12
2	2	0	0	2	0	1	12
1	3	0	0	2	0	1	12
0	4	0	0	2	0	2	12
3	0	0	0	2	1	0	21
2	1	0	0	2	1	0	21
1	2	0	0	2	1	0	12
0	3	0	0	2	1	0	12
3	0	0	0	3	0	0	12
2	1	0	0	3	0	0	12
1	2	0	0	3	0	0	12
0	3	0	0	3	0	0	12
5	0	0	1	0	0	1	21
4	1	0	1	0	0	1	21
3	2	0	1	0	0	1	21
2	3	0	1	0	0	1	21
1	4	0	1	0	0	1	21
0	5	0	1	0	0	2	21
4	0	0	1	0	1	1	21
3	1	0	1	0	1	1	21
2	2	0	1	0	1	1	21
1	3	0	1	0	1	1	21
0	4	0	1	0	1	2	21
3	0	0	1	0	2	0	21
2	1	0	1	0	2	2	21
1	2	0	1	0	2	2	21
0	3	0	1	0	2	2	21
2	0	0	1	0	3	0	21
1	1	0	1	0	3	0	21
0	2	0	1	0	3	0	21
4	0	0	1	1	0	1	12
3	1	0	1	1	0	1	12
2	2	0	1	1	0	1	12
1	3	0	1	1	0	1	12
0	4	0	1	1	0	2	12
3	0	0	1	1	1	0	12
2	1	0	1	1	1	0	12
1	2	0	1	1	1	0	12
0	3	0	1	1	1	0	12
3	0	0	1	2	0	0	21
2	1	0	1	2	0	0	21
1	2	0	1	2	0	0	12
0	3	0	1	2	0	0	12
4	0	0	2	0	0	1	21
3	1	0	2	0	0	1	21

Table G. 5 (continued)

2	2	0	2	0	0	1	21
1	3	0	2	0	0	1	21
0	4	0	2	0	0	2	21
3	0	0	2	0	1	0	21
2	1	0	2	0	1	2	21
1	2	0	2	0	1	2	21
0	3	0	2	0	1	2	21
2	0	0	2	0	2	0	21
1	1	0	2	0	2	0	21
0	2	0	2	0	2	0	21
3	0	0	2	1	0	0	12
2	1	0	2	1	0	0	12
1	2	0	2	1	0	0	12
0	3	0	2	1	0	0	12
3	0	0	3	0	0	0	21
2	1	0	3	0	0	2	21
1	2	0	3	0	0	2	21
0	3	0	3	0	0	2	21
2	0	0	3	0	1	0	21
1	1	0	3	0	1	0	21
0	2	0	3	0	1	0	21
2	0	0	4	0	0	0	21
1	1	0	4	0	0	0	21
0	2	0	4	0	0	0	21
5	0	1	0	0	0	1	12
4	1	1	0	0	0	1	12
3	2	1	0	0	0	1	12
2	3	1	0	0	0	1	12
1	4	1	0	0	0	1	12
0	5	1	0	0	0	2	12
4	0	1	0	0	1	1	12
3	1	1	0	0	1	1	12
2	2	1	0	0	1	1	12
1	3	1	0	0	1	1	12
0	4	1	0	0	1	2	12
3	0	1	0	0	2	0	12
2	1	1	0	0	2	0	12
1	2	1	0	0	2	0	12
0	3	1	0	0	2	0	12
4	0	1	0	1	0	1	12
3	1	1	0	1	0	1	12
2	2	1	0	1	0	1	12
1	3	1	0	1	0	1	12
0	4	1	0	1	0	2	12
3	0	1	0	1	1	0	21
2	1	1	0	1	1	0	21

Table G. 5 (continued)

1	2	1	0	1	1	0	12
0	3	1	0	1	1	0	12
3	0	1	0	2	0	0	12
2	1	1	0	2	0	0	12
1	2	1	0	2	0	0	12
0	3	1	0	2	0	0	12
4	0	1	1	0	0	1	12
3	1	1	1	0	0	1	12
2	2	1	1	0	0	1	12
1	3	1	1	0	0	1	12
0	4	1	1	0	0	2	12
3	0	1	1	0	1	0	12
2	1	1	1	0	1	0	12
1	2	1	1	0	1	0	12
0	3	1	1	0	1	0	12
3	0	1	1	1	0	0	21
2	1	1	1	1	0	0	21
1	2	1	1	1	0	0	12
0	3	1	1	1	0	0	12
3	0	1	2	0	0	0	12
2	1	1	2	0	0	0	12
1	2	1	2	0	0	0	12
0	3	1	2	0	0	0	12
4	0	2	0	0	0	1	12
3	1	2	0	0	0	1	12
2	2	2	0	0	0	1	12
1	3	2	0	0	0	1	12
0	4	2	0	0	0	2	12
3	0	2	0	0	1	0	21
2	1	2	0	0	1	0	21
1	2	2	0	0	1	0	12
0	3	2	0	0	1	0	12
3	0	2	0	1	0	0	12
2	1	2	0	1	0	0	12
1	2	2	0	1	0	0	12
0	3	2	0	1	0	0	12
3	0	2	1	0	0	0	21
2	1	2	1	0	0	0	21
1	2	2	1	0	0	0	21
0	3	2	1	0	0	0	12
3	0	3	0	0	0	0	12
2	1	3	0	0	0	0	12
1	2	3	0	0	0	0	12
0	3	3	0	0	0	0	12

Table G. 6

Scheduling Policy For Figure 4. 10 (1, 40 Pages)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	1	12
2	3	0	0	0	0	1	12
1	4	0	0	0	0	1	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	0	21
3	1	0	0	0	1	0	21
2	2	0	0	0	1	0	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	0	21
1	2	0	0	0	2	0	21
0	3	0	0	0	2	0	21
4	0	0	0	1	0	0	12
3	1	0	0	1	0	0	12
2	2	0	0	1	0	0	12
1	3	0	0	1	0	0	12
0	4	0	0	1	0	0	12
4	0	0	1	0	0	0	21
3	1	0	1	0	0	2	21
2	2	0	1	0	0	2	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	0	21
1	2	0	1	0	1	0	21
0	3	0	1	0	1	0	21
3	0	0	2	0	0	0	21
2	1	0	2	0	0	0	21
1	2	0	2	0	0	0	21
0	3	0	2	0	0	0	21
4	0	1	0	0	0	0	12
3	1	1	0	0	0	0	12
2	2	1	0	0	0	0	12
1	3	1	0	0	0	0	12
0	4	1	0	0	0	0	12

Table G. 7

Scheduling Policy For Figure 4. 10 (1, 50 Pages)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	2	12
2	3	0	0	0	0	2	12
1	4	0	0	0	0	2	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	0	21
3	1	0	0	0	1	2	21
2	2	0	0	0	1	2	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	0	21
1	2	0	0	0	2	0	21
0	3	0	0	0	2	0	21
4	0	0	0	1	0	1	12
3	1	0	0	1	0	1	12
2	2	0	0	1	0	1	12
1	3	0	0	1	0	0	12
0	4	0	0	1	0	0	12
3	0	0	0	2	0	0	12
2	1	0	0	2	0	0	12
1	2	0	0	2	0	0	12
0	3	0	0	2	0	0	12
4	0	0	1	0	0	0	21
3	1	0	1	0	0	2	21
2	2	0	1	0	0	2	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	0	21
1	2	0	1	0	1	0	21
0	3	0	1	0	1	0	21
3	0	0	2	0	0	0	21
2	1	0	2	0	0	0	21
1	2	0	2	0	0	0	21
0	3	0	2	0	0	0	21

Table G. 7 (continued)

4	0	1	0	0	0	1	12
3	1	1	0	0	0	1	12
2	2	1	0	0	0	1	12
1	3	1	0	0	0	1	12
0	4	1	0	0	0	0	12
3	0	1	0	1	0	0	12
2	1	1	0	1	0	0	12
1	2	1	0	1	0	0	12
0	3	1	0	1	0	0	12
3	0	2	0	0	0	0	12
2	1	2	0	0	0	0	12
1	2	2	0	0	0	0	12
0	3	2	0	0	0	0	12

Table G. 8

Scheduling Policy For Figure 4. 10 (3, 50 Pages)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	1	12
2	3	0	0	0	0	1	12
1	4	0	0	0	0	1	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	0	21
3	1	0	0	0	1	0	21
2	2	0	0	0	1	0	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	0	21
1	2	0	0	0	2	0	21
0	3	0	0	0	2	0	21
4	0	0	0	1	0	0	12
3	1	0	0	1	0	0	12
2	2	0	0	1	0	0	12
1	3	0	0	1	0	0	12
0	4	0	0	1	0	0	12
4	0	0	1	0	0	0	21
3	1	0	1	0	0	2	21
2	2	0	1	0	0	2	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	0	21
1	2	0	1	0	1	0	21
0	3	0	1	0	1	0	21
3	0	0	2	0	0	0	21
2	1	0	2	0	0	0	21
1	2	0	2	0	0	0	21
0	3	0	2	0	0	0	21
4	0	1	0	0	0	0	12
3	1	1	0	0	0	0	12
2	2	1	0	0	0	0	12
1	3	1	0	0	0	0	12
0	4	1	0	0	0	0	12

Table G. 9

Scheduling Policy For Figure 4. 10 (3, 55 Pages)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	2	12
2	3	0	0	0	0	2	12
1	4	0	0	0	0	2	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	1	21
3	1	0	0	0	1	1	21
2	2	0	0	0	1	1	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	0	21
1	2	0	0	0	2	0	21
0	3	0	0	0	2	0	21
4	0	0	0	1	0	0	12
3	1	0	0	1	0	2	12
2	2	0	0	1	0	2	12
1	3	0	0	1	0	2	12
0	4	0	0	1	0	2	12
3	0	0	0	1	1	0	12
2	1	0	0	1	1	0	12
1	2	0	0	1	1	0	12
0	3	0	0	1	1	0	12
4	0	0	1	0	0	1	21
3	1	0	1	0	0	1	21
2	2	0	1	0	0	1	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	0	21
1	2	0	1	0	1	0	21
0	3	0	1	0	1	0	21
3	0	0	1	1	0	0	12
2	1	0	1	1	0	0	12
1	2	0	1	1	0	0	12

Table G. 9 (continued)

0 3	0 1	1 0	0 12
3 0	0 2	0 0	0 21
2 1	0 2	0 0	0 21
1 2	0 2	0 0	0 21
0 3	0 2	0 0	0 21
4 0	1 0	0 0	0 12
3 1	1 0	0 0	2 12
2 2	1 0	0 0	2 12
1 3	1 0	0 0	2 12
0 4	1 0	0 0	2 12
3 0	1 0	0 1	0 12
2 1	1 0	0 1	0 12
1 2	1 0	0 1	0 12
0 3	1 0	0 1	0 12
3 0	1 1	0 0	0 12
2 1	1 1	0 0	0 12
1 2	1 1	0 0	0 12
0 3	1 1	0 0	0 12

Table G. 10

Scheduling Policy For Figure 4.11 (60 Pages, No Sharing)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	1	12
2	3	0	0	0	0	1	12
1	4	0	0	0	0	1	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	0	21
3	1	0	0	0	1	2	21
2	2	0	0	0	1	2	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	0	21
1	2	0	0	0	2	0	21
0	3	0	0	0	2	0	21
4	0	0	0	1	0	0	12
3	1	0	0	1	0	0	12
2	2	0	0	1	0	0	12
1	3	0	0	1	0	0	12
0	4	0	0	1	0	0	12
4	0	0	1	0	0	0	21
3	1	0	1	0	0	2	21
2	2	0	1	0	0	2	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	0	21
1	2	0	1	0	1	0	21
0	3	0	1	0	1	0	21
3	0	0	2	0	0	0	21
2	1	0	2	0	0	0	21
1	2	0	2	0	0	0	21
0	3	0	2	0	0	0	21
4	0	1	0	0	0	0	12
3	1	1	0	0	0	0	12
2	2	1	0	0	0	0	12
1	3	1	0	0	0	0	12
0	4	1	0	0	0	0	12

Table G. 11

Scheduling Policy For Figure 4. 11 (60 Pages, Sharing)

STATE						DECISION	
MEM		CPU		I/O			
1	2	1	2	1	2	A1	A4
5	0	0	0	0	0	1	12
4	1	0	0	0	0	1	12
3	2	0	0	0	0	1	12
2	3	0	0	0	0	1	12
1	4	0	0	0	0	2	12
0	5	0	0	0	0	2	12
4	0	0	0	0	1	0	21
3	1	0	0	0	1	2	21
2	2	0	0	0	1	2	21
1	3	0	0	0	1	2	21
0	4	0	0	0	1	2	21
3	0	0	0	0	2	0	21
2	1	0	0	0	2	2	21
1	2	0	0	0	2	2	21
0	3	0	0	0	2	2	21
2	0	0	0	0	3	0	21
1	1	0	0	0	3	0	21
0	2	0	0	0	3	0	21
4	0	0	0	1	0	1	12
3	1	0	0	1	0	1	12
2	2	0	0	1	0	1	12
1	3	0	0	1	0	1	12
0	4	0	0	1	0	0	12
3	0	0	0	2	0	1	12
2	1	0	0	2	0	1	12
1	2	0	0	2	0	1	12
0	3	0	0	2	0	0	12
2	0	0	0	3	0	0	12
1	1	0	0	3	0	0	12
0	2	0	0	3	0	0	12
4	0	0	1	0	0	0	21
3	1	0	1	0	0	2	21
2	2	0	1	0	0	2	21
1	3	0	1	0	0	2	21
0	4	0	1	0	0	2	21
3	0	0	1	0	1	0	21
2	1	0	1	0	1	2	21
1	2	0	1	0	1	2	21
0	3	0	1	0	1	2	21

Table G. 11 (continued)

2 0	0 1	0 2	0 21
1 1	0 1	0 2	0 21
0 2	0 1	0 2	0 21
3 0	0 2	0 0	0 21
2 1	0 2	0 0	2 21
1 2	0 2	0 0	2 21
0 3	0 2	0 0	2 21
2 0	0 2	0 1	0 21
1 1	0 2	0 1	0 21
0 2	0 2	0 1	0 21
2 0	0 3	0 0	0 21
1 1	0 3	0 0	0 21
0 2	0 3	0 0	0 21
4 0	1 0	0 0	1 12
3 1	1 0	0 0	1 12
2 2	1 0	0 0	1 12
1 3	1 0	0 0	1 12
0 4	1 0	0 0	0 12
3 0	1 0	1 0	1 12
2 1	1 0	1 0	1 12
1 2	1 0	1 0	1 12
0 3	1 0	1 0	0 12
2 0	1 0	2 0	0 12
1 1	1 0	2 0	0 12
0 2	1 0	2 0	0 12
3 0	2 0	0 0	1 12
2 1	2 0	0 0	1 12
1 2	2 0	0 0	1 12
0 3	2 0	0 0	0 12
2 0	2 0	1 0	0 12
1 1	2 0	1 0	0 12
0 2	2 0	1 0	0 12
2 0	3 0	0 0	0 12
1 1	3 0	0 0	0 12
0 2	3 0	0 0	0 12

BIBLIOGRAPHY

[1] M. T. Alexander, "Time Sharing Supervisor Programs", Computing Center, The University of Michigan, May 1969.

[2] B. Arden, B. Galler, T. O'Brien, F. Westervelt, "Program and Addressing Structure in a Time-Sharing Environment", Journal of the ACM, 13, 1, January 1966.

[3] B. Arden, "Time-Sharing Systems: A Review", Proc. of IEEE International Conv, 15, Part 10, March 1967, pp. 23-35.

[4] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", IBM Systems Journal, 5,2, 1966, pp. 78-101.

[5] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, N. J., 1957.

[6] R. Bellman, "A Markovian Decision Process", Je of Mathematics and Mechanics, 6, 5, (1957) pp. 679-684.

[7] R. S. Burington and C. C. Torrance, Higher Mathematics With Application to Science and Engineering McGraw-Hill, New York, 1939.

[8] W. Chang, "A Queuing Model for a Simple Case of Time-Sharing", IBM Systems Journal, 5, 2, 1966, pp. 115-125.

[9] E. G. Coffman, "Analysis of Two Time-Sharing Algorithms Designed for Limited Swapping", Journal of the ACM 15, 3, July 1968, pp. 341-353.

[10] E. G. Coffman and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment", CACM, 11, 7, July 1968, pp. 471-474.

[11] D. R. Cox and W. L. Smith, Queues, Wiley, New York, 1961.

[12] P. J. Denning, "Resource Allocation in Multiprocess Computer Systems", Project MAC Tech. Rept. MAC-TR-50, 1968 (Thesis).

[13] P. J. Denning, "The Working Set Model for Program Behavior", Communications of the ACM, 11, 5, May 1968, pp. 323-333.

[14] J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM, 12, 4, Oct. 1965, pp. 589-602.

[15] M. Eisenberg, "Multi-Queues with Changeover Times", Tech. Rept. No. 35, Operations Research Center, M. I. T., April 1968 (Thesis).

[16] D. W. Fife, "The Optimal Control of Queues, with Applications to Computer Systems", Tech. Rept. No. 170, Cooley Electronics Laboratory, University of Michigan, October 1965.

[17] D. W. Fife, "An Optimization Model for Time-Sharing", Proc. AFIPS 1966 SJCC, 28, pp. 97-104.

[18] G. H. Fine, C. W Jackson and P. V. McIsaac, "Dynamic Program Behavior Under Paging", Proc. 21th National Conf. of ACM, Thompson Book Co., Washington, D. C., 1966.

[19] F. R. Gantmacher, The Theory of Matrices, Chelsea, New York, 1960.

[20] D. P. Gaver, Jr., "Probability Models for Multiprogramming Computer Systems", Journal of the ACM, 14, 3, July 1967, pp. 423-438.

[21] C. T. Gibson, "Time-Sharing with the IBM System 360 Model 67", AFIPS Conference Proceedings 28, (SJCC 1966), pp. 61-78.

[22] M. Greenberger, "The Priority Problem", MIT Project MAC, November 1965, MAC-TR-22.

[23] R. A. Howard, Dynamic Programming and Markov Processes, MIT Press, 1960.

[24] W. S. Jewell, "Markov Renewal Programming", Parts I and II, Operations Research, 11, pp. 938-971, (1963).

[25] L. Kleinrock, "Certain Analytic Results for Time-Shared Processors", IFIP Congress 68 Proceedings, August 1968, D119-D125.

[26] L. Kleinrock, Communication Nets Stochastic Message Flow and Delay McGraw-Hill, 1964.

[27] L. Kleinrock, "Time-Shared Systems A Theoretical Treatment", JACM, 14, 2, April 1967.

[28] B. Krishnamoorthi and R. C. Wood, "Time Shared Computer Operations with Both Interarrival and Service Exponential", Journal of the ACM, 13, 3, July 1966, pp. 317-338.

[29] B. W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems", Communications of the ACM, 11, 5, May 1968, pp. 347-360.

[30] D. J. Lasser, "Productivity of Multiprogrammed Computers-Progress in Developing an Analytic Prediction Method", CACM 12, 12, December 1969, 678-684.

[31] M. A. Leibowitz, "An Approximate Method for Treating a Class of Multiqueue Problems", IBM Journal, 5, 3, July 1961.

[32] M. A. Leibowitz, "A Note on Some Fundamental Parameters of Multiqueue Systems", IBM Journal, 6,4, pp. 470-471, October 1962.

[33] M. L. Liou, "A Novel Method of Evaluating Transient Response", Proc. of IEEE, 54, 1, January 1966, pp. 20-23.

[34] P. M. Morse, Queues, Inventories and Maintenance, Wiley, 1958.

[35] J. M. McKinney, "A Survey of Analytical Time-Sharing Models", Computing Surveys, 1, 2, June 1969, pp. 105-116.

[36] N. R Nielsen, "An Analysis of Some Time-Sharing Techniques", Communications of the ACM, 14, 2, February 1971, pp. 79-90.

[37] A. Papoulis, Probability, Random Variables, and Stochastic Processes, McGraw-Hill, 1965.

[38] E. Parzen, Stochastic Processes, Holden-Day, 1962.

[39] T. B. Pinkerton, "The MTS Data Collection Facility", Memorandum 18, CONCOMP, The University of Michigan, June 1968.

[40] T. B. Pinkerton, "Performance Monitoring in a Time-Sharing System", CACM, 12, 11, Novwmber 1969, pp. 608-610.

[41] T. B. Pinkerton, "Program Behavior and Control in Virtual Storage Computer Systems", Tech. Rept. 4, CONCOMP, University of Michigan, April 1968.

[42] R. Pyke, "Markov Renewal Processes: Definitions and Preliminary Properties", and "Markov Renewal Processes with Finitely Many States", Ann. Math. Stat., 33, pp. 1231-1259 (1961).

[43] B. Randell and C. Kuehner, "Dynamic Storage Allocation Systems", Communications of the ACM, 11, 5, May 1968, pp. 297-306.

[44] P. J. Rasch, "A Queueing Theory Study of Round-Robin Scheduling of Time-Shared Computer Systems", Journal of the ACM, 17, 1, January 1970, pp. 131-145.

[45] J. Riordan, Stochastic Service Systems, Wiley, 1962.

[46] T. L. Saaty, Elements of Queueing Theory, McGraw-Hill, 1961.

[47] J. H. Saltzer and J. W. Gintell, "The Instrumentation of Multics", CACM, 13, 8, August 1970, pp. 495-500.

[48] A. L. Scherr, "An Analysis of Time-Shared Computer Systems", MAC-TR-18, Cambridge, M. I. T. Project MAC, 1965.

[49] J. E. Shemer and G. A. Shippey, "Statistical Analysis of Paged and Segmented Computer Systems", IEEE Trans. on Electronic Computers, EC-15, 6, December 1966, pp. 855-863.

[50] J. L. Smith, "Markov Decisions on a Partitioned State Space, and the Control of Multiprogramming", 07742-4-T, Systems Engineering Laboratory, University of Michigan, April 1967.

[51] J. L. Smith, "Multiprogramming Under a Page on Demand Strategy", Communications of the ACM, 10, 10, October 1967, pp. 636-646.

[52] Lajos Takacs, Introduction to the Theory of Queues, Oxford University Press, New York, 1962.

[53] J. Todd, Survey of Numerical Analysis, McGraw-Hill, 1962.

[54] University of Michigan, "MTS: Michigan Terminal System", University of Michigan Publication Distribution Services, Volumes I, II, III, February 1971.

[55] V. L. Wallace and D. L. Mason, "Degree of Multiprogramming in Page-on-Demand Systems", CACM, 12, 6, pp. 305-308, June 1969.

[56] V. L. Wallace and R. S. Rosenberg, "RQA-1, The Recursive Queue Analyzer", 07742-1-T, Systems Engineering Laboratory, University of Michigan, February 1966.

[57] E. S. Walter and V. L. Wallace, "Further Analysis of a Computing Center Environment", CACM, 10, 5, May 1967.

[58] D. J. White, "Dynamic Programming, Markov Chains, and the Method of Successive Approximations", Journal of Mathematical Analysis and Applications, 6, pp. 373-376 (1963).