PERFORMANCE EVALUATION OF A MULTIPROCESSOR IN A REAL TIME ENVIRONMENT

by

Jaynarayan H. Lala

B. Tech (Honors) Indian Institute of Technology, Bombay
1971

S.M. Massachusetts Institute of Technology
1973

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Science
In Instrumentation

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
FEBRUARY 1976

Signature of Author _____ 28 JAN 1976

Certified by _____
Thesis Supervisor

Certified by _____
Thesis Supervisor

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Instrumentation Doctoral Committee

ACKNOWLEDGEMENT

This report was prepared by The Charles Stark Draper Laboratory, Inc., under Grants GJ-36255 and DCR74-24116 from the National Science Foundation.

The author is deeply indebted to his advisor Professor Albert Hopkins for his invaluable guidance and advice. In addition, Prof. Hopkins' ever optimistic attitude proved to be a constant source of encouragement. The author would like to express his gratitude to the other members of the committee, Professors Wallace Vander Velde and Stuart Madnick for their constructive criticism of this thesis.

Thanks are also due to many individuals at the Digital Development and Digital Computation Groups of the C.S. Draper Laboratory for their assistance and pertinent comments. In particular, J.F. McKenna's invaluable effort in setting up the experimental multiprocessor, CERBERUS, is gratefully acknowledged.

The publication of this report does not constitute approval by the Charles Stark Draper Laboratory or the National Science Foundation of the findings or conclusions contained therein. It is published only for the exchange and stimulation of ideas.

PERFORMANCE EVALUATION OF A MULTIPROCESSOR
IN A
REAL TIME ENVIRONMENT

by

Jaynarayan H. Lala

Submitted to the Department of Aeronautics and Astronautics on December 31, 1975, in partial fulfillment of the requirements for the degree of Doctor of Science.

ABSTRACT

This thesis is concerned with the performance evaluation of a multiprocessor for real time applications. The real time environment imposes some very challenging requirements on computer systems, such as high reliability, availability and promptness of response. Multiprocessor computers are finding increasing use in such applications because they can be designed to include such qualities as graceful degradation and ease of expansion.

A three-processor bus-centered computer system was instrumented to facilitate performance monitoring through hardware. The real time environment was simulated by a work load consisting of a set of periodic jobs. The jobs were programmed to consume various system resources in a controlled manner, and not to perform any actual real time functions. Hence the jobs are named pseudo-jobs. The pseudo-jobs were parameterized to facilitate an easy variation of job-mix characteristics. An executive program was developed to dispatch jobs, to accept job requests and to perform other operating functions. Software was also written to help monitor system performance.

An analysis of the experimental results showed that the throughput efficiency is very sensitive to the ratio of average job length to the executive program length. For low values of this ratio, the executive program and the executive lockout together account for a large percentage of processor time. A detailed study of the job start delay statistics resulted in the following important conclusion: although the average job start delay increases without bound as the load approaches 100 percent, the maximum percentile delay is found to increase nearly linearly in the range of low to moderately high load factors. This indicates that multiprocessor systems may be operated in real time applications at relatively high load factors without proportionately penalizing the system performance. Another important result concerns the comparison of two job scheduling strategies. A first-come first-serve job dispatch strategy appears to be more suitable for multiprocessors than shortest-job-first strategy, which reduces the average delay slightly but significantly widens the delay spread. The frequency distribution of delay in all cases is found to be of a hyperbolic type, indicating that a large percentage of jobs start with short or no delays even at high load factors.

An analytical solution was obtained for the limiting case of zero executive program length. Numerical solutions of an expanded Markov model showed good qualitative agreement with the experimental results. A further refinement of the model resulted in an excellent agreement between the theoretical and the experimental results. The Markov process method proved to be a very useful mathematical tool for validating the experimental results.

Thesis Supervisor: Albert L. Hopkins, Jr.
Title: Associate Professor of Aeronautics & Astronautics

Thesis Supervisor: Wallace E. Vander Velde
Title: Professor of Aeronautics and Astronautics

Thesis Supervisor: Stuart E. Madnick
Title: Assistant Professor of Management Science

TABLE OF CONTENTS

Chapter		Page
I	INTRODUCTION	13
II	A SURVEY OF PERFORMANCE EVALUATION TECHNIQUES OF COMPUTER SYSTEMS	17
	2.1 Introduction	17
	2.2 Motivation for Evaluating Computer Performance	18
	2.2.1. Measurement and Analysis	18
	2.2.2. Performance Projection	19
	2.2.3. Performance Prediction	19
	2.3 Performance Evaluation Techniques	19
	2.3.1. Hardware Monitoring	19
	2.3.2. Software Monitoring	20
	2.3.3. Artificial Workloads	21
	2.3.3.1. Instruction-Mixes and Kernel Programs	21
	2.3.3.2. Benchmark and Synthetic Programs	22
	2.3.3.3. Probabilistic Job-Mix Models	23
	2.3.4. Simulation	23
	2.3.5. Analytical Models	24

Chapter		Page
III	PAST RESEARCH IN MULTIPROCESSOR PERFORMANCE EVALUATION	25
	3.1 Introduction	25
	3.2 Number of Processors Versus Processor Speed	27
	3.3 Memory Interference in Multiprocessors	28
	3.4 Executive Lockout	33
	3.5 Input Output Control	36
	3.6 Conclusions	38
IV	CERBERUS SYSTEM DESCRIPTION	39
	4.1 Introduction	39
	4.2 System Architecture	40
	4.3 Processor Architecture	42
	4.4 Microprogram and Scratchpad Memories	45
	4.5 Main Memory Interface	47
	4.6 Instruction and Bus Speeds	47
	4.7 CERBERUS Assembly Language	49
	4.8 Cross Assembler and Linkage Editor	51
	4.9 Special Instrumentation	53
V	MULTIPROCESSOR PERFORMANCE EVALUATION EXPERIMENTS	57
	5.1 Introduction	57
	5.1.1. Synchronized Job Load	60
	5.1.2. Asynchronous Job Load	60
	5.1.3. Scheduling Strategy for an Asynchronous Job Load	63
	5.2 Synthetic Job Mix	66
	5.3 The Waitlist	69
	5.4 The Executive	72

Chapter		Page
VI	EXPERIMENTAL RESULTS	77
	6.1 Introduction	77
	6.2 System Utilization	78
	6.3 Job Starting Delay Distribution	89
	6.4 FCFS Versus SJF	105
	6.5 Summary	113
VII	THEORETICAL MODELS AND RESULTS	115
	7.1 Introduction	115
	7.2 A Simple Analytical Model	116
	7.3 Exponential Distribution Markov Model	126
	7.4 Markov Model Results	134
	7.5 Erlang Model and Result	151
	7.6 Summary	166
VIII	CONCLUSION	167
	8.1 Introduction	167
	8.2 Results and Their Significance	168
	8.3 Perspective Summary	171
	8.4 Areas for Further Research	172
	BIBLIOGRAPHY	175

LIST OF ILLUSTRATIONS

Figure		Page
4.1	CERBERUS Structure	41
4.2	Processor Structure	43
4.3	Register Structure	44
4.4	Main Memory Interface	48
4.5	Hardware Monitor	55
5.1	Synchronized Job-Load	61
5.2	Short Job vs. Long Job	65
5.3	The Waitlist	70
6.1	Per Cent Useful Job Step Computation vs. System Load for R = 1, 10 and 50	80
6.2	Per Cent Overheads vs. Load for R = 1, 10 and 50	81
6.3	Per Cent Exec, Wait Exec and Bus Time vs. Load for R = 1	82
6.4A	Per Cent Exec, Wait Exec and Bus Time vs. Load for R = 10 (Low Job-Step-Length Variance)	83
6.4B	Per Cent Exec, Wait Exec and Bus Time vs. Load for R = 10 (High Job-Step-Length Variance)	84
6.5	Per Cent Exec, Wait Exec and Bus Time vs. Load for R = 50	85
6.6	Throughput Efficiency vs. R for Load = 30, 50, 70 and 90 Per Cent	88
6.7	Per Cent Job, Exec, Wait Exec vs. Load for R = 10 (Bus Use = 0)	90
6.8	Normalized Mean Job Start Delay vs. Load for R = 1, 10 and 50	91

Figure		Page
6.9	Absolute Mean Job Starting Delay vs. Mean Job Step Length	93
6.10A	Probability Distribution Function (PDF) of Delay for R = 1 (Histogram)	94
6.10B	Cumulative Distribution Function (CDF) of Delay for R = 1	95
6.11A	PDF of Delay for R = 10 (Histogram)	96
6.11B	CDF of Delay for R = 10	97
6.12A	PDF of Delay for R = 50 (Histogram)	98
6.12B	CDF of Delay for R = 50	99
6.13	PDF of Delay for R = 1 (Continuous)	100
6.14	PDF of Delay for R = 50 (Continuous)	101
6.15	Percentile Delay vs. Load for R = 1	103
6.16	Probability Delay is Greater Than X vs. Load for R = 1	104
6.17	Probability Delay is Greater Than 0 vs. Load for R = 1, 10 and 50	106
6.18	Normalized Mean Delay vs. Load for R = 1 (FCFS vs. SJF)	107
6.19	PDF of Delay for R = 1 (FCFS vs. SJF)	108
6.20	Normalized Mean Delay vs. Load for R = 10 (FCFS vs. SJF)	109
6.21	PDF of Delay for R = 10 (FCFS vs. SJF)	110
6.22	Normalized Mean Delay vs. Load (FCFS vs SJF) for R = 10 (High Variance)	111
6.23	PDF of Delay for R = 10 (FCFS vs. SJF)	112
7.1	State Transition Diagram for an M-Processor Markov Model	117
7.2	Probability Delay is Greater than Zero vs. Load (Analytical Solution)	120
7.3	Normalized Mean Delay vs. Load (Analytical Solution)	122
7.4	State Transition Diagram for 3-Processor Markov Model	123
7.5	Delay Paths	125

Figure		Page
7.6A	Exponential Markov Model Without Bus Use	127
7.6B	Exponential Markov Model Without Bus Use	128
7.7	Per Cent Useful Computation vs. Load for R = 1, 10 and 50 (Exponential Model)	137
7.8	Throughput Efficiency vs. R for Load = 30, 50, 70 and 90% (Exponential Model)	138
7.9	Per Cent Job, Exec and Wait Exec vs. Load for R = 10 (Exponential Model)	139
7.10	Normalized Mean Delay vs. Load for R = 1, 10 and 50 (Exponential Model)	140
7.11	Absolute Mean Delay vs. Mean Job Step Length for Load = 25, 50 and 90 Per Cent (Exponential Model)	141
7.12	Probability Delay > Zero vs. Load for R = 1, 10 and 50 (Exponential Model)	142
7.13	Probability Normalized Delay > d vs. Load for d = 0, 2 and 6 and R = 1 (Exponential Model)	143
7.14	Probability Normalized Delay>d vs. Load for d = 0, and 2 and R = 50 (Exponential Model)	144
7.15	Normalized Max. Delay vs. Load for R = 1 (Exponential Model)	145
7.16	PDF of Interarrival Times	146
7.17	Exponential Markov Model with Bus Use	149
7.18	Normalized Max. Delay vs. Load for R = 50 (Exponential Model)	150
7.19A	Erlang Markov Model	152
7.19B	Erlang Markov Model	153
7.20	Normalized Mean Delay vs. Load for R = 1, 10 and 50 (Erlang vs. Experiment)	161
7.21	Max. Delay vs. Load for R = 1 (Erling vs. Exper.)	162
7.22	PDF of Delay for R = 1 and Load = 25% (Experiment, Exponential & Erlang)	163
7.23	PDF of Delay for R = 1 and Load = 75% (Experiment, Exponential & Erlang)	164
7.24	PDF of Delay for R = 10 and Load = 50% (Experiment, Exponential & Erlang)	165

LIST OF TABLES

Table		Page
4.1	Instruction Speed	50
4.2	Bus Speed	50
5.1	Apollo Guidance Computer (AGC) Job-Mix Characteristics	59
5.2	AGC Job-Mix Characteristics	59
6.1	Mean Iteration Period vs. Per Cent System Load	86
7.1	Cumulative Distribution Function (CDF) of Delay (Approx. vs. Exact) for R = ∞ and Load = 25%	135
7.2	CDF of Delay (Approx. vs. Exact) for R = ∞ and Load = 50%	135
7.3	CDF of Delay (Approx. vs. Exact) for R = ∞ and Load = 75%	136

CHAPTER I

INTRODUCTION

Computer system design has historically been more of an art than a science. The design of any system that is as complex as a present day computer system can not become a science until the effects of all variables upon the system become well understood. Take for instance the case of the aircraft industry. In its infancy it was merely a matter of luck who stayed up in the air and who came crashing down to the ground. This is in a sense true of all complex systems including computers. To understand the impact of design variables on the system, to make computer design a science, has been the major motivating factor in the performance evaluation of computer systems. Performance evaluation can help in system design in a number of ways: by measurement and analysis of existing systems, by projecting the impact of additions and expansions on system performance, and by predicting performance of new designs.

The present work is concerned with the performance evaluation of a specific design for a specific set of applications: real time applications of multiprocessor computer systems. The real time environment imposes some challenging requirements on computer systems. These include high reliability, availability and promptness of response. Multiprocessors can apparently satisfy all of these requirements successfully. These and other qualities, such as graceful degradation and ease of expansion, are reasons for turning to multiprocessor designs for real time applications.

There has been considerable research in the area of multiprocessor performance evaluation. Mallach [28,29] has analyzed a multiprocessor computer configuration for spaceborne guidance computer applications. Memory interference in multiprocessors has been studied by Covo [11], Kurtzberg [22], Lehman [25] and Mitchell [32] among others. King [21] and Madnick [27] report on the problem of software lockout. The input/output requirements of multiprocessors have been the subject of study by Chang [6] and Colson [8]. While Baer [3], Lampson [24] and Ramamoorthy [34] have researched yet another aspect of multiprocessor performance: parallel processing of tasks and scheduling strategies for such applications. Even though the impact of each problem may be well understood, the overall effects are not easy to extrapolate or predict. This thesis presents the results of performance evaluation experiments conducted on the CERBERUS multiprocessor which was developed at the C.S. Draper Laboratory. The aim of the experiments was to investigate the effect of job mix characteristics in a real time environment on the multiprocessor performance. Since no simulation can stand on its own, various Markov models were also developed for comparison of results.

Chapter two surveys the techniques that have been used to evaluate computer system performance. It also analyzes the types of workloads that have been used to simulate operating environments.

Chapter three describes in detail various aspects of past research in the area of multiprocessor performance evaluation. These include the hardware configurations evaluated in each case, the evaluation techniques employed, the job mixes used, and the significant results that were obtained from these studies.

Chapter four is a description of the hardware, software and special instrumentation of the test bed for the present set of experiments, the CERBERUS multiprocessor.

Chapter five goes into the details of the job-mix, the simulation techniques and job dispatch strategies used in the present experiments. Results obtained from these experiments are described in Chapter six. Theoretical models were developed to validate the experimental results. These are described in the next chapter. Chapter seven also discusses the fidelity of the mathematical results. Chapter eight concludes the thesis with a summary of the major results and their significance and a brief description of the areas for further research and investigation.

CHAPTER II

A SURVEY OF PERFORMANCE EVALUATION TECHNIQUES OF COMPUTER SYSTEMS

2.1 Introduction

The present generation of computer systems have come a long way from the days of the Whirlwind I and the Mark I computers. First the development of the solid state technology and then the Large Scale Integration of electronic circuits reduced the complexity of the basic building blocks of computers. However, the same developments have also made it possible for computer system designers to increase the complexity of the overall system architecture. Performance prediction was trivial when the computers consisted only of a single processor, a magnetic core memory and a teletype. The computational tasks were limited mostly to scientific calculations, and the criterion of performance was usually the throughput [5].

Throughput is still a major criterion for optimizing system performance. However, the complexities of computer architecture and diversity of computer applications have introduced many more subtle factors in the evaluation of system performance. Today, instead of a single type of memory, there usually is a memory hierarchy with a varying speed/capacity/cost combination in a single computer system. The same is true of input and output devices, and an increasing number of systems are being designed with multiprocessing capabilities. Job-mix characteristics have also undergone at least as much change as the computer architecture.

Computer applications now vary from "number-crunching" to printing pay-checks. Due to these developments, performance criteria have been expanded to include a wide range of hardware characteristics as well as the capabilities of the operating system [26]. These, for example, may include such measures as machine organization (word size, control and data paths, etc.), secondary storage and peripheral unit performance, special features (virtual storage, scratchpad memories, etc.). They may also include software efficiency measures such as the speed of assembly and compilation of source code.

A popular measure of goodness of the overall system, apart from the throughput, is the system responsiveness. This would be determined by the average turn-around time in a batch processing system, by mean job starting delay in a real time application, by mean response time to a command in a time sharing system etc.

2.2 Motivation for Evaluating Computer Performance

The specific performance criteria depend very much on the final goal of the performance evaluation. Some of the general purposes of performance evaluation are the following.

2.2.1 Measurement and Analysis

This is concerned with performance evaluation of an existing system. This may be undertaken to determine, for example, if there are any bottlenecks in the system or the extent to which the resources are being utilized. Measurement and analysis of existing systems are also helpful in comparison and selection procedures.

2.2.2 Performance Projection

When several alternatives exist to expand or change an existing system, performance projection can help in choosing among different courses of action. For example, throughput may be increased either by increasing memory size or by changing to a faster processor. A performance analysis can point to the best way of achieving the desired throughput.

2.2.3 Performance Prediction

Is it better to have few fast or many slow processors in a multiprocessor system? Such questions can usually be answered by analyzing and predicting the performance of the designs involved. In general, performance prediction is oriented toward design of new computer systems.

2.3 Performance Evaluation Techniques

There are a number of evaluation techniques. The following gives their brief description along with some applications.

2.3.1 Hardware Monitoring

Hardware monitoring can be used only for the first of the three purposes outlined above. It may also be used, however, to provide a data base for projecting the performance of an existing system. A major advantage of this technique is that it does not perturb the parameters being measured, that is, it does not alter the system performance. This is due to the fact that this technique utilizes a number of passsive probes (sensors),

that are connected to control lines and registers throughout the computer system, to collect the performance statistics.

Hardware monitoring has been used extensively to evaluate processor utilization, input/output channel activities, program behavior (e.g. memory address reference patterns) and many other interesting statistics [17]. Hardware monitors are usually used in conjunction with the execution of actual programs. However, they may also be used with one of the "artificial" workloads discussed in Section 2.3.3.

2.3.2 Software Monitoring

Software monitoring has the same limitations (in terms of applications) as does the hardware monitoring technique. In addition, it may alter the variables of interest in some cases. However, the degradation of useful time can be kept to a minimum (as low as 1% of CPU usage) by designing the software package carefully [38]. The obvious advantage of this technique over the previous one is the flexibility of changing the nature of statistics to be gathered. Also hardware monitors tend to produce more voluminous data while some preprocessing and reduction of raw data is possible in software statistics gathering packages.

Even though no two software probes may look alike, the fundamental principle of operation is the same. Basically a probe is designed such that it is informed about the nature and time of each and every status change of the device (or resource) without unduly affecting the performance of the device. This technique has been used to gather statistics on program turnaround time, memory utilization, input/output device utilization etc. [40].

2.3.3 Artificial Workloads [5,9,10,26]

Workload characterization is a crucial problem in the performance evaluation of computer systems. Irrespective of the kind of approach used, that is, analytical models, simulation or hardware/software monitors, we must state the nature of tasks which will be performed by the computer system. Different performance evaluation techniques, however, may require varying degrees of details of the computer job-mix. Also each technique has its limitations as to what kinds of loads it can handle. For example, most analytical models require the service time distributions to be exponential or hyperexponential to keep the mathematical models tractable. Similarly, hardware monitoring techniques are very often used in a situation where the performance evaluator has little or no control over the system software or workload. That would not necessarily be bad, since the system is executing a real workload, except that measurements have to be taken over long periods of time in such cases.

On the other hand, in some instances the computer system is totally accessible. This would be the case, for example, when a new computer design is breadboarded or when a system is to be selected among several competing designs. In such cases it is important to choose an artificial workload that is representative of the real operating environment and which exercises all parts of the system. The following describes some of the methods in use in this area.

2.3.3.1 Instruction-Mixes and Kernel Programs

Memory access times and add times (time to execute an add instruction) have been used in the past to give a vague idea of the raw computing power of a machine. Obviously such measures of performance do not reflect upon the word size, number of addresses per instruction, the data path and the total system architecture, and hence can not be considered

representative of the actual system performance. The same can be said of the execution time of an instruction-mix which, for example, may include add, multiply, divide and indirect address instructions in proportion to their actual frequency of occurrence in the real workload. The next step in this direction is a kernel program. A kernel program consists of one or more routines, which are representative of the actual workload, such as matrix inversion, file update or differential equation solution. Kernel programs, like instruction-mixes, are not actually executed on the machine. Rather they are coded in assembly language and their execution time is obtained by adding individual instruction execution times. Once again, the time to execute a kernel program is a measure that emphasizes only the computational aspects of the system.

2.3.3.2 Benchmark and Synthetic Programs

A benchmark is a program that is actually executed by the machine as distinct from a kernel program that is not. As such it can be made much more complex and hence more representative of the actual workload. There has been some emphasis on standardizing the benchmarks so as to provide a common reference for comparison of computer systems. However this defeats the purpose of simulating the actual job-mix which would not, of course, be standard for all the applications.

A synthetic program is basically a parameterized representation of the real workload. That is, each aspect of the job-mix, such as computation time, I/O time, storage requirements etc., is represented by a parameter. The job-mix, thus, can be easily altered by changing the parameters.

Each of the above techniques is normally used in conjunction with hardware and/or software monitors. The total number of runs and the composition of the benchmark or the parameters of the synthetic job for each run depends upon the nature and complexity of the computer system.

A multiprocessor may require a greater number of runs than a uniprocessor, and a time sharing system would require a more complex workload than a batch processing system.

2.3.3.3 Probabilistic Job-Mix Models

With mathematical models of computer systems it is difficult to represent program characteristics in microscopic detail since only a limited number of variables can be handled by such models. The overall job-mix properties or the macroscopic details, however, are relatively easily modeled by probabilistic distributions. In studies of multiprogrammed computer systems, for example, it is not uncommon to find a job-mix represented mainly by percent compute and I/O times with relative ordering of compute and I/O activity described by a probability distribution [21].

2.3.4 Simulation

This method of evaluating the performance of computer systems can be divided into three phases. The first phase is concerned with the modeling of the system. The type of model referred to here is a recursive functional relationship between design variables and measures of performance [16]. The second phase consists of translating the model into a computer program, very often using a specialized simulation language. The last phase of simulation is the statistical analysis of the data obtained from the computer programs.

Simulation is one of the more popular methods of evaluating computer performance. There is a tendency to accept the results obtained by this method without any qualifications, notwithstanding the fact that simulation is only as close to reality as the model itself. Therefore it is necessary to validate the simulation results.

Apart from validation, there are some other disadvantages of using simulators. First, a unique simulator is usually constructed for each new problem. That is, a general purpose parameterized program capable of simulating any proposed configuration does not exist. Second, the cost and effort involved in designing a simulator can be appreciable. Third, the results of simulation lack the generality and the niceties of a closed form solution obtained from a mathematical model. And last, very often the delay between statement of a problem and its solution using simulation technique is so long as to make the results useless in a decision making process [39].

On the plus side simulation is capable of handling more variables and more complex systems than mathematical techniques. It can be used where analytical models would not lend themselves to closed form solutions.

2.3.5 Analytical Models

Most mathematical models of computer systems are based upon queueing theory and Markov processes. A limited number of papers using graph theory [34] and decision theory [2] have also appeared in literature.

Most of pros and cons of mathematical models have already been pointed out in the preceding section. In addition, in some cases they can also be optimized fairly easily. The disadvantage of these models is the following. Almost always some simplifying assumptions must be made in order to retain the mathematical tractability of the model. This tends to reduce the sphere of applicability of the results; in short, a validation problem.

Therefore it is not uncommon to find instances where analytical models are used in conjunction with simulation [6,28].

CHAPTER III

PAST RESEARCH IN MULTIPROCESSOR PERFORMANCE EVALUATION

3.1 Introduction

Multiprocessor computer systems have been proposed, designed and built for many reasons. Some of the major advantages offered by such an architecture include increased reliability, graceful degradation, expandability and speed of computation [19,20]. There is some confusion in the literature at present as to the definition of multiprocessing. Baer [3], for example, defines multiprocessing as simultaneous processing of two (or more) portions of the same program by two (or more) processing units. This seems to be a rather restrictive definition of multiprocessing and can be better used to define parallel processing. That is, multiprocessing should, and does, include processing of N (N>1) independent tasks on N processors.

The advent of multiprocessing has given rise to a host of interesting problems. For example, to effectively exploit the speed of parallel processing it is necessary to convert sequential programs into parallel ones. Some problems, such as vector addition or matrix manipulation, lend themselves naturally to parallel processing. Others may require a non-trivial amount of effort to recognize parallelism in them. Having decomposed a large program, a weather prediction model for example, into many small concurrently executable segments, it is necessary, then, to schedule and allocate the subtasks to processors such that some

performance measure is optimized. These topics have been dealt with extensively in the literature [3,24,25,34]. However the scheduling strategies for parallel processing of jobs will not be discussed here further since they do not refer to a specific multiprocessor structure.

Apart from speeding up the computation of a single problem by utilizing the inherent parallelism of a multiprocessor, there are many other means of improving multiprocessor performance. These other ways directly relate to the organization of the machine and its response to different operating environments. For example, while the number of processors is a common variable in almost all the multiprocessor studies, the specific problem of choosing the number of processors and their speed, keeping the throughput constant, has been studied only by Mallach [29].

Processor members of a multiprocessor, by definition, communicate chiefly through shared storage modules as distinct from multicomputers, where communication amongst processors is through Input/Output channels [6]. In a multicomputer, if a processor fails its associated memory module also becomes useless, and vice versa. This does not happen in a multiprocessor. However, sharing of resources brings about access conflicts which degrade the system performance in terms of throughput efficiency. This brings us to the problem of memory interference which has been studied by Lehman [25], Mitchell et al [32] and Kurtzberg [22] among others. Covo [11] suggests a means of reducing interference by partial program memory replication in an optimal fashion.

Other resources which are shared in a multiprocessor computer system are the supervisory programs that schedule and allocate tasks to processors. Madnick [27] reports on the problem of multiprocessor software lockout. King [21] has suggested a switch strategy, which basically consists of multiprogramming the multiprocessor, to limit the degradation in performance due to lockout.

Many different philosophies exist for handling the Input/Output requirements of a multiprocessor. For example, each processor may handle its own I/O functions. Or, there may be one processor (or more depending upon the load) dedicated to I/O activity. These and other variations have been studied by Chang et al [6] and Colson [8].

The types of workloads used in performance evaluation studies to simulate appropriate operating environments range from instruction mixes to synthetic programs. Lehman [25], Covo [11] and Mallach [29] have used benchmarks, while Mitchell et al [32] simulate an instruction mix execution and King et al [21] use a probabilistic job mix model in their mathematical model. Hardware monitors, simulators and mathematical models have all been employed usefully to evaluate system performance in these studies.

The following sections report in greater detail upon the various aspects of the multiprocessor performance outlined above. They also describe the actual hardware configurations which were evaluated in each case, the performance evaluation techniques and the job mixes used and any significant results that were obtained from these studies.

3.2 Number of Processors Versus Processor Speed

The number of processors in a multiprocessor system impacts not only the system performance but also the overall system reliability and fault-tolerant capabilities of the machine. Obviously a single-fast-processor system can not even endure a single fault in the processor . We shall, however, concern ourselves only with the throughput performance aspects of the system, remembering that system reliability should be given appropriate consideration when designing a multiprocessor.

Mallach [29] considers a bus centered multiprocessor consisting of processors, memories and an I/O controller all connected by a data bus. The shared storage units are used only for data, and it is assumed that each processor has its own copy of all programs. The evaluation techniques used here consist of simulation and Queueing theory models. The workload executed by the simulator can be described as a benchmark since it consisted of a set of lunar-landing Apollo Guidance Computer programs. In the case of the queueing model the job mix is characterized by Poisson arrivals and exponentially distributed service times. Using as performance criteria mean job acceptance delay and system-full time fraction (i.e. percent of time all processors are busy), it was concluded that a system with more, slower processors performs better than a system with fewer, faster ones. However, it was also found that the effect of less bus contention, occurring in a system with fewer processors, tends to offset the effects of better processor utilization in a system with more processors.

Eventually, of course, processors can not be made indefinitely slower and more numerous, especially for many real-time applications, for the following reason. In such situations there are a number of tasks that must be executed periodically at predetermined intervals. Ordinarily, one iteration of such a task must return its results before the next can begin. Thus the individual processors must be sufficiently fast so that each such periodic task can be completed within its allotted time on one processor.

The number of processors also has an effect on storage interference. This is treated more fully in the following section.

3.3 Memory Interference in Multiprocessors

There are a number of ways of connecting P processors to S storage units in a multiprocessor computer system. The most flexible and efficient, from the view point of performance, of these methods would

provide a direct connection between each processor and each memory module. In such a hardware configuration the P processors can access P memory modules, each processor talking to a different storage unit, simultaneously. This type of interface between processors and memories, known as the crossbar switch, is one end of the spectrum providing extreme flexibility at the expense of circuit complexity. The other end of the spectrum is a time-mltiplexed memory bus connecting all the units in the system. In such a hardware configuration, a very inexpensive one, only one processor can communicate to a memory module at any given time. However, some flexibility can be achieved here by overlapping transmission of data to/from one storage unit and the read/write cycle of another storage unit. Between the crossbar switch and the time-multiplexed bus there are other methods of connecting processors and memories varying in cost,reliability and performance.

One problem, however, is common to all of these methods. When more than one processor wants to access the same storage module all except one must wait their turn (according to some priority scheme). This phenomenon, known as memory conflict or storage interference, results in each processor being idle some of the time. The amount of time a processor wastes in waiting to gain access to memory is a function of many factors. Clearly it should increase with the number of processors and decrease with the number of storage units. Apart from these it also depends on how programs/data are distributed in the memory modules and how the processors are scheduled to accesss these programs/data.

Lehman [25] considers a crossbar switch type of multiprocessor system in which individual processors have an IBM 360 like structure and execute an augmented subset of System/360 machine language. The evaluation technique used here is a parameterized simulator in which the number of storage modules and processors as well as the instruction execution times can be selected for each run. The type of workload executed

by the simulator can be characterized as a benchmark. Three sets of experiments were conducted using the following three benchmarks:

1. a numerical analysis program,

2. a matrix multiplication program, and

3. an electrical network analysis program.

The study found that storage interference degrades the system performance (presumably measured in terms of processor time wasted due to interference) by 20% for a storage module-to-processor ratio of two, dropping to 5% for the ratio of eight. Addition of local processor stores (scratchpads) and their use as instruction buffers almost eliminated the interference. These results refer to the concurrent execution of a single loop of the type 1 benchmark. The second set of experiments were conducted for the type 2 benchmark (multiplication of two 40X40 matrices). It was found that total processor time (sum total of time that individual processors were active in the program) increased linearly with the number of processors. This number was varied from 2 to 16 keeping the number of storage modules constant at 64. Storage interference (measured as the processor time spent waiting for storage-access) increased from 0.4% to 7% to 48% of total processor time as the storage/processor ratio was reduced from 32 to 4 to 1. The ratio of unity was obtained by reducing the number of storage modules from 64 to 16.

For the third job mix the total run time (from the time the first processor begins the first task to the time the last processor finishes the last task) decreased rapidly from 100 to 20 units and the processor time increased only slightly from 100 to 150 units as the number of processors was increased to 10. Beyond that, the storage interference worsened so

much that the run time took a turnaround and increased sharply. Here too, the number of storage modules was kept constant at 64.

Mitchell et al [32] report on the interference problem in a multiprocessor system that they evaluated using simulation techniques as well as actual hardware monitoring. The system consisted of K processors, M read only program stores (with interleaving of odd and even addresses), and L read-write variable stores used for data/instructions. The interface between processors and memories was a crossbar switch. The job mix consisted of timed tasks derived from a real time scenario. Four kinds of instruction mixes - all branch instructions, arithmetic and logical instructions, and all 'No op' instructions - were used to evaluate the system throughput.

The effects of varying the processor-to-storage module ratio on the system performance are clearly demonstrated by this study. For example, when the number of processors was varied from 1 to 7, holding the number of program stores at one, the throughput registered a signigicant initial increase and then levelled off around 4 processors depending on the job mix. On the other hand, when the number of program stores was also increased so that the processor/storage module ratio remained constant at one, the throughput increased linearly all the way though at a slightly lesser pace than the uniprocessor throughput extrapolation.

A high storage module to processor ratio, however, does not by itself guarantee a good system performance, unless the total usage of the storage units is distributed uniformly across all the memory modules. This was aptly demonstrated by the same simulation study. The hardware configuration consisted of 10 processors, 9 program stores, and 10 variable stores. The distribution of load on the memory units in the simulation study was varied by changing the probability of choosing a particular memory unit from zero to unity. This way the system throughput when all processors

executed out of the same unit was found to be about 40% of that when each processor had its own dedicated memory.

There are a number of ways of reducing memory conflict in a multiprocessor system. For example, some or all of program memory units may be replicated a certain number of times. In addition to this, programs or jobs may be assigned to memory modules in a way that minimizes the memory conflict. These two approaches have been analyzed by Covo [11] and Kurtzberg [22] respectively.

Covo considers m processors connected to k storage modules through a time-multiplexed memory bus. Storage module j (j=1,2, ... k) is of size x_j and is replicated n_j (n_j =<m) times. The problem addressed here is the determination of n_j, given the program contents of each memory unit as well as the access rate of each program, such that a desired throughput would be achieved at a minimum cost. This problem was solved by the author using dynamic programming techniques. The results showed that partial replication may cause 30 to 40 percent reduction in the combined cost of processor and program memory units, compared to full replication.

Two points are worth mentioning here. First of all it is necessary to have a knowledge of the access frequency of all programs. Second, the savings in cost as compared to full replication of all program memory units depends very much on what the frequency distribution is like. For example, very often an 80-20 or 90-10 rule may be applicable to the frequency distribution. That is, 20 percent (or 10) of all code is referenced 80 (or 90) percent of the time. In that case complete replication of only those units that contain the most frequently referenced programs would result in improved performance at minimal cost.

The allocation of programs to memories has been treated by Kurtzberg [22] for a two processor system. Two cases were analyzed:

a) the processor schedule is specified

b) no definite schedule is specified.

For each of these cases the author formulates an allocation algorithm, using quadratic programming models, such that the conflict is minimized.

The memory contention problem in multiprocessors has been researched rather extensively. Examples of some recent studies are [4,7,33].

3.4 Executive Lockout

In a multiprocessing system there are parts of the operating system or the executive program that should not be executed concurrently by more than one processor. Examples include job dispatcher or scheduler, any programs that modify the work queues, and test programs that exercise parts of the system by simulating failures. When more processors than one queue up for the executive, to fetch the next job for example, all but one of the processors must wait their turn to get access to the executive. Idling of processors due to this phenomenon is known as the executive lockout.

It is sometimes difficult to distinguish the executive lockout phenomenon from a memory conflict problem. There are several reasons for this. First of all, they both originate as a result of sharing of resources. Second, their impact on the system performance is similar: it is negative. Third, both of them are modeled using almost identical methods. The important difference, however, comes about from the fact that memory conflict is due to sharing of hardware, i.e. , the memory modules, while executive lockout is due to sharing of software, i.e. the executive program.

The memory conflict is present only because of cost constraints. If this constraint were relaxed completely, then, at least in principle, a system could be designed with as many copies of each memory module as there are processors in the system, thereby totally overcoming the problem. The same can not be said of the executive lockout. It is a necessary evil that we must live with to keep everything in order.

Madnick [27] considers a simplified Markov process model of the multiprocessor to study this problem. Assuming that job lengths and executive lengths are exponentially distributed random variables with mean values of L and E time units respectively, he derives some interesting results. (Since it may be possible to run parts of the executive concurrently, E here refers only to non-concurrently executable parts of the supervisory program). The expected number of locked out processors (i.e. the number of processors waiting for the executive at any given moment) shows a quasi-exponential increase as a function of the total number of processors in the system, for a fixed ratio of E to L. For example, for E/L = 0.05 and 15 processors in the system, it is expected that on the average one processor will be idled by executive lockout at any given moment. Expanding the system to 40 processrs results in an expected average of 19 idle processors, a further expansion to 41 processors results in 20 idle processors; evidently the point where additional hardware is totally useless has been reached. The lockout state ratio (E/L) itself has considerable impact on the system performance. For a 20 processor configuration, for example, the expected number of idle processors increases from 2 to 10 to 15 as the ratio E/L is increased from 0.05 to 0.10 to 0.15.

The operating system or the executive program usually consists of many distinct non-concurrently executable programs (King et al [21] refer to these as serial programs). One of these is the job dispatcher. However, there may be some other program segments that are independent of the job dispatcher. Now, if a processor is idled because it is free to start a next job but can not access the dispatcher then there is nothing that can

remedy the situation. However, if the processor is locked out because it can not execute other parts of the operating system then this idle time may be used to do other available work. In particular, this other work may be a process (job) that is ready to run. If the system is multiprogrammed then a number of processes, depending on the degree of multiprogramming, are ready to run at any given time. King et al refer to this as the switch strategy since a locked out processor, in some instances, may switch, rather than wait, to another task. They compare the throughput for switch and wait strategies using a simplified queueing network model. The executive program is assumed to consist of K serial programs, all of which may be executed concurrently.

It was found that after a certain point the throughput of the wait strategy becomes independent of the degree of multiprogramming and approaches a constant value. The switch strategy throughput can be increased further, though not indefinitely, by increasing the degree of multiprogramming. It was also found in this study that the throughputs of both strategies increase rapidly as the number of serial programs, K, is increased, and the difference between the wait and the idealized switch strategies decreases as K is increased. (The overheads in switching to another process were neglected in this study, hence the term 'ideal switch strategy'). Therefore, in an actual situation with a certain amount of switching overhead, the switch strategy would not be better unless the superiority of the idealized switch strategy is more than enough to compensate for the overhead of switching.

One may ask the question why not use the switch strategy to overcome the memory conflict problem also. The answer is that it may be, but is not likely to be, worth while. The reason is that when memory conflict arises out of interference in fetching instructions it is going to be for short durations at relatively high frequency. Sometimes, however, it may be due to shared data in which case it will be for a relatively longer period of time. But data conflict between related tasks is a rarer phenomenon, and hence does not affect the system performance appreciably.

3.5 Input Output Control

A number of philosophies exist for treating the input/output requirements of a multiprocessor. We will denote the control program that manages the I/O functions for the system as the I/O executive, and a processor that runs this program as the I/O processor. The difference in philosophies originates from the treatment of the I/O executive and the I/O processor(s). The I/O executive becomes a floating executive, for example, when each processor unit of the multiprocessor is given freedom to run the I/O executive program. On the other hand, if one or more processors are designated to be I/O processors, then the I/O program becomes known as a dedicated I/O executive. There is also a third category, a semi-dedicated executive. The semi-dedicated I/O executive is one that can reside in any processor, that is, the I/O processor designation can be changed to apply to any processor member of the system under the master program or the main executive program control.

Each of these approaches has certain pros and cons [8]. For example, the advantage of a floating I/O executive is that the individual programs call the executive as needed, and therefore the I/O system control resides to a large degree within the jobs running on the system. Thus, computing time is spent on running the I/O executive only when necessary. The advantage of the dedicated executive is that one processor (or more, depending on the load) controls all the I/O jobs and makes all the decisions. The I/O bus contention is minimum in this configuration. However the capacity of the processor(s) may be underutilized. The advantages and the disadvantages of these approaches may be diminished or magnified depending on the structure and operating environment of the system. Consider, for example, the highly fault-tolerant network connecting many local processors to the central multiprocessor complex, as described by Smith [36]. In this system the I/O processor must perform many tasks in addition to the conventional load of such a processor. These include the network reconfiguration in case of a failure, polling the local processors

(LP's) to detect errors or messages from any of the LP's etc. In a semi-dedicated or dedicated executive, these additional tasks may be run during the idle time of the I/O processor thus utilizing its capacity to a fuller extent. However, in a floating executive, they have to be considered as independent tasks which must be run at specific intervals of time and, therefore, increase the overall load on the system [8].

There is a certain advantage of a semi-dedicated executive over a fully dedicated one and it stems from the system fault-tolerance considerations. In the event of a failure of the I/O processor the system can continue to function with a semi-dedicated executive but not with the fully dedicated one.

Chang et al [6] compare the performance of floating and dedicated I/O executives for a multiprocessor system to a multiprogrammed uniprocessor computer. Their results indicate that going from multiprogramming to multiprocessing with a floating I/O executive decreases the throughput (the J processors of the multiprocessor are considered to be J times slower than the uniprocessor). The throughput difference decreases with the decrease of J. The degree of multiprogramming is assumed to be fixed and equal to the number of storage modules.

Colson [8] has studied, using Monte Carlo techniques, the performance of a semi-dedicated executive for a system described in Reference [20]. The results indicate a nearly linear I/O processor utilization as a function of the task arrival rate even as saturation is approached. Also studied was the average delay of a task requiring service by the I/O processor. This was found to be a conventional exponential-like behavior. The queue length of the tasks in the I/O processor, on the other hand, increased linearly with load with a sharp upturn near saturation which is attributed to pseudo-pipelining effect of the local processor delays.

3.6 Conclusions

By and large various aspects of the multiprocessor performance have been studied and understood; the area of memory conflict stands out particularly well in this respect. And even though the impact of each problem may be well understood the overall effects are not easy to extrapolate or predict. For example, an optimum number of processors and/or memory modules can be computed considering the memory conflict or the executive lockout by itself. However the sum total or the global optimum may be quite different. Particularly lacking is data on system performance in an actual operating environment considering all aspects of the system.

CHAPTER IV

CERBERUS SYSTEM DESCRIPTION

4.1 Introduction

The CERBERUS multiprocessor was designed and fabricated at the C.S.Draper Laboratory originally under a contract from the Manned Spacecraft Center of the National Aeronautics and Space Administration and later funded by the National Science Foundation - to demonstrate fault-tolerance and failure recovery techniques [15,19,30]. One of the main features of the system is that its three processor elements can be configured under the operator's control to act in one of two different modes. One is the multiprocessing mode. The other is the Triple Modular Redundant (TMR) mode. In the TMR mode the three processors execute identical programs in "tight synchronism" comparing the results of each microstep at the end of each microinstruction. The processors are synchronized with the help of a common timing source, which is a fault-tolerant clock also developed at the Draper Laboratory [12]. The TMR feature was incorporated into the system to show the feasibility of operating three processors in tight synchronism, which has since been successfully demonstrated. This mode of operation of the system is not directly relevant to performance evaluation of the computer as a multiprocessor and hence will not be discussed further.

CERBERUS is a microprogrammed multiprocessor. Each processor is provided with a copy of the microprogram memory. Each processor element also has its own cache or scratchpad memory. This is in addition

to a single storage module shared by all three processors. There is a special performance monitoring panel interfaced to the system to facilitate hardware performance measurement. System software support includes a cross assembler and a cross linkage editor which can be run using batch or time sharing facilities of an IBM 360/75 computer. These and other aspects of the CERBERUS system are described in detail in the following sections.

4.2 System Architecture

The CERBERUS sytem represents a part of an overall design of a fault-tolerant multiprocessor that has been put forth by Hopkins [19] and Green et al [15]. The philosophy underlying the design has been discussed in detail in Reference [19]. The discussion here therefore is limited to a description of the architecture of the multiprocessor under consideration.

The system, when configured as a multiprocessor, looks as shown in Figure 4.1. Intercommunication between the processor elements is through a shared memory. Access to the main memory is via a time-multiplexed fully parallel bus, and is controlled by a memory interface unit. The interface unit uses "round-robin" scheduling to service processor requests to access the storage unit. A processor may read or write a block of words at a time. The block may consist of as little as a single word or as many as 256 consecutive locations (equal to the size of the scratchpad).

Very much similar to the memory bus is a time-multiplexed fully parallel Input/Output bus controlled by an I/O interface unit. The I/O interface unit, unlike the memory interface unit, does not directly connect to an Input/Output device. Rather it connects to one node of a network of nodes, each of which is capable of directly interfacing to an I/O device or a subsystem computer. The I/O network, as it is known, is a set of redundant

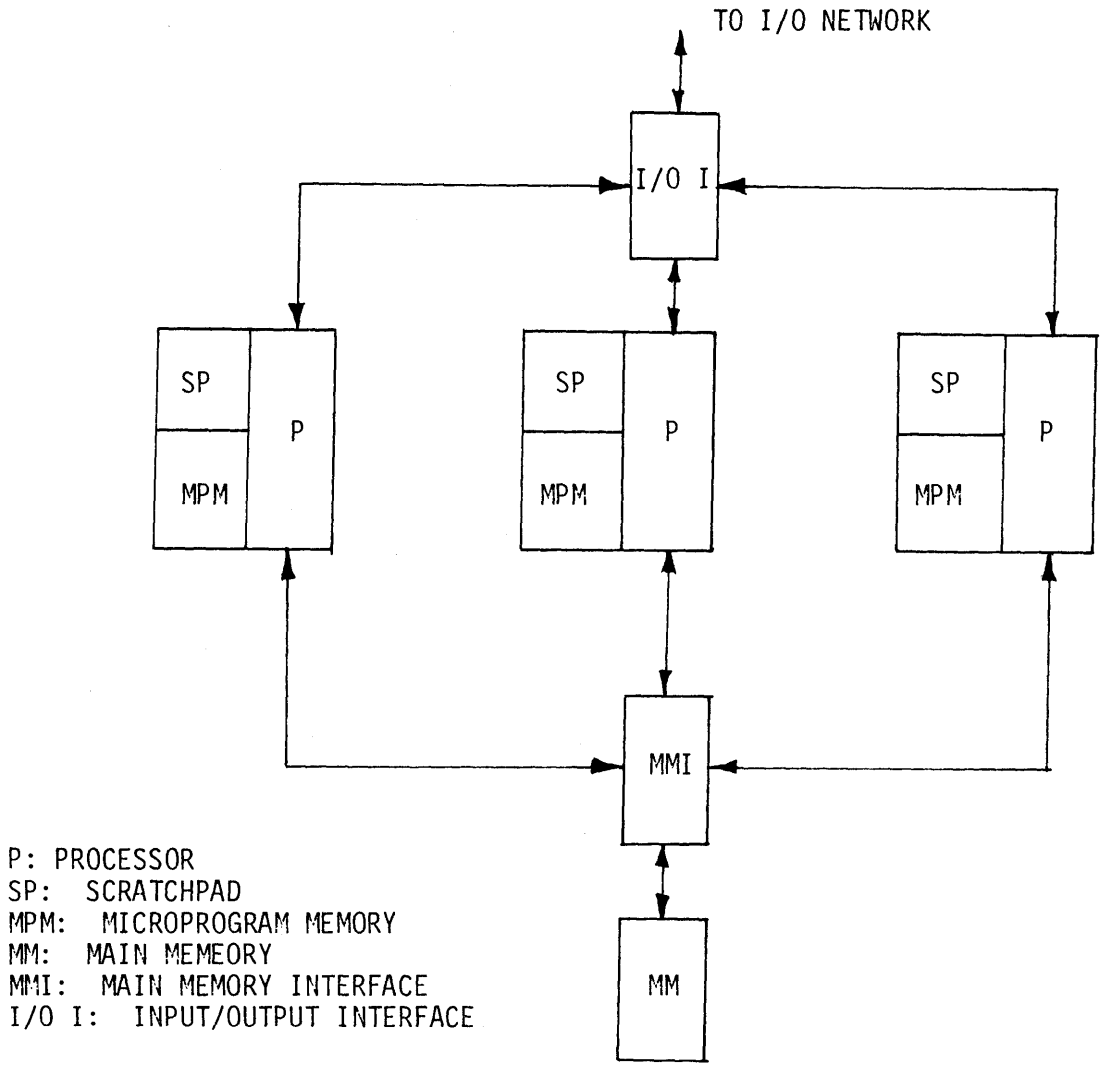


Figure 4.1. CERBERUS Structure.

and flexible connections between the central computer complex and the local processors and I/O devices [36,37]. Since the performance evaluation project does not include any I/O activity, the I/O details are largely irrelevant to the present discussion. Suffice it to say, then, that a teletype unit, interfaced to the computer through one of the network nodes, is used to control the performance evaluation experiments. In addition to this, there is also a 300 characters/second optical tape reader that is used initially to load programs and data into the main memory as well as to load the microprogram into the microprogram memory.

4.3 Processor Architecture

The internal structure of the processor elements is shown in Figures 4.2 and 4.3. It is basically an accumulator-oriented machine. That is, in all two-operand instructions (such as add) one operand is assumed to be the accumulator. The processor then need fetch only one operand from the scratchpad to execute the instruction. The other registers shown in Figure 4.3 are used by the microprograms only, and are unavailable to the user. The only exception is the Processor Status Register or the PSR. It is an eight-bit half-register which can be used by the programmer to reflect current status of the processor. It is used for performance evaluation purposes as explained in a later section.

Turning our attention back to the processor architecture, an advantage of an accumulator-oriented machine is that it shortens the instruction width (in terms of bits per instruction). The disadvantage is that all intermediate results must be transferred from the accumulator to the scratchpad. Since the basic design philosophy calls for triplication and majority voting of all processor units to mask failures, the system does not employ any error detecting and/or correcting codes within the processors. The reasons for this choice have been amply discussed by Hopkins [19].

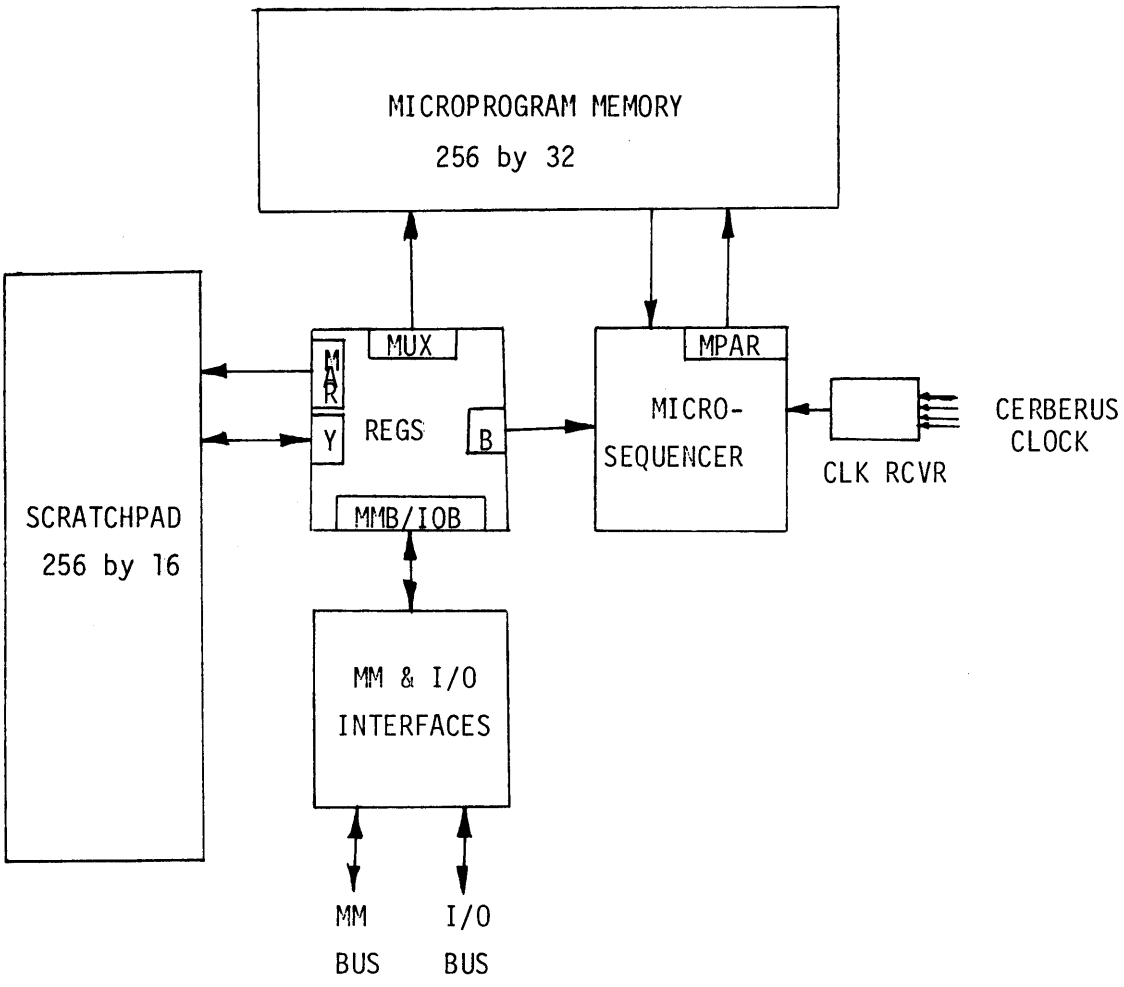
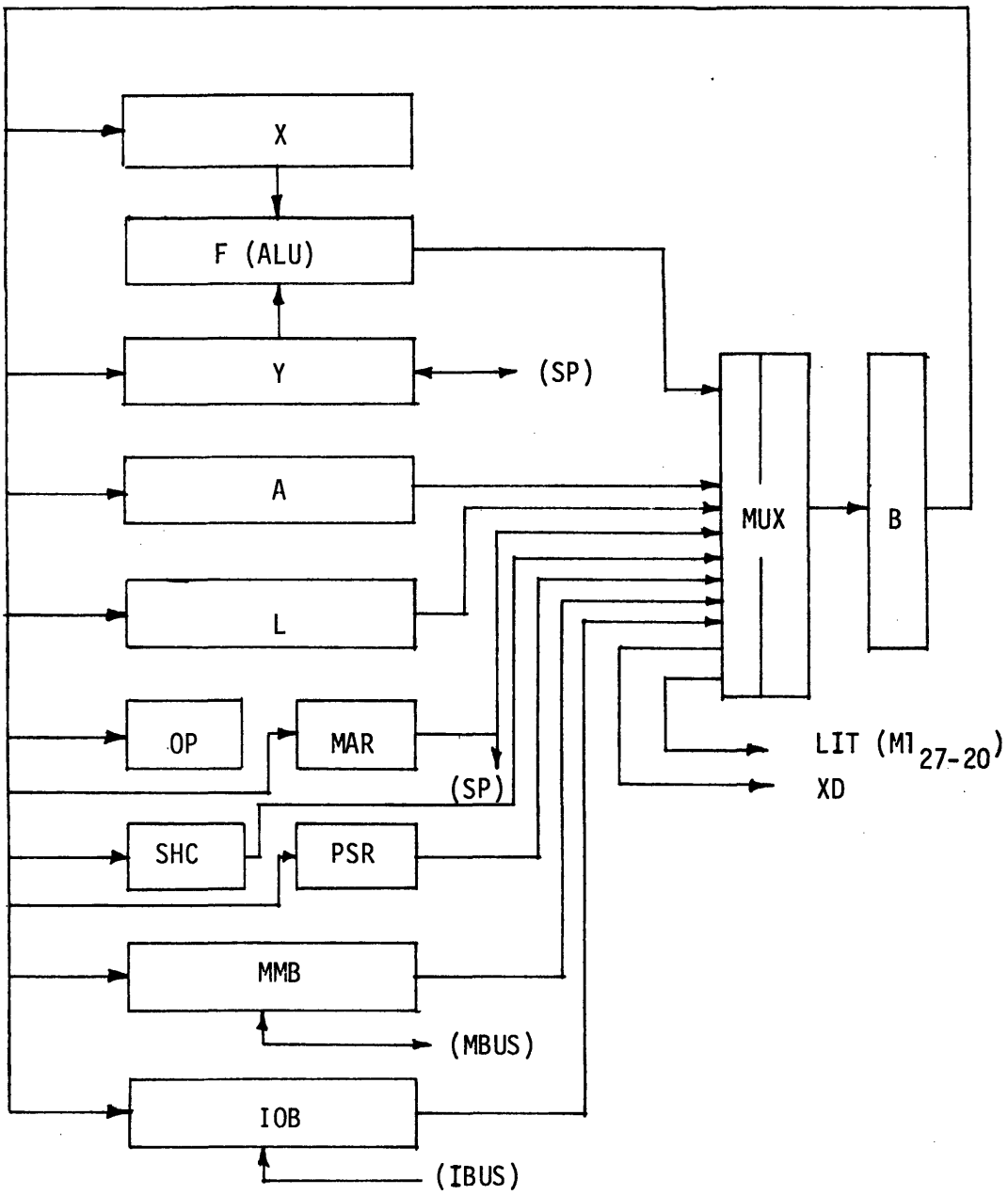


Figure 4.2. Processor Structure.



ALU: ARITHMETIC LOGIC UNIT
A,L: UPPER & LOWER ACCUMULATORS
B : BUFFER
LIT: LITERAL
XD : EXTERNAL DATA (MONITOR)
IOB: I/O BUFFER

MAR: MEMORY ADDRESS REGISTER
SHC: SHIFT COUNTER
OP : OPCODE REGISTER
PSR: PROCESSOR STATUS REGISTER
MMB: MAIN MEMORY BUFFER

Figure 4.3. Register Structure

The processors are driven by a common redundant clocking mechanism. Each clock pulse results in the execution of a single microinstruction. The steps within each microinstruction, the so called nanosteps, are timed by a high frequency internal timer independent to each processor. The building blocks used in the CERBERUS processors consist mostly of small scale and medium scale integrated circuits using Transistor Transistor Logic.

The CERBERUS instruction set can be divided into two types of instructions:

1. those that are executed out of the scratchpad or the scratchpad instructions and

2. others that are executed out of the main memory or the main memory instructions.

When a processor executes instructions out of the main memory it continually occupies the main memory bus. Other processors during that time are prevented from accessing the memory. In the scratchpad mode a processor does not occupy the bus. Most of the instructions are of the scratchpad type. The main memory instructions include those that begin and end a job, transfer information between the main memory and the scratchpad and transfer control to a different part of the program. These instructions, in general, control the flow of the program.

4.4 Microprogram and Scratchpad Memories

As pointed out in the preceding section each processor is provided with a copy of the microprogram. This microprogram resides in a random access semiconductor read/write memory with an access time of ap-

proximately 200 nanoseconds. The reasons for choosing a read/write memory are obvious. The microprogram and hence the whole instruction set can be changed at will, and this is important in an evolving system or a bread board. The disadvantage of such a memory over a ROM (read-only memory) is that it is comparatively less reliable. There are 32 256-by-1 memory chips in the microprogram memory; therefore it can hold a maximum of 256 microinstructions, each of which is 32 bits wide. The scratchpad that goes with each processor is also composed of sixteen of the same elements. Thus each scratchpad can hold 256 instruction and/or data words since each machine language instruction is 16 bits wide.

All programs begin and end in the main memory. A typical sequence of actions taken by a processor to execute such a program may be as follows. Let us assume that a processor has determined the job that is to be run next and has obtained its starting location. The processor may do this by running the executive program. It then proceeds to execute the new program. The first few instructions (main memory instructions) of the program would move a block of (scratchpad) instructions and any relevant data into the scratchpad. Having done this the processor would release control of the memory bus and begin to execute the scratchpad instructions. At some point it would become necessary to return to the main memory mode for any one of the following reasons;

1. return results ,

2. obtain more instructions/data ,

3. end of the program.

The processor would then wait until enabled to talk on the bus. A processor may go through a number of such cycles to complete a program. The last instruction of all programs must be 'ENDJOB'. This results in

transfer of control to the Executive. If another processor is running the Executive at that time the processor that just became free would have to wait until the Executive is unlocked. The last instruction of the Executive is 'BEGIN'. Execution of this instruction results in the release of the Executive and transfer of control to the starting location of the job to be run next.

4.5 Main Memory Interface

The memory interface unit shown in Figure 4.4 is a very versatile unit and does more than just control access to the memory bus. It has a sixteen bit wide parallel majority voter to vote on the three bus lines coming from the three processors. This is, of course, done only in the TMR mode of operation. It has a look-ahead capability which can be used to cut down instruction or data fetch time. It also has logic to set, reset or invert bits in main memory.

The processor talking on the bus sends a command to the interface unit specifying the type of operation to be performed such as read/write, bit operations etc., and the number of words to be fetched/sent, among other things. The interface interprets the command and takes appropriate actions. It also has a timer so that it can disable a processor after a certain length of time. This way a failed processor is prevented from making the bus unusable. The main memory can be divided into two parts; one to hold instructions and the other for data. The memory interface unit has the logic to compare addresses of all write requests and thus disallow any overwrites into the instruction area.

4.6 Instruction and Bus Speeds

The processor speed is determined by the clock speed. The clock frequency is 680 kilocycles per second. A processor therefore executes a

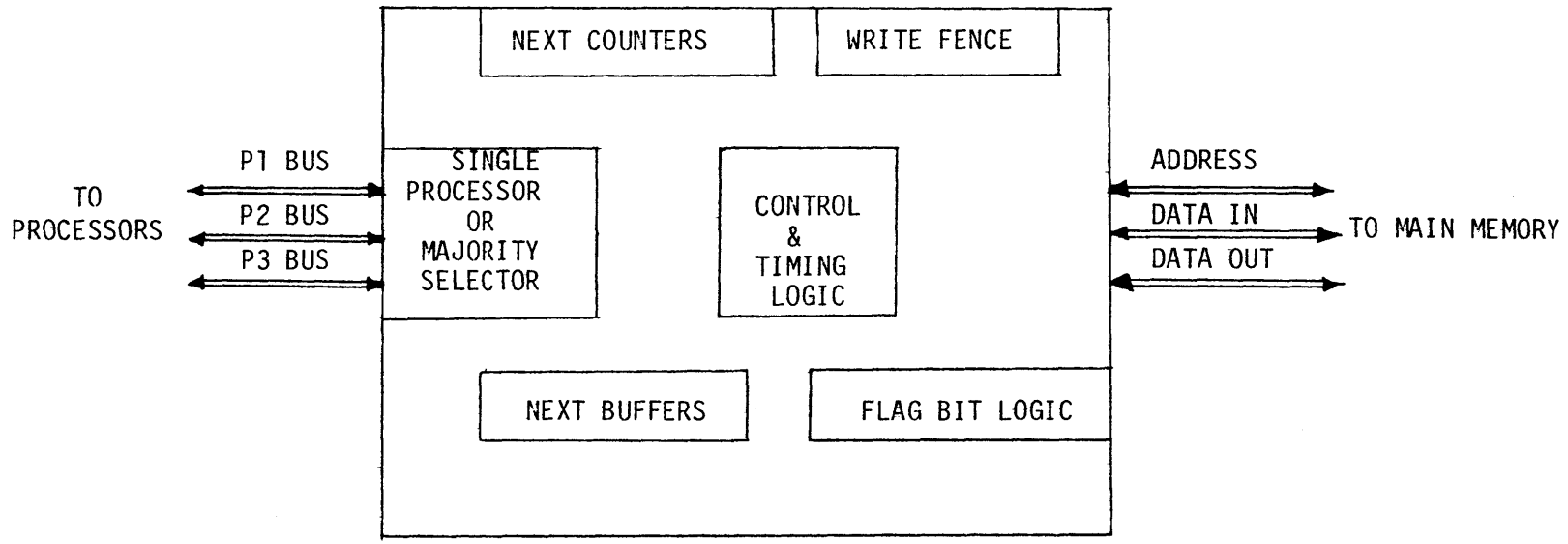


Figure 4.4. Main Memory Interface.

TABLE 4.1

INSTRUCTION	ADDRESSING MODE	# INSTRUCTIONS/SEC
ADD, SUB, LOAD, STORE	DIRECT	113,000
" "	INDIRECT	97,000
" "	INDEXED	75,000
MPY, DIV	DIRECT	14,000

TABLE 4.2

BLOCK SIZE	BUS SPEED WORDS/SEC
256	220,000
128	215,000
64	205,000
32	187,000
16	160,000

provided to use current values of location counters as operands in a program statement. Other pseudo-operators are provided to define symbols and constants, to signal start and end of the program as well as to separate various Control Sections (CSECTS) within a program. The CAL also accepts comments on each statement or between two statements for purposes of program documentation. An example program written in CAL which is actually used to interface the system to a teletypewriter unit is listed in Reference [7].

4.8 Cross Assembler and Linkage Editor

Translation of source programs which are written in CAL into an object code acceptable to the machine is done by an Assembler program. The Assembler itself is coded in the IBM 360 Assembly Language and runs on an IBM 360/75 system. Such an assembler that runs on a computer other than its target computer is known as a cross assembler. The reasons for not assembling programs on CERBERUS itself are two-fold. First, it would be fairly difficult, if not impossible, to fit an assembler program into the 4096 word CERBERUS memory. Second, in the absence of a cross assembler the self-assembler program itself would have to be translated into the machine language by hand. This second limitation ceases to be applicable once a cross assembler has been programmed and is running.

The CAL Cross Assembler, hereafter referred to simply as the Assembler, has two passes, i.e. it goes over the source program twice. In the first pass it basically makes a symbol list and assigns values to all the symbols defined in the program. More than a thousand different names or symbols can be used in a single program. While assigning values to symbols the Assembler will flag any names used more than once. In the first pass, then, the Assembler collects certain information used later on for assembly.

In the second pass the Assembler produces a word of object code for each statement, except for comments and statements involving some pseudo-operators, of the source code. It also formats the object code and punches it on a paper tape in a form acceptable to the machine. The first step in the translation procedure is identification of the operator. The Assembler looks through a table of valid pseudo-operators for a match. If such a match is found it takes action appropriate for that pseudo-op. For example, if the pseudo-op was ORG (origin) it would initialize the location counter(s) to value(s) specified in the operand field(s) of the source statement. No object code is produced by this statement and the Assembler goes on to work on the next statement. If no match is found then the Assembler looks through the instruction mnemonic (opcode) table. In case the operator being compared is a valid opcode a match would be found and its value would be obtained from the table. In addition to this the Assembler would also determine the type of the instruction from the opcode table. The instructions are divided, according to the number and type of operands required, into seven different categories. There is a separate routine to handle each category of instructions. This makes it easier to assemble different parts of the source statement into object code and also helps in diagnosing assembly language errors.

The Assembler converts other fields into their respective numerical values with the help of other tables. For example, it looks into a Mode Table for the type of addressing (indexed, indirect etc.) and into a Symbol Table to obtain values of operators that are not defined numerically. Many diagnostic aids are also provided. For example, symbols not defined in the program or invalid opcodes are flagged as errors by the Assembler. The Assembler produces two different forms of output: a punched paper tape ready for loading into the CERBERUS main memory, and a printed output showing object code and addresses beside each source statement. Assembly errors may be found and corrected at once by assembling the

program on-line using the time sharing facilities of the System 360. A listing of an example of an on-line session is shown in Reference [23].

A Cross Linkage Editor is also avaiable to link-edit and assemble CERBERUS programs.The Linkage Editor differs from the Assembler only in one respect. It has the capability of assembling many inter-related separate programs and producing a single machine language program. By interrelationship we mean that one program may use variables or operands defined or produced in another program. The program that uses operands or names from another program declares these names to be EXTERNAL. The program that defines or produces these same names declares them to be GLOBAL. The Linkage Editor then takes care of transferring values of GLOBAL operands from one program to another. In order to do this another pass over the source code is necessary. The Linkage Editor is therefore a three pass assembler. In addition to passes one and two of the assembler it has a pass zero which precedes the other two passes. In pass zero the linkage editor goes over all the programs once making a table of GLOBAL symbols and assigning values to these symbols. This table is later on used by pass one to obtain values of EXTERNAL symbols in each program.

The Linkage Editor can accommodate over a thousand different GLOBAL symbols for all programs and about the same number of local symbols for each program in its tables. Up to four thousand lines of source code may be assembled by the Linkage Editor in a single run.

4.9 Special Instrumentation

A real time clock was added to the multiprocessor to facilitate simulation of a real time environment and workload. The CERBERUS clock drives the real time clock through a variable divider. The least count of the clock may be changed from approximately 2 microseconds to 0.376 milliseconds using the divider. The CERBERUS clock frequency is assumed

to be 680 KHz. The real time clock represents the current time as a 16-bit wide word. One or more processors may copy this word simultaneously into their respective scratchpads by executing an instruction TIME. However, since this word overflows in approximately 25 seconds (with the highest possible least count) it is necessary to keep track of real time by using a double precision word. This is done by reserving two words in an executive data base for real time. Every time the Executive is run this double precision time is updated. This strategy will succeed if the Executive program is run at least once in 25 seconds. This is guaranteed to happen if the jobs are of comparatively much shorter duration, say several seconds at most, which is presently the case.

There is also a hardware Performance Monitoring Circuit which was added specifically for the purposes of evaluating system performance. Figure 4.5 shows details of the circuit. Basically there is an 8-bit wide Processor Status Register or PSR in each processor. The high order four bits of this register may be set to a desired value under program control. By letting each value correspond to a unique processor state (such as idle, waiting for executive, running executive), we can break down the processor time into as many as sixteen different categories. The duty cycles of processor states as indicated by the PSR contents are displayed on a set of DC milli-ammeters. The contents of the PSRs are routed to the three meters through a set of state selectors, processor selectors and low pass filters as shown in the figure. The function of the state selectors is obvious. Duty cycles of any one or a combination of states (e.g. running executive, waiting for executive or running and waiting for executive) may be chosen by the state selector to be displayed. The processor selector allows one to display the duty cycle for one particular processor or a combination of them. For example, one may display the percent of time processor one is busy running a useful task or one may choose to observe the percent of time all three processors are simultaneously busy running useful tasks. The function of the low pass filter is to suppress high frequency motion of the meter needle so that the needle would point to a steady time averaged

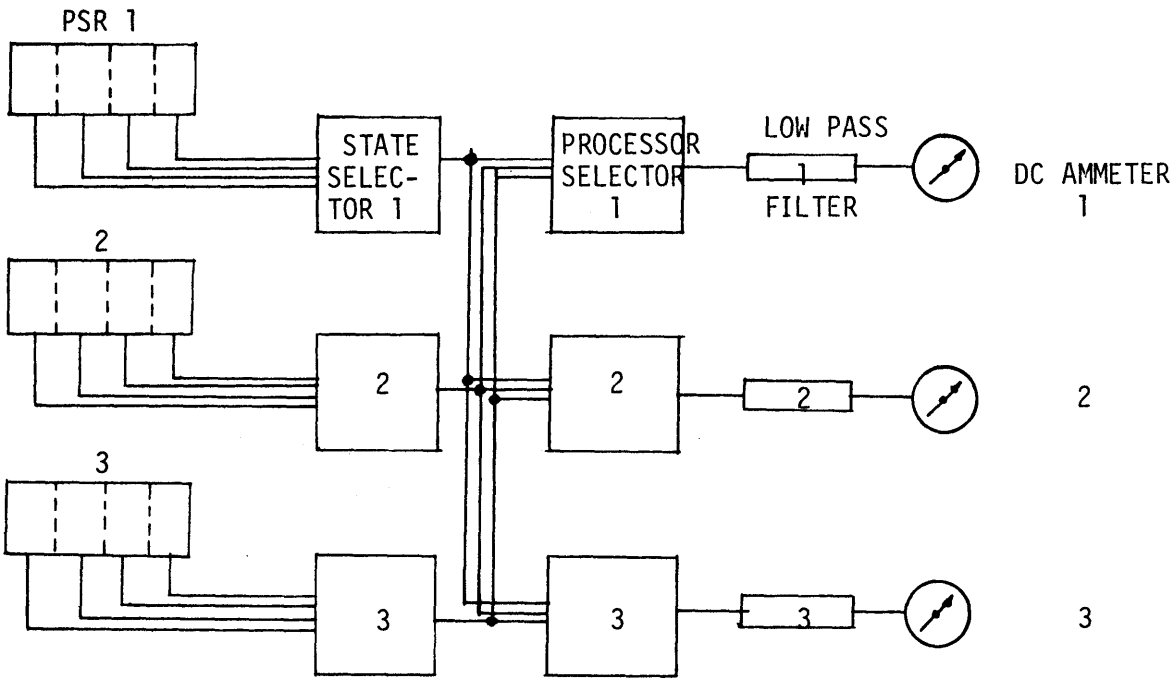


Figure 4.5. Hardware Monitor

value. The processor time was divided into the following five categories in the performance evaluation experiments.

1. Idle

2. Running Executive

3. Running a useful task

4. Using or waiting for the memory bus

5. Waiting for the Executive.

This will be discussed in greater detail in the next chapter.

Various details of the CERBERUS multiprocessor system can be found in References [1,12,13,23,31].

CHAPTER V

MULTIPROCESSOR PERFORMANCE EVALUATION EXPERIMENTS

5.1 Introduction

The goal of the present study has been to evaluate performance of a multiprocessor computer system in a real time environment. The architectural details of such a multiprocessor, CERBERUS, have already been described in the preceding chapter. The general purposes of performance evaluation as well as the numerous evaluation techniques used have also been discussed in the earlier chapters. What is intended to be done here is to report on a specific situation studied, the evaluation techniques employed, the performance criteria used, and the results obtained.

Let us, first of all, explore in detail the characteristics of a typical real time situation faced by a computer. This can be done by analyzing the job-mix of a real time application. Mallach [28,29] for example, has developed an estimate of the tasks that an aerospace guidance computer may have to perform by analyzing the jobs taken from the computer programs written for the Apollo lunar mission. We shall recapitulate here in short his assumptions and findings.

The largest computational loads maintained on the Apollo Guidance Computer during the mission were imposed during the descent phase of the lunar landing. The programs were analyzed to determine their execution times in terms of the number of instructions executed, data requirements, and cylic execution rates for periodic jobs. Forty three distinct job sequences

were identified in this phase of the mission. Of these 22 were found to be periodic in nature remaining active all through the landing phase. The rest of the jobs were either initiated by crew command or by another job. Most of these non-periodic one-time jobs are also required to be started at definite points in time. Let us concentrate our attention, however, on periodic jobs which constitute the major part of the total computational load on the system. A breakdown of the periodic load by execution time (in terms of number of instructions) and iteration period is shown in Tables 5.1 and 5.2.

Looking further into the nature of functions actually performed by these jobs we find that the job responsible for the central navigation loop is the longest of all (approximately 10,000 instructions) and is repeated at an interval of two seconds. The digital autopilot function is one of the most frequently executed jobs (every one tenth of a second) and is about 500 instructions long. Other jobs executed at this high frequency are mostly concerned with displaying information and monitoring crew commands. Some other moderately long jobs are thrust axis gimbal control (464 instructions), acceleration (212), and gyro torque calcuations (165). Each of these three jobs has an iteration period of two seconds.

Digital autopilot, navigation, gimbal controls and other jobs involve measurement of certain variables (probably at known times), estimation or computation of errors, and finally digital filtering and determination of actuator commands (in case of autopilot) or position (in case of navigation for example). In short, the computer is involved in digital compensation and control of many loops simultaneously. Each of these loops resembles a typical sampled data control system. This same type of job-mix, though not necessarily of the same composition in terms of job lengths or iteration periods, is likely to be encountered in other real time applications too. Examples include chemical process control, nuclear reactor control etc. Having seen the nature of the workload, let us now discuss how these jobs

TABLE 5.1

ITERATION PERIOD MSEC	NUMBER OF JOBS
120	4
240	1
480	5
2000	12
TOTAL	22

TABLE 5.2

EXECUTION TIME (Number of Instructions)	NUMBER OF JOBS
0-20	3
20-40	3
40-60	3
60-80	3
80-100	2
100-200	4
200-1000	3
>1000	1
TOTAL	22

may be actually scheduled for execution. There are two approaches to the problem of scheduling. These are discussed in what follows.

5.1.1 Synchronized Job Load

In this case the periodic load is completely prescheduled in such a fashion as to load the computer slightly less than one hundred per cent (maybe as low as 60-70 per cent). The non-periodic one-time jobs are scheduled by the computer for its otherwise free or idle time. This situation is depicted in Figure 5.1. In this figure the job lengths are proportional to the widths of the boxes. The jobs are started at the points marked on the time scale in the same figure. All jobs with the same iteration period have been lumped together and the details of their relative schedule are not shown. It is seen that when the total periodic load is scheduled in this manner all jobs would be executed on time leaving the computer free part of the time. This idle time may be used for non-scheduled or low priority jobs.

5.1.2. Asynchronous Job Load

A second approach to scheduling the work load is not to schedule the jobs at all. Eventually of course each job must start at a certain point in time. However in this approach to the problem no time slots are assigned to jobs a priori. Each job, whether periodic or not, is executed as it becomes due if a processor is available. Some priority scheme may be used, for example, to give repetitive jobs a higher priority than the non-periodic ones.

Let us now discuss the pros and cons of each strategy. First of all, to be able to preschedule and synchronize the workload one must know the detailed characteristics of the job-mix such as iteration periods, execution

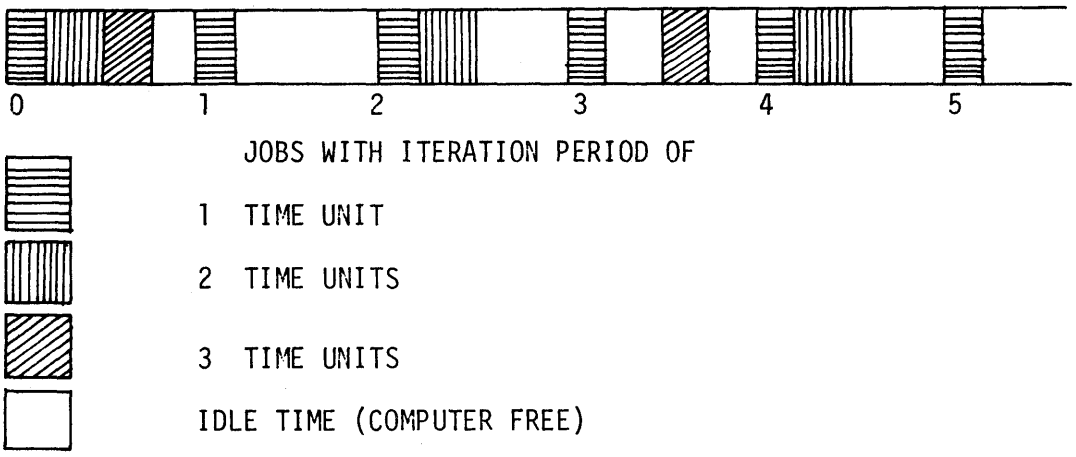


Figure 5.1. Synchronized Job Load.

times and relative order of execution, if any. In most real time situations this information would be available, though not necessarily sorted out in the form needed for scheduling purposes. In short the information may have to be determined laboriously. Second, the workload and its characteristics may change from one phase of a mission to another. In addition there may be a transition period between two phases where one set of jobs terminates and another set begins. Prescheduling would have to take care of all these situations. Third, most real time applications are likely to be very sensitive to computer failures. Therefore the computer systems would have to be fault-tolerant. Using a fault-tolerant computer does not, however, guarantee a toal immunity to faults. What it does imply is that a single fault will not cripple the system completely, but may degrade or reduce the system capacity. This is what is known as the graceful degradation property of certain fault-tolerant systems. What would be the impact of a reduced computing capacity on job scheduling? If prescheduling loads the system to near saturation, which may be totally acceptable in the absence of failures, a slight degradation in capacity may lead to inordinately long job queues. A new scheduling or a selective job execution may be needed under these circumstances. This too would have to be preplanned in the system. On the plus side, the advantage of synchronizing the job load is the ability to execute programs without delays, possibly utilizing a higher system capacity.

In the case of the second approach, where the jobs are executed in the order they are scheduled, one only needs to know the total loads imposed on the system during various mission phases. Degradation in capacity would no doubt affect both strategies. However in the second approach a degradation in capacity is equivalent to an increase in the load factor, the consequences of which are easier to predict and plan for. Also, last minute changes in computer programs, such as a change in the iteration frequency of a job, or the addition of a new job, does not affect this strategy at all. Often a point cited in favor of synchronizing the job load is the argument that synchronizing the job load in the face of graceful degradation or other

adverse conditions may not help, but will certainly not hurt the system performance. That, however, is not evident and may, in fact, be a false assumption. That is, a badly scheduled system may or may not perform better than a simpler strategy of executing jobs at the times for which they are scheduled. Even with a synchronized job load the only way to predict system performance in the face of adverse conditions will be to consider the load as totally random and not synchronized.

The schedule of synchronized jobs shown in Figure 5.1 is for a single processor system. That is, no two jobs overlap in the time frame. For a multiprocessor system this schedule would have to be altered to reflect the simultaneous execution of many jobs. In addition it would be necessary to have an idea of the processor conflict in sharing memory, Executive program and other resources.

5.1.3 Scheduling Strategies for an Asynchronous Job Load

Within the confines of an asynchronous job load a number of alternatives exist for choosing the order in which to run jobs. The simplest of all strategies is the 'First-Come-First-Serve' (FCFS) philosophy. Other more exotic schemes may utilize criticality of job function etc. as a priority criterion. For example, an aircraft control function such as an autopilot job may have priority over a navigation job while the navigation job would take precedence over a display routine. Such priority schemes increase the system management overheads, do not improve the overall computer performance and have to be implemented as a necessity to guarantee a certain response time to a privileged class of jobs. On the other hand if one could rearrange the jobs based on their total running time (i.e. the job lengths) there may be a possible improvement in performance as compared to the simpler FCFS scheme. For example, executing the shorter jobs first would reduce the mean job starting delay as shown by the following simple example.

Consider a uniprocessor system with no Executive overheads. Let there be two jobs in the waitlist due at the same time. Assume that job A is of duration L and job B is of duration l (l<L). Running the shorter job (B) first results in a mean delay of l/2 while in the other case the mean delay would be larger (L/2). A little thought shows that no matter what the waitlist looks like, executing shorter jobs first will always result in a smaller mean delay.

Now let us turn our attention to a multiprocessor system, say, with two processors. Consider a situation exemplified by Fig. 5.2.It is seen that by the time a processor becomes available job A is overdue by l1 and job B is also due. The mean delay will not be affected by a choice of jobs as long as another processor becomes available before P1 completes its present job (A or B). In this case the mean delay is (l1+l2)/2. However the delay distribution depends on the order in which the jobs are run. If the shorter job is given higher priority, then the two delays are 0 and l1+l2. In the normal FCFS schedule the delays would be l1 and l2. Therefore the SJF (Shortest-Job-First) schedule seems to

1. widen the delay spread and

2. increase the short delay probability
   while keeping the average delay constant.

Of course, the mean delay will decrease a little if P1 were to complete job B before another processor becomes free. The higher the number of processors in the system the more unlikely this will be. On the other hand the shorter job B is compared to A (higher job length dispersion) the more likely it will be.

This simple illustration shows that although the SJF strategy may perform only a little better than the FCFS schedule for a multiprocessor it might be interesting to see just how well the two stand up to each other.

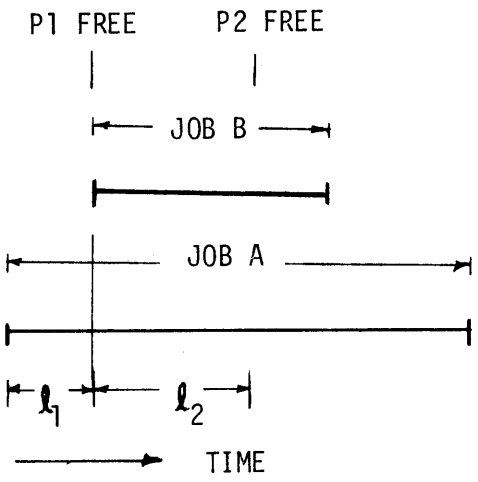


Figure 5.2. Short Job Vs. Long Job

A section of the next chapter describes the multiprocessor system performance results for the two schemes.

The aim here is to investigate the impact of job-mix characteristics on computer performance. This will be done by executing a set of pseudo-jobs or a synthetic workload on an actual system, CERBERUS. The pseudo-jobs do not perform any useful functions other than to simulate all the characteristics of a real life situation faithfully. The workload is parameterized in such a manner as to facilitate an easy variation of its characteristics. At the core of the experiments is a program called the Executive. This is a floating Executive, i.e. any processor may run this program. The Executive performs a number of functions. It keeps the waitlist of jobs ordered. It is also a process scheduler and job dispatcher since it extracts the job to be run next. The Executive also performs some other functions which are necessary only for performing evaluation experiments such as collecting job start delay statistics etc.

The following sections describe in detail the synthetic workload, the Executive and other simulation software.

5.2 Synthetic Job Mix

The synthetic workload executed in the performance evaluation experiments has been tailored specifically to reproduce only the periodic jobs encountered in a real time situation. Non-periodic one time jobs or other low priority tasks would normally be executed only when no periodic job is due, and a processor is available. Therefore our artificial workload is general enough for predicting system performance with respect to the periodic load. The total workload here consists of twenty five such periodic jobs. The reasons for selecting this number are as follows. This number is large enough to make the job-mix parameters, to be described shortly, significant. Second, the waitlist (a list of all jobs in the system) remains

small enough to be accommodated all at once in the scratchpad and still leave some room for instructions. There is, however, no limit on the number of jobs which can be handled as such. If the waitlist becomes large it can be divided into smaller parts. This would make the programs that process this file slightly more complex. The results obtained with 25 jobs would still be valid in such situations. Each job in this workload is defined completely by four parameters, i.e., number of instructions, iteration period, and two bus load parameters. These are described in the following paragraphs.

1. Number of Instructions

The number of instructions determines the length of the job. The pseudo-job executes a main loop of instructions K times. There are approximately ten instructions of the LOAD and STORE type (with direct addressing mode) in this loop. Thus the job length, KX10 instructions, may be varied by specifying the parameter K. A different value of K may be chosen for each of the 25 pseudo-jobs. This is true of the other three parameters also. As a matter of fact there are four tables, called the parameter tables, one for each parameter in the main memory, that must be filled with appropriate values before each run of the experiment. The system load can be varied from zero to one hundred per cent by changing the iteration frequency of the jobs.

2. Bus Load

Each processor in a multiprocessor system must obtain a certain number of words from the shared storage during the course of executing a job. If each processor has a copy of all the programs in its own dedicated memory, then the words to be fetched from the shared memory would only be data words. In case of partial replication of programs some instructions

would also have to be fetched. In the absence of any replication all program instructions must be read from the memory. The load imposed on the memory bus therefore depends upon the strategy used. The number of words fetched by each job in the artificial workload is determined by a pair of parameters L and M. Two parameters are needed to control the total bus traffic generated by each job as well as the length of each transaction. Each pseudo-job transfers a block of length L, i.e. L words of instruction/data, between the main memory and a processor scratchpad in one transaction. The frequency of this transaction is determined by M. The main loop of the pseudo-job described earlier, consisting of about ten instructions, is executed M times between the two main memory transactions of a job. The total volume of traffic thus generated on the bus is approximately K.L/M words for each job. For example, for K=450, L=128 and M=19 there will be 23 main memory transactions of 128 words each fetching a total of 2944 words during a job step.

3. Iteration Period

Each job is repeated at an interval of N time units. The time unit used here is the least count of the real time clock. This is the parameter that also controls computational load on the system for a given set of the other three parameters. The system load can be varied from zero to one hundred per cent by changing the iteration frequency of the jobs.

The overall job-mix characterics may, thus, be represented by the distribution functions of four parameters viz. K, L, M and N. For example, if all the parameters are assumed to have a normal distribution (the actual distribution function may of course be different for each parameter) we would describe the job-mix characteristics by four mean values and the corresponding standard deviations. The computer program, called the job program, for all the pseudo-jobs is the same. Only the parameter values are different. Therefore the address of the job in the waistlist actually

points to its values in the parameter tables. Once the Executive has obtained the address of the job to be run next it can read the values pointed to by the job address and substitute them directly into the job program. The parameter values, however, are transformed before substitution in order to save some preparatory time of each run of the experiment. The transformation is two-fold. The parameter value is multiplied by a constant and then added to another constant that may be positive, zero or negative. Thus there is a pair of global constants for each of the four parameters. This allows one to choose a desired distribution of each parameter by choosing appropriate values for these constants before each run without actually modifying the complete tables.

5.3 The Waitlist

The waitlist is an ordered list of all jobs waiting to be executed. Here the term 'waiting' does not imply already overdue jobs. Rather it means jobs that have been scheduled in advance to start at a certain time and which therefore must wait until they become due. The ordering of the list is done by the job start times alone. This is not the same as the First-Come-First-Serve scheduling policy. For example, if the job on top of the list is due ten units of time from now it would be executed first only if no other job is inserted in the list ahead of it during these ten time units. However, if two or more jobs are due at the same time (or overdue at a given instant) then one of the two schemes described will be used to choose the order of jobs. These are First-Come-First-Serve (FCFS) and Shortest-Job-First (SJF).

Each job entry in the waitlist occupies six words as shown in Figure 5.3. The first two words indicate the (double precision) absolute time at which the job is supposed to start. The next word in the job entry is divided into two half-words which contain the job address and the queue position. The job address, as descibed earlier, simply points to the relative position

HEAD OF WAITLIST	CURRENT TIME (Double Precision)	NOT USED	LAST JOB POINTER	FIRST JOB POINTER	PROCESSOR STATUS WORD
TYPICAL JOB ENTRY	JOB START TIME	JOB ADDR. | QUEUE POS.	BACKWARD POINTER	FORWARD POINTER	START DELAY
FIRST JOB	JOB START TIME	JOB ADDR. | QUEUE POS.	0	FORWARD POINTER	START DELAY
LAST JOB	JOB START TIME	JOB ADDR. | QUEUE POS.	BACKWARD POINTER	0	START DELAY

Fig. 5.3. The Waitlist.

of parameter values of the job in the parameter tables. The queue position will be explained a little later. The next two words point to entries corresponding to jobs that are due immediately before and after this job respectively. That is, each job is linked to its predecessor as well as successor through pointers or links. The job due first has no predecessor. Therefore its backward pointer is null. Similarly the forward pointer of the last job is also null.

It is necessary to chain the waitlist, at least in one direction, to keep the list ordered since the job entries themselves always retain their physical places in the list and are not moved around. Also it is sufficient to chain the list only in one direction to process the file either to obtain the first job or to insert another job in the list at its appropriate place. However by chaining the list in both directions the search can proceed either from top to bottom or vice versa. Before starting the search, if we knew approximately that the correct position of the new job to be inserted in the list is in the lower half of the list we would then begin to search from the bottom. If the opposite were true the search would begin from top. However it is necessary to know the approximate position of the job in the queue in order to cut down on the file search time. This is where the job queue position indicated by half of the second word of each job entry comes into play. It tells us the queue position where the same job was inserted last time. Jobs that have a relatively smaller iteration period compared to the average iteration period of all jobs are always likely to be inserted towards the top of the list. The opposite is true of jobs with relatively larger time periods.

The last word in the job entry keeps track of the job starting delay. The time at which the Executive picks up the first job is recorded as the actual starting time of the job even though the job does not get started until some time later when the Executive terminates. This job starting time is used to compute the starting delay of each job step. A job step is defined here as one iteration of a job. The first entry in the waitlist is

the head of the waitlist. The first two words in this entry indicate the double precision absolute time. This time is updated at the beginning of each Executive run. The next word is not used. The next two words are the pointers to the first and the last job in the queue. The last word, the Processor Status Word (PSW), indicates the status of the processor controlling the Executive. This word indicates one of the two modes: IDLE or EXECUTIVE. This will be explained in greater detail in the following section.

5.4 The Executive

The Executive program actually performs all those functions which would be called for in a real system as against pseudo-jobs which only consume various system resources but do not perform any real function. In addition the Executive also collects certain system performance statistics. Since most of the Executive activity is concerned with manipulating the waitlist only one processor can be allowed to run the Executive program at any given time. Therefore there is a lock associated with this program. As described earlier this lock is automatically administered by the microprogram. Upon executing the last instruction of a pseudo-job which is always 'ENDJOB' the processor begins the Executive program if it is available or else waits until the program does become free. The major functions of the Executive in a real time application are:

1. Accept requests for jobs,

2. dispatch jobs and

3. keep an ordered list of all jobs.

It would also need to keep track of all the resources if it were a dedicated Executive. However since this is a floating Executive that function is obviated.

In general the first two functions listed above are quite independent. In other words, when a processor becomes free it runs the Executive to obtain the job due next. This is the job dispatch function. The Executive must also insert job requests in the waitlist whenever such requests are received. Most often a periodic job would place a request for its next iteration some time during its present run. It may also place requests for other jobs, and occasionally there may be such a request from the crew or the operator. In the present set of experiments some of these fine points have been ignored. It is assumed here that each periodic job calls itself by placing a request in the waitlist and each job places one and only one such request. The total number of jobs in the system therefore remains constant. The Executive functions, then, may be briefly restated as follows:

1. Obtain address of the job due next,

2. place a request for its next iteration in the waitlist and

3. reorder the waitlist.

When a processor begins to run the Executive it must first of all find out if a job is ready to run. To determine this the processor obtains the starting time of the job due next and compares it to current absolute time. If this is greater than the current time the job is not due yet. In this case the processor has a choice of two actions. It could relinquish the control of the Executive and go into a wait loop before coming back to the Executive. Alternatively, it may retain control of the Executive until the job becomes due. The processor should release the Executive program if the program might be needed by another processor for a purpose other

than obtaining the next job. For example, another processor might want to place a request for a new job in the waitlist. However, as outlined earlier every job calls itself before starting out and no other job requests are placed in the waitlist. Therefore in this case it would be unnecessary to release the Executive. This is the strategy adopted in the simulation. That is, the processor retains control of the Executive and waits until the first job becomes due.

During this time the processor, even though it controls the Executive, is really idling. Therefore it switches its status in the Processor Status Register to 'IDLE'. Also, as described in the preceding section, there is a Processor Status Word (PSW) in the main memory. This word indicates the status of the processor controlling the Executive. The processor switches the PSW to 'IDLE'. Therefore when another processor becomes free and finds the Executive locked it (the new processor) would switch itself into 'IDLE' mode. It would change its status from 'IDLE' to 'WAIT EXECUTIVE' when the processor controlling the Executive switches the PSW from 'IDLE' to 'EXECUTIVE'.

If the first job in the waitlist, pointed to by the first job pointer in the head of the waitlist, is overdue the processor records the delay in starting the job if using the FCFS policy. For SJF schedule it looks through the waitlist to compare job lengths of all jobs overdue and determine the address of the shortest job. Having done this it obtains the job parameter values, pointed to by the job address, from the parameter tables. One of these parameters is the job iteration period. The processor transforms the iteration period (as well as other parameter values) and computes the time for the next cycle of the job by adding the iteration period to the time the job was supposed to start. This way a delay in one cycle of the job does not result in a loss of an iteration of the job. This is necessary to maintain the required frequency of iteration of the job.

To reorder the waitlist an appropriate search procedure as indicated by the job queue position is begun. The pointers in the job entries are adjusted when the new position for the job is found and the queue position also is updated accordingly. The processor then transforms and substitutes the job parameter values K, L, M in the job program to compose a pseudo-job ready to run. The last instruction of the Executive is 'BEGIN'. This releases the Executive and the processor begins to execute the pseudo-job.

CHAPTER VI

EXPERIMENTAL RESULTS

6.1 Introduction

Various aspects of the performance evaluation experiments have been described in the preceding chapters. These include the experimental facilities of the CERBERUS multiprocessor and associated instrumentation, the simulation workload and the job dispatch strategy. To recapitulate briefly, the workload, or the job mix, is composed of twenty-five periodic jobs. That is, all the jobs are repetitive and due at specific times. The priority of a job is determined by the time at which it is supposed to start. Two strategies are used to dispatch jobs due at the same time: i) First-Come-First-Serve and ii) Shortest-Job-First. Each job is characterized by four parameters: the job-length, the job iteration period and the magnitude and distribution of the memory bus traffic generated by the job. Values of all these parameters are chosen prior to each run of the experiment. There is a single ordered waitlist which contains pertinent information about each job step, e.g. its starting time, a pointer to its parameters etc.

Two different kinds of statistics were gathered from the simulation experiments. The first type is concerned with system utilization. All the results concerning the breakdown of processor time have been collected under the broad heading of system utilization. The second type concerns the distribution of job starting delay.

It is convenient at this point to define a few parameters which are used hereafter to describe the results. First of all, the per cent system load is defined as (100 minus per cent idle). In other words the per cent load includes the per cent of time a processor is busy running a job step or the Executive program, waiting for the Executive or the memory bus and fetching instructions/data from the memory. The load on the system is increased by increasing the job dispatch rate, that is, by decreasing the iteration period of the jobs. Another interesting parameter is the ratio, R, of the mean job step length to the mean executive run time. This is a normalized measure of the mean job step length. The third definition concerns the measure of job starting delay. We define normalized delay as the delay measured in units of mean job step lengths.

In the CERBERUS experimental facility, the bus bandwidth is fixed at a value such that the rate at which instructions can be fetched by a processor is only slightly greater than the rate at which programs are executed. Therefore the maximum possible throughput efficiency for this series of experiments is limited to slightly over sixty per cent. The throughput efficiency is defined here as the percentage of time a processor is busy doing useful computation, i.e., running a job step, when the system is saturated. At the saturation point there is no idle time and the load equals one hundred per cent. The throughput efficiency may also be defined, for loads less than a hundred per cent, as the per cent processor time devoted to useful job step computation at a given load factor. The following sections discuss the results obtained from the simulation experiments. All the results of the next two sections pertain to the FCFS job dispatch strategy. Section 6.4 compares the system performance for the FCFS and the SJF schedules.

6.2 System Utilization

The various measured criteria of performance included the percentages of processor time devoted to:

1. useful job step computation,

2. wait for and use of memory bus,

3. wait for the Executive program,

4. use of the Executive program and

5. idle time.

It was also found that useful job step computation rate is nearly linearly dependent on the job dispatch rate for the three values of R (1, 10, 50) investigated. This is shown in Figure 6.1. The same is true of the overhead functions, that is, all functions other than useful job step computation and idling, as shown in Figure 6.2. The breakdown of overheads into the three components mentioned above is shown in Figures 6.3, 6.4 and 6.5 for R=1, 10 and 50 respectively. The effects of saturation are found to be not strong. This indicates that there is essentially no degradation in multiprocessor performance as far as system utilization is concerned as the job step dispatch rate is increased. That is, the ratio of percent job step computation to overheads remains nearly constant.

The probability distribution of iteration period was Gaussian for this set of experiments. As mentioned in an earlier chapter, the system load is varied by changing the job iteration periods. Since the mean job length is different for each value of R a different value of mean iteration period will be required to obtain a given load factor. That is, the range of variation of the mean iteration period to obtain the full range of load factors is different for each value of R. These values of mean iteration periods and the corresponding load factors are tabulated in Table 6.1. The distribution of job step length was also Gaussian in all the cases, the mean values being 4.5, 45 and 225 milliseconds for R=1, 10 and 50 respectively. The

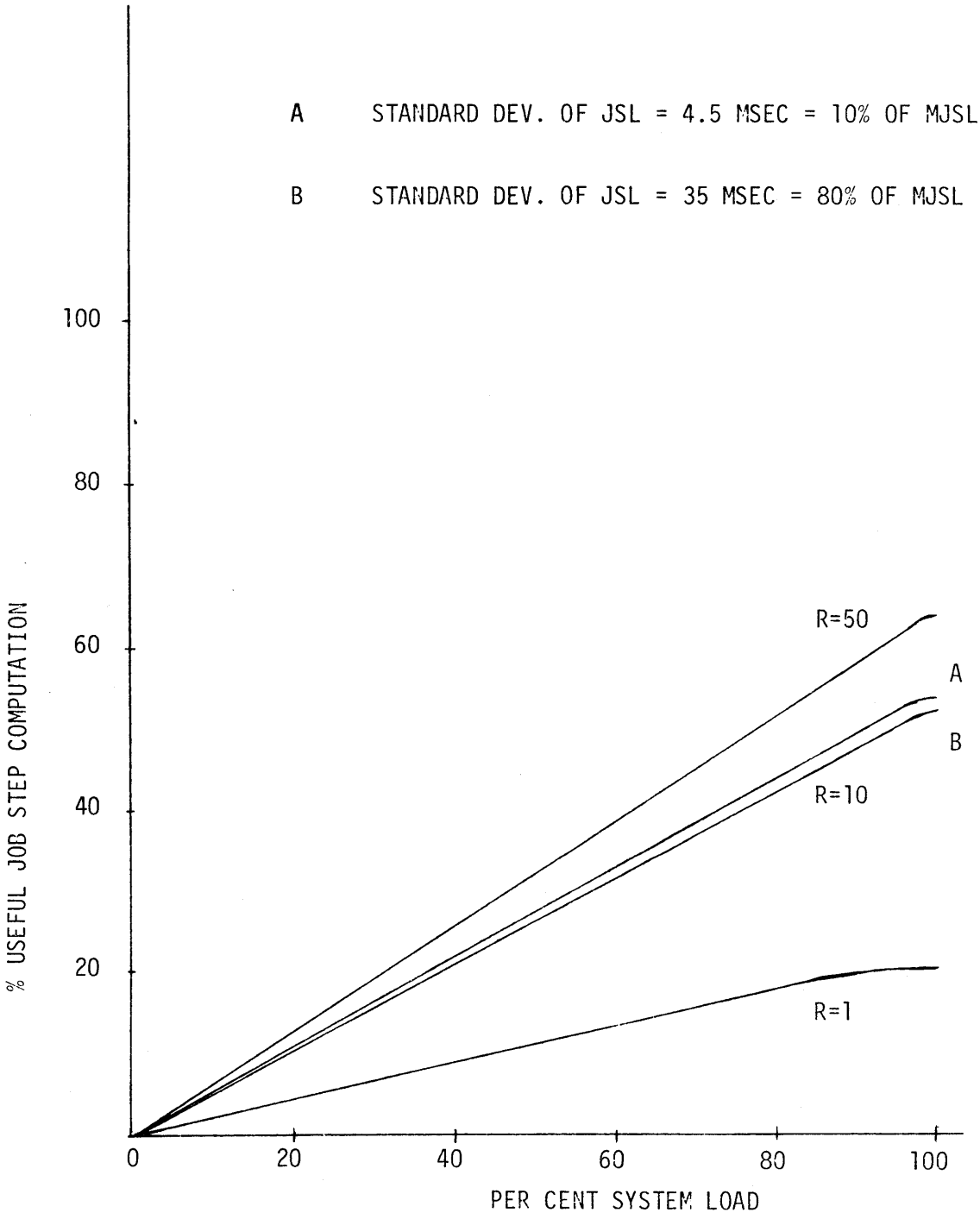


Figure 6.1. Throughput Efficiency Vs Load.

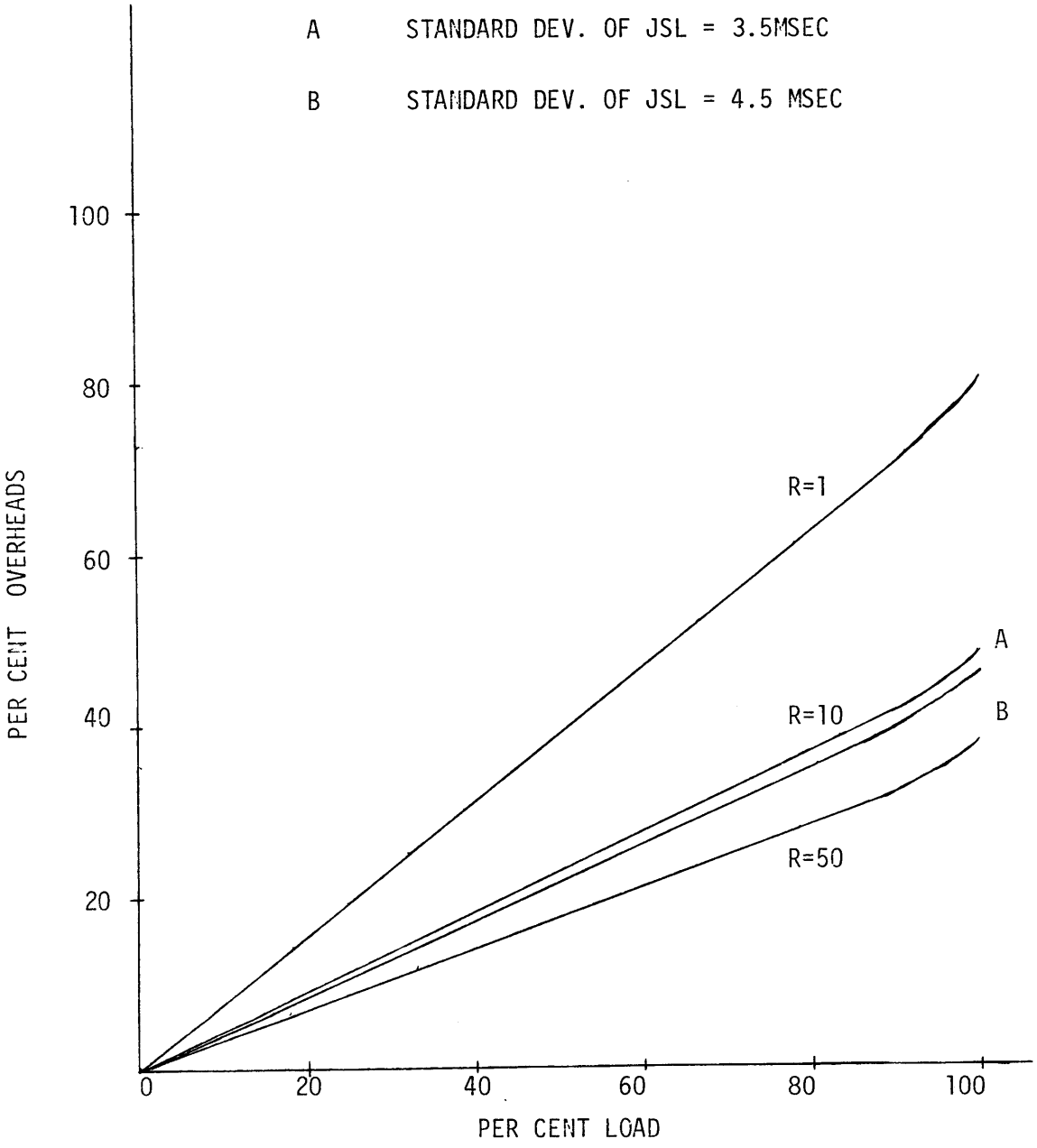

A STANDARD DEV. OF JSL = 3.5MSEC
B STANDARD DEV. OF JSL = 4.5 MSEC
PER CENT OVERHEADS
100
80
60
40
20
0
20
40
60
80
100
PER CENT LOAD
R=1
R=10
R=50
A
B

Figure 6.2.

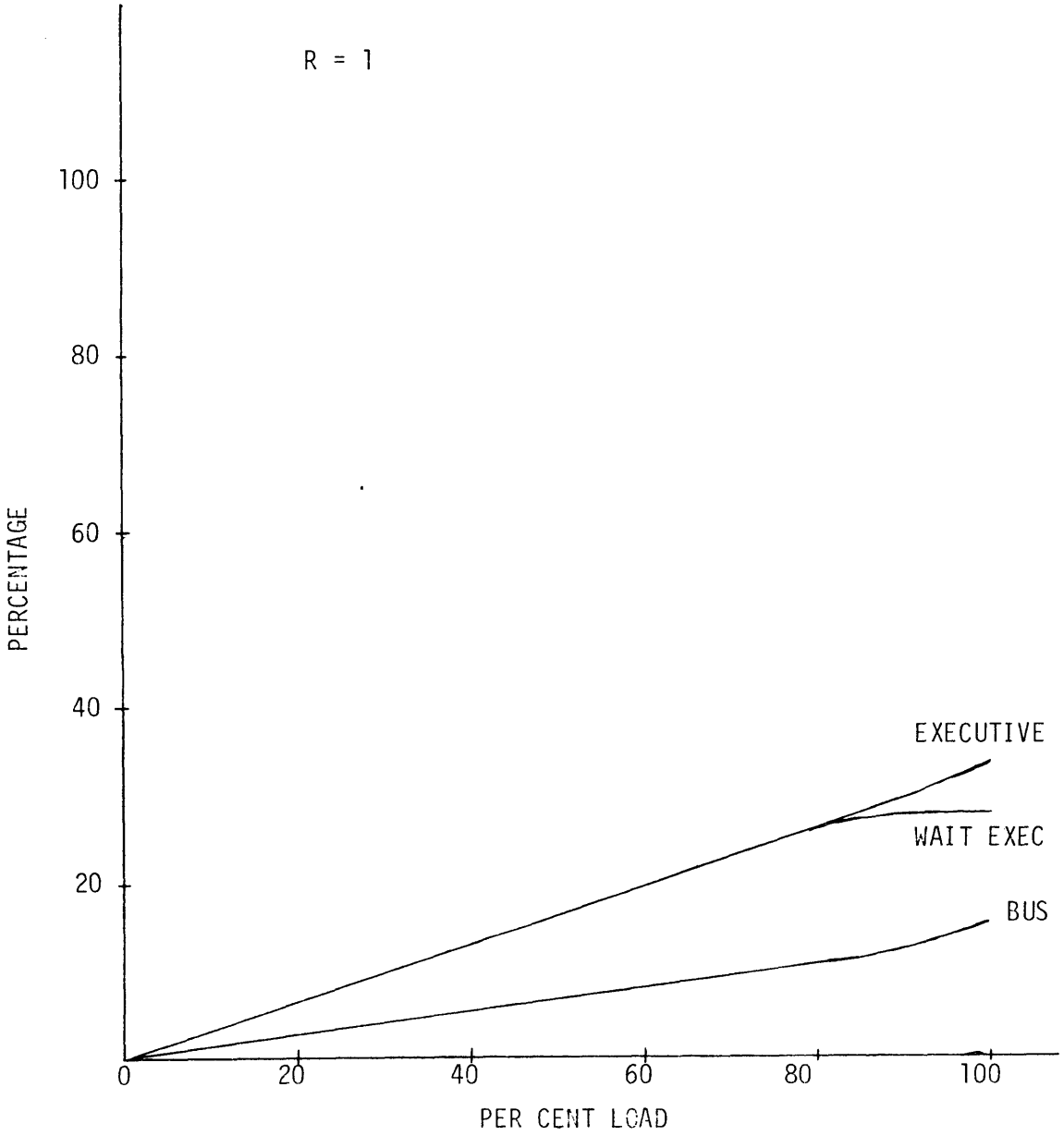

R = 1
100
80
60
40
20
PERCENTAGE
EXECUTIVE
WAIT EXEC
BUS
0
20
40
60
80
100
PER CENT LOAD

Figure 6.3.

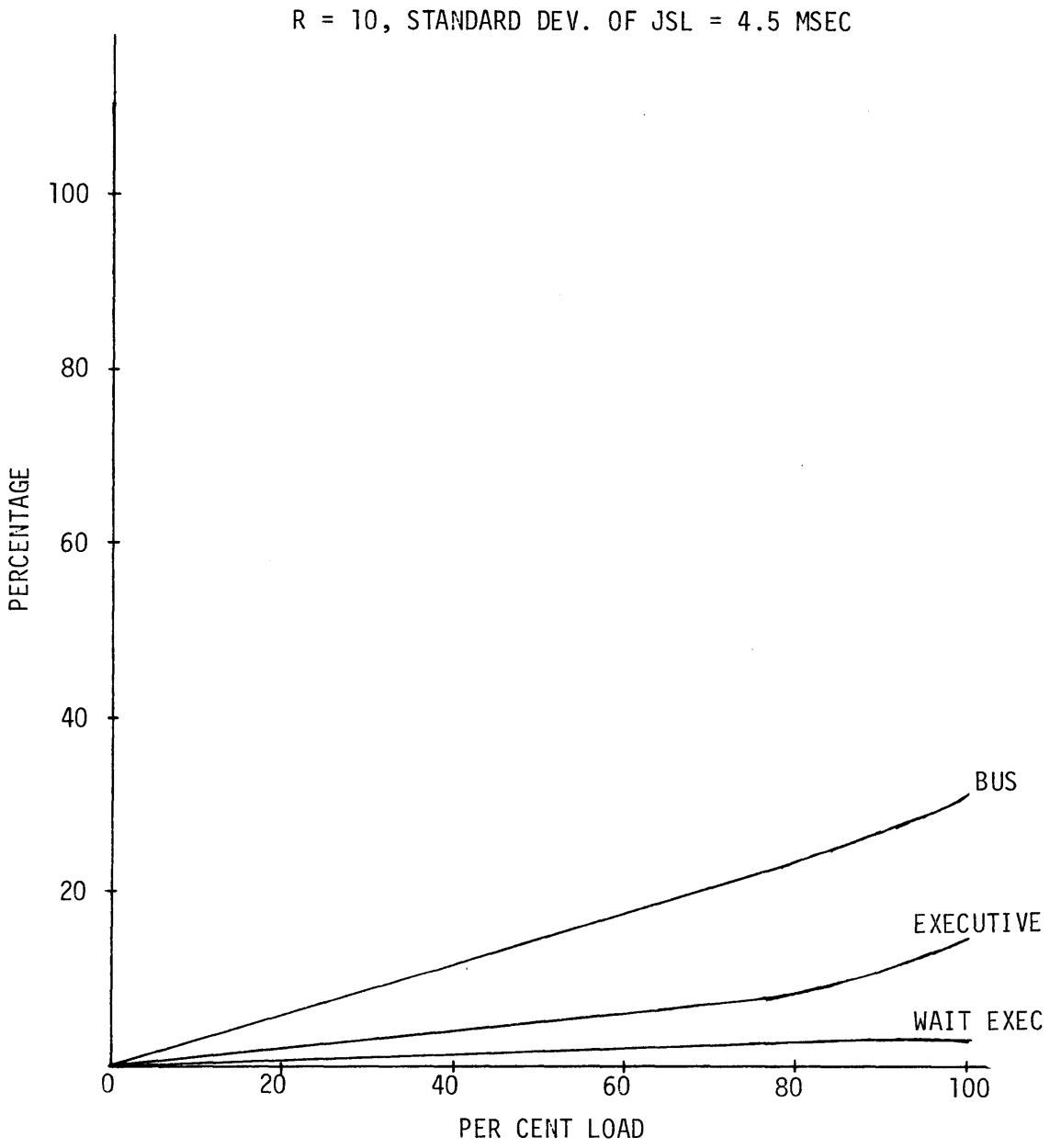

R = 10, STANDARD DEV. OF JSL = 4.5 MSEC
PERCENTAGE
100
80
60
40
20
0
20
40
60
80
100
PER CENT LOAD
BUS
EXECUTIVE
WAIT EXEC

Figure 6.4A.

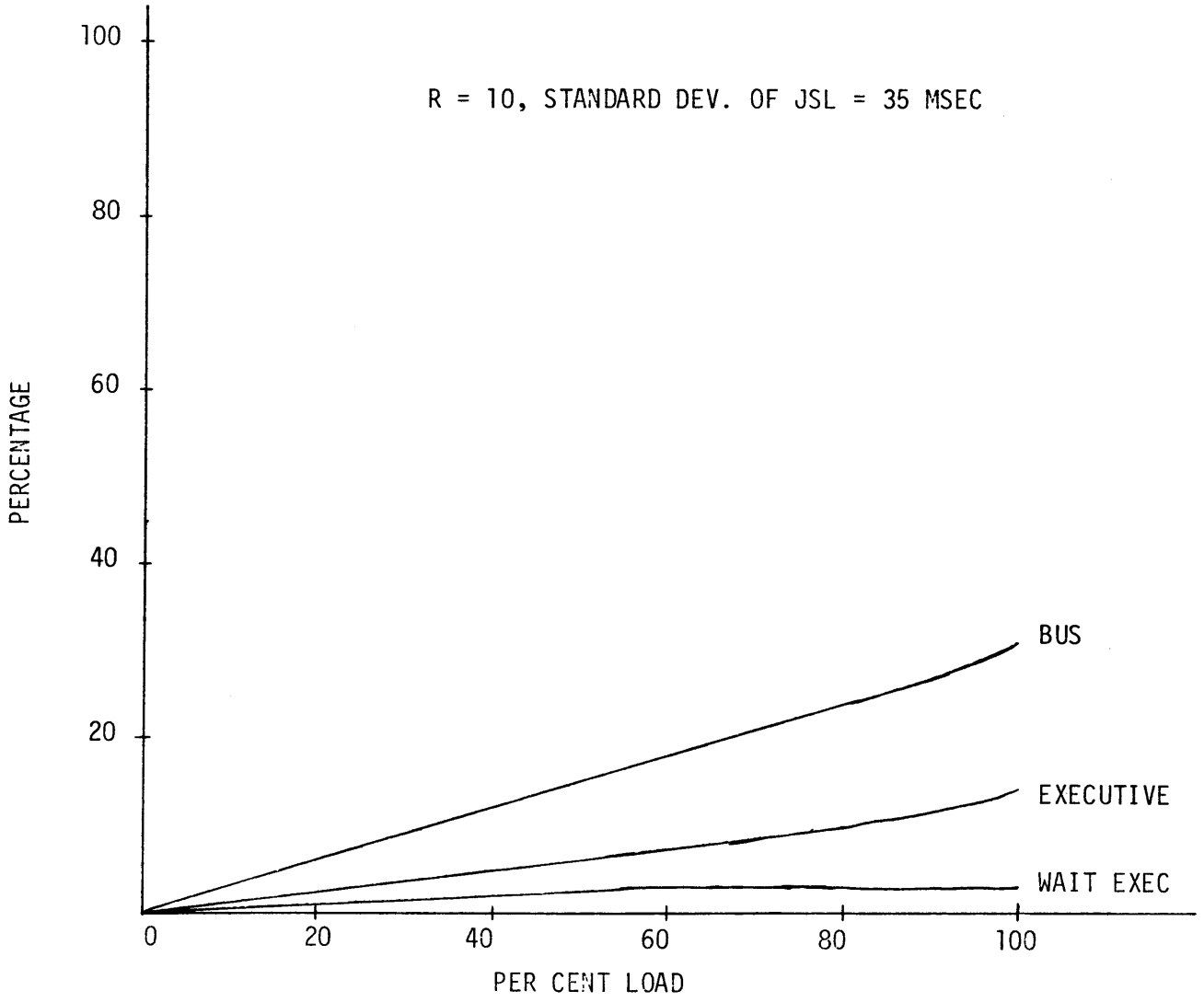

R = 10, STANDARD DEV. OF JSL = 35 MSEC
100
80
60
40
20
PERCENTAGE
BUS
EXECUTIVE
WAIT EXEC
0
20
40
60
80
100
PER CENT LOAD

Figure 6.4B.

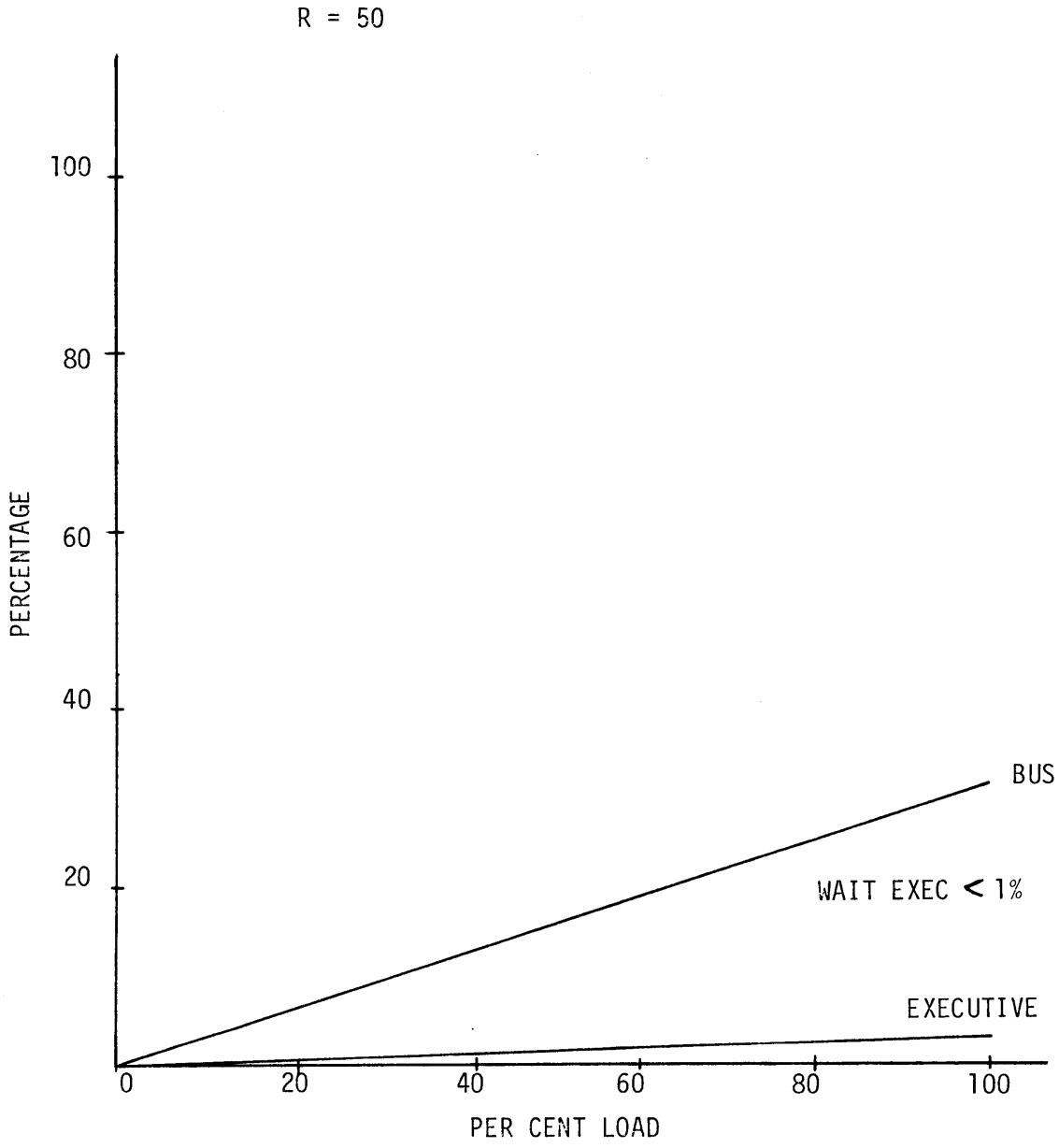

R = 50
100
80
60
40
20
PERCENTAGE
BUS
WAIT EXEC < 1%
EXECUTIVE
0
20
40
60
80
100
PER CENT LOAD

Figure 6.5.

TABLE 6.1

R	LOAD %	MEAN ITERATION PERIOD MSEC
1	3	10.5
1	98	0.3
10	7	10.5
10	99	0.8
50	26	12.0
50	98	3.2

magnitude of memory bus traffic was constant for all jobs and was arrived at by assuming a ratio of approximately 1.5 for the number of instructions executed to the number of instructions fetched. This ratio has been obtained by Green et al [15] by analyzing Apollo Guidance Computer jobs. For R=10, for instance, the job length is 45 milliseconds. Therefore the job must execute 4500 instructions, each instruction being 10 microseconds long, and hence must fetch 3000 instructions from the main memory. The data words per job, as found by Mallach [28], are negligible compared to the number of instructions. The bus traffic was divided into blocks of 128 words and distributed uniformly over the entire length of the job.

To study the impact of the job length distribution on the system performance the standard deviation of the normally distributed job step length was increased from 4.5 milliseconds (10% of MJSL) to 35 milliseconds (80% of MJSL). Qualitatively the wider distribution caused the per cent overheads to increase and the per cent useful job step computation to decline as shown in Figures 6.1 and 6.2. The quantitative impact, as shown in the same figures, was, however, not significant at all.

Figure 6.6 shows throughput efficiency as a function of R for various values of load factors. It is seen that when the job lengths and the executive length are comparable (R=1), the multiprocessor throughput efficiency is approximately 40 per cent of its value when the executive length is an order of magnitude less than the job length (R=10). Further shortening of the Executive produces diminishing returns as evidenced by the low slopes corresponding to R>10. Shortening of the Executive is equivalent to making the job steps longer as far as the system performance is concerned. The only important parameter is the ratio of the mean job length to the mean executive length.

The efficiency of the executive program becomes an important issue when job step lengths are shortened to a point comparable to the executive length. For one thing, the executive program itself consumes a large

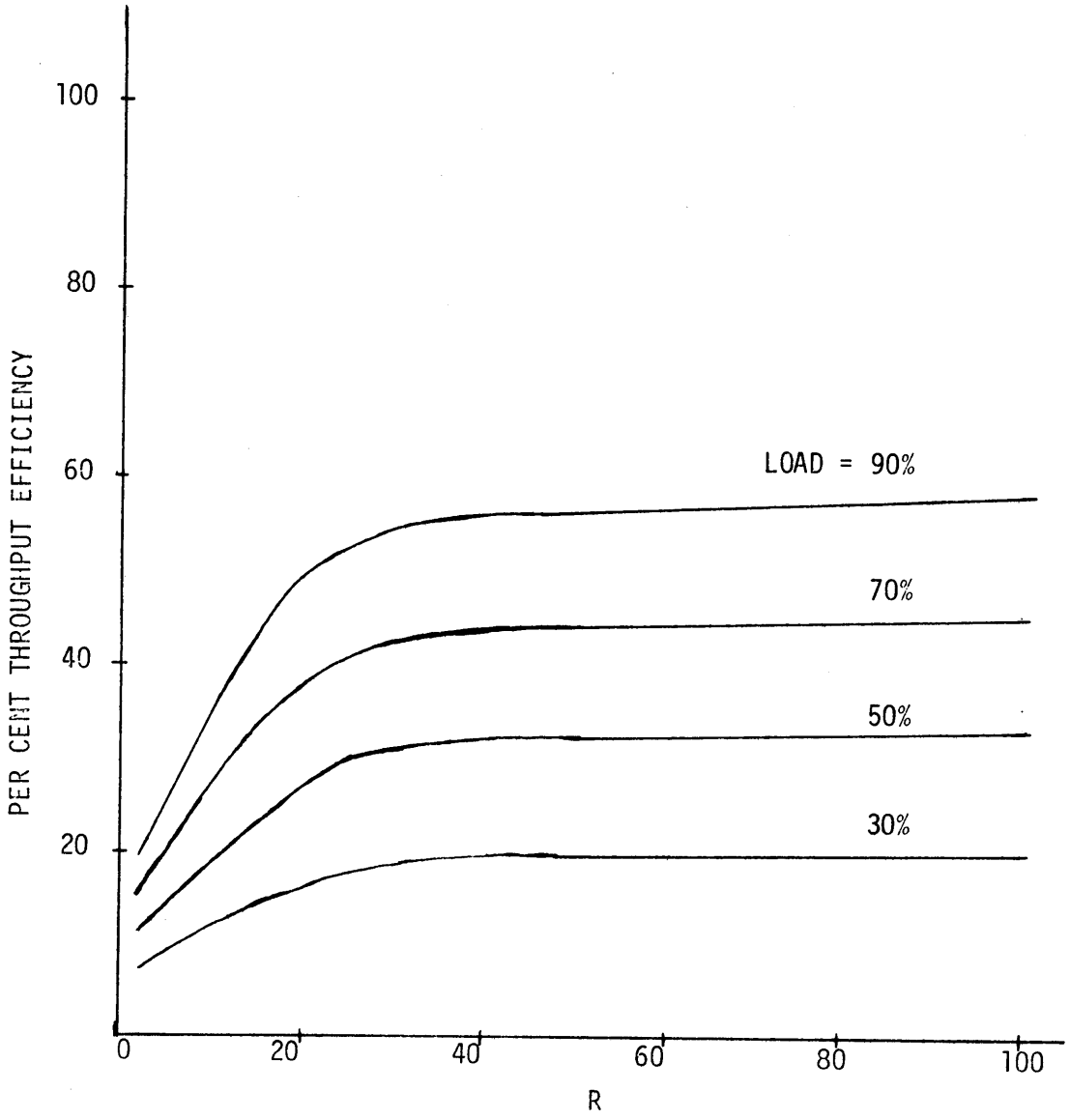


Figure 6.6. Throughput Efficiency Vs R.

proportion of available resources. More significant, however, is the impact of the executive's lockout of other processors waiting for a new job step. With only three processors running, executive lockout accounts for about 27 per cent of all processor time at full load when R=1. With more processors running, this would clearly rise. When job steps are longer, executive lockout time is less of an issue. When R=10, it drops to 3 per cent at full load, and when R=50 it is down to 0.5 per cent.

The remaining overhead time is spent in waiting for the memory bus and fetching the programs. It is assumed that all the programs reside in the shared storage module. If, however, each processor were to have a copy of all the programs the throughput efficiency could be increased significantly. This is clearly demonstrated by the results shown in Figures 6.4a and 6.7. Figure 6.4a shows for R=10 the per cent processor time devoted to job step computation when all programs must be read from the main memory. Figure 6.7 shows the same performance criterion for the same value of R when each processor is provided a copy of all the programs. The throughput efficiency at saturation increases from 55% to slightly over 80% in this case.

6.3 Job Starting Delay Distribution

The importance of job starting delay as a performance criterion in a real time system can not be overemphasized. Extensive measurements regarding the probability distribution of the delay were made in the simulation experiments.

Figure 6.8 shows the normalized mean job starting delay as a function of the load, i.e., the normalized job step dispatch rate. The normalized delay decreases as the ratio of the mean job step length to executive running time is increased. This is due to the fact that the higher the ratio R, the lower is the probability of executive access conflict between processors.

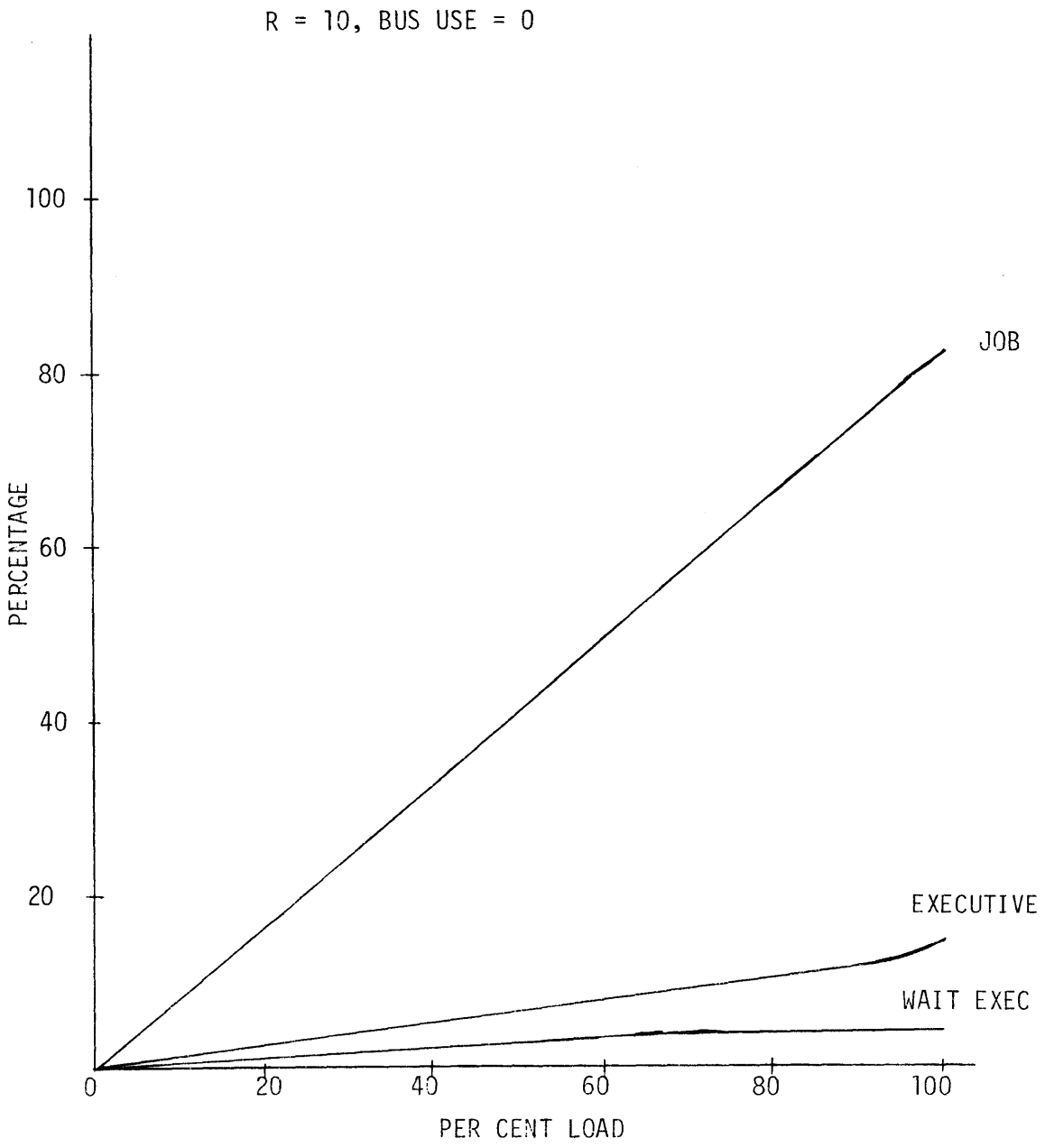

R = 10, BUS USE = 0
100
80
60
40
20
PERCENTAGE
JOB
EXECUTIVE
WAIT EXEC
0
20
40
60
80
100
PER CENT LOAD

Figure 6.7.

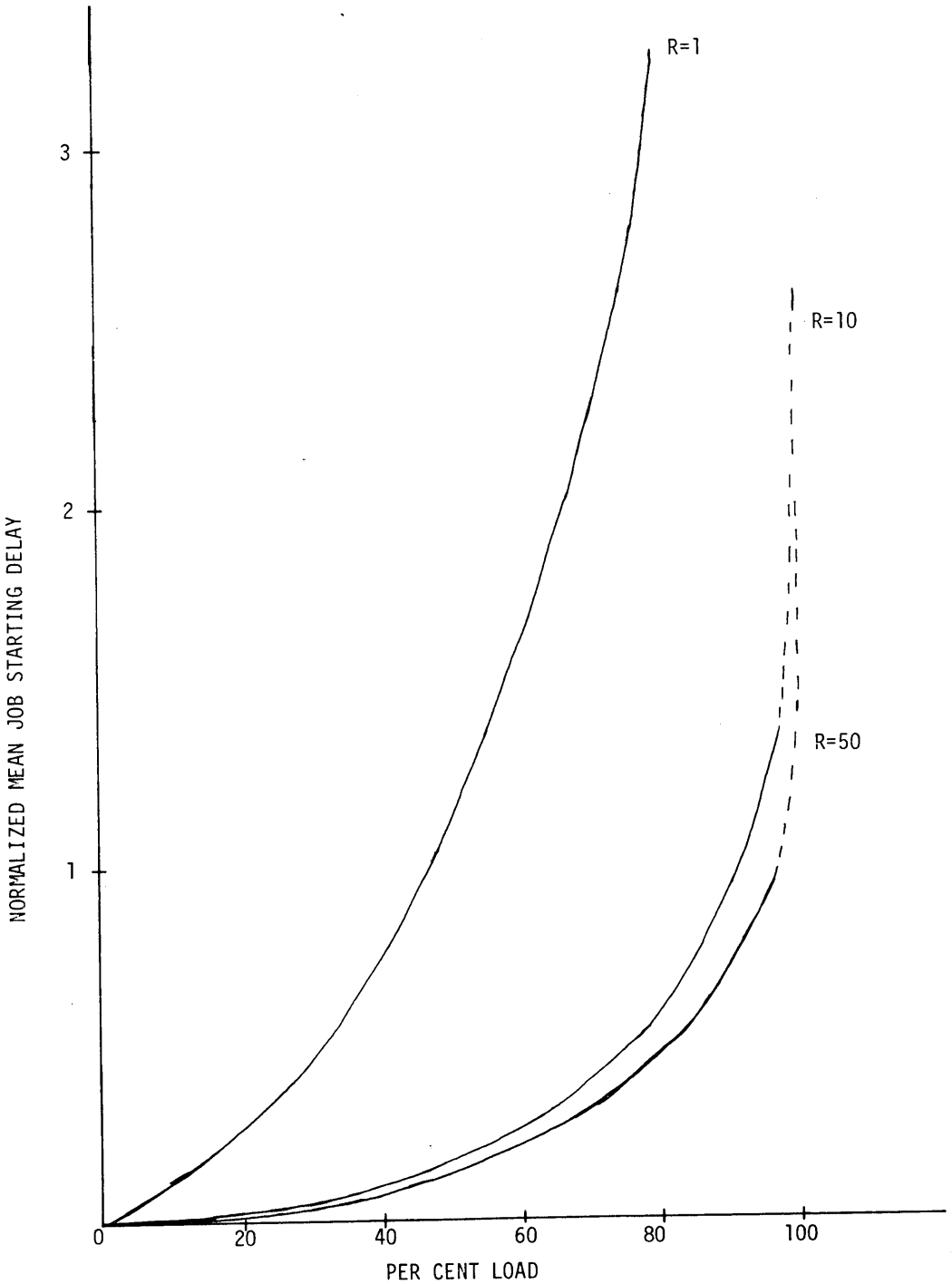


Figure 6.8. Normalized Delay Vs Load.

For each value of R the mean delay function increases without bound as the load approaches saturation, as found by Mallach [28] and Rosenberg [35].

In some real time applications the important performance criterion may be the absolute rather than the normalized measure of mean starting delay. In other words the maximum limit on the mean delay may be specified in time units. Therefore it is instructive to look at the mean delay statistics from this point of view also. Figure 6.9 shows the mean delay, measured in milliseconds, as a function of the mean job step length for three values of the system load. It is seen that a linear relationship exists between these two parameters, with the slope of the function increasing rapidly with the system load. Also, while the normalized delay decreases with R the absolute delay increases with R.

The next set of figures shows the probability distribution function (PDF) and the cumulative distribution function (CDF) histograms for a number of load factors for the three values of R. The number of samples, that is, the number of job steps from which each of these delay histograms is derived is different for each value of R and load factor. This is due to the fact that the run time for these simulation experiments was held constant at approximately ten minutes. Therefore the number of samples was in the range of several thousand for R=50 while it was several tens of thousand for R=1. While Figures 6.10, 6.11 and 6.12 show the actual histograms obtained from the experiments, the next two Figures (6.13 and 6.14) depict the smoothed qualitative versions of the PDFs for R=1 and 50 respectively.

Referring to Figures 6.13 and 6.14, for low system loads the delay distribution can be described as a monotonically decreasing function with a very sharp initial decline indicating that a large percentage of jobs start without an appreciable delay. These characteristics are mostly retained for loads approaching ninety per cent. Beyond that, however, the function shape changes considerably and for 99 per cent load the probability function,

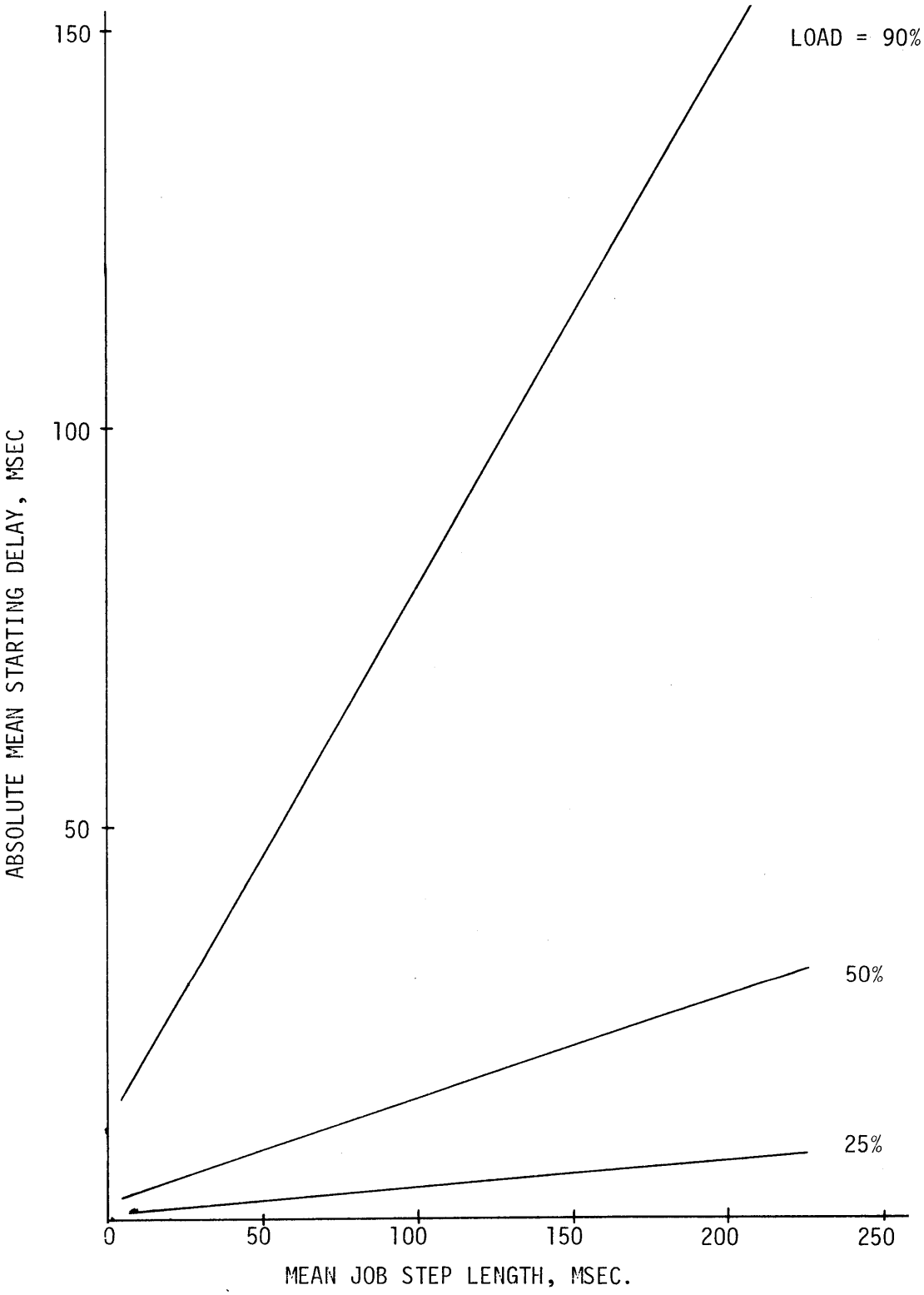

150
100
50
ABSOLUTE MEAN STARTING DELAY, MSEC
LOAD = 90%
50%
25%
0
50
100
150
200
250
MEAN JOB STEP LENGTH, MSEC.

Figure 6.9.

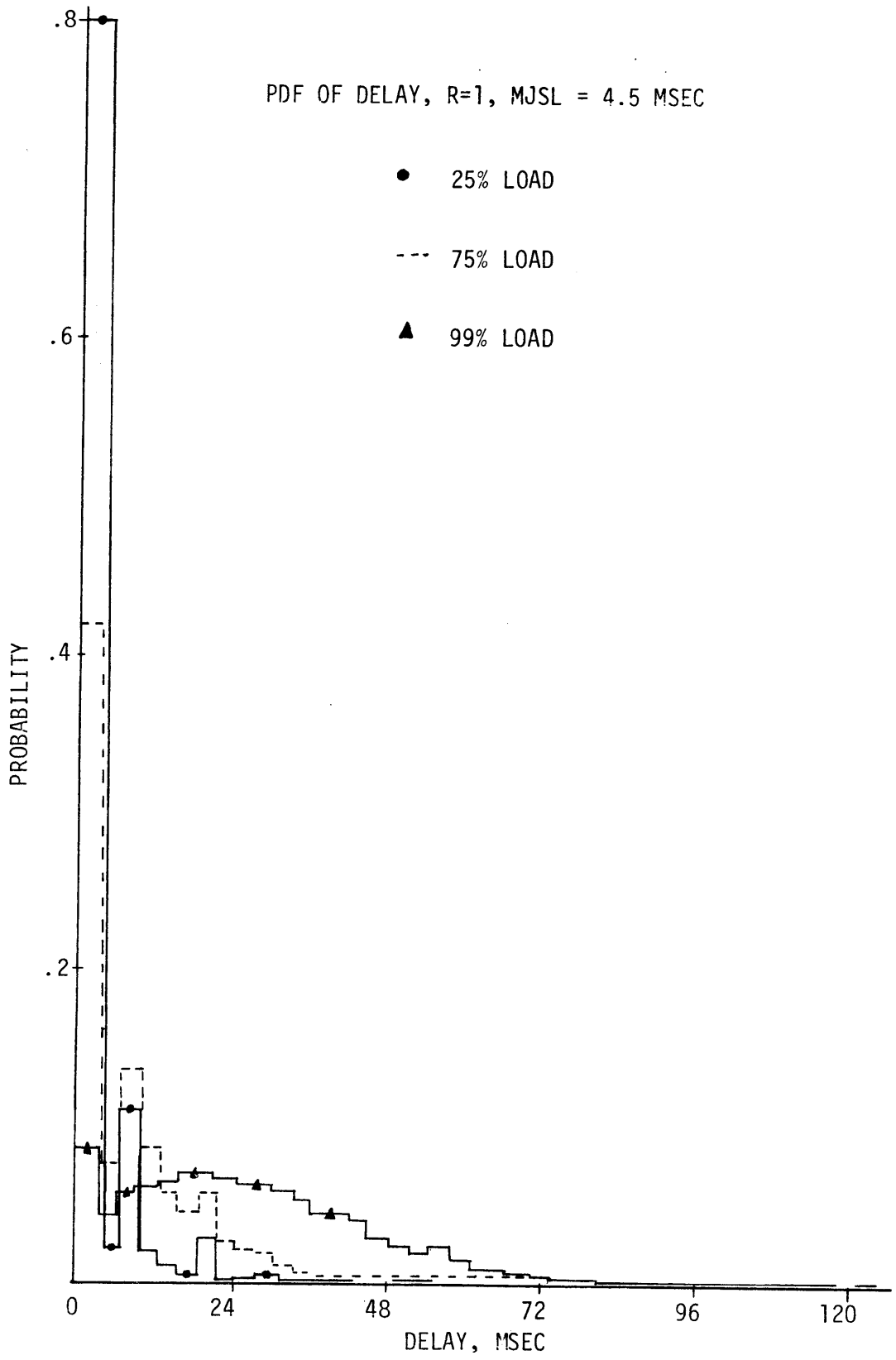

PDF OF DELAY, R=1, MJSL = 4.5 MSEC
25% LOAD
75% LOAD
99% LOAD
PROBABILITY
.8
.6
.4
.2
0
24
48
72
96
120
DELAY, MSEC

Figure 6.10A.

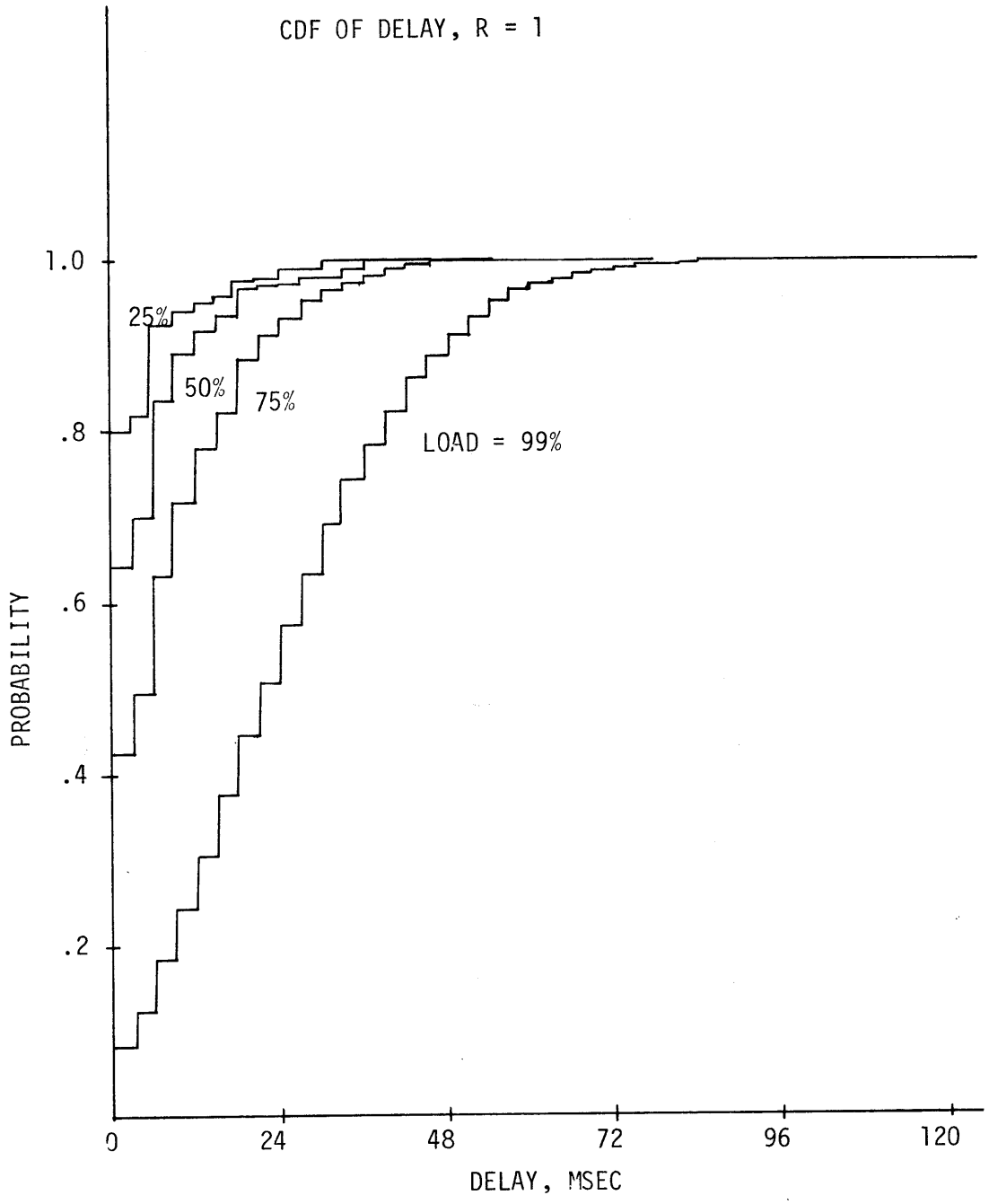

CDF OF DELAY, R = 1
1.0
.8
.6
.4
.2
PROBABILITY
25%
50%
75%
LOAD = 99%
0
24
48
72
96
120
DELAY, MSEC

Figure 6.10B.

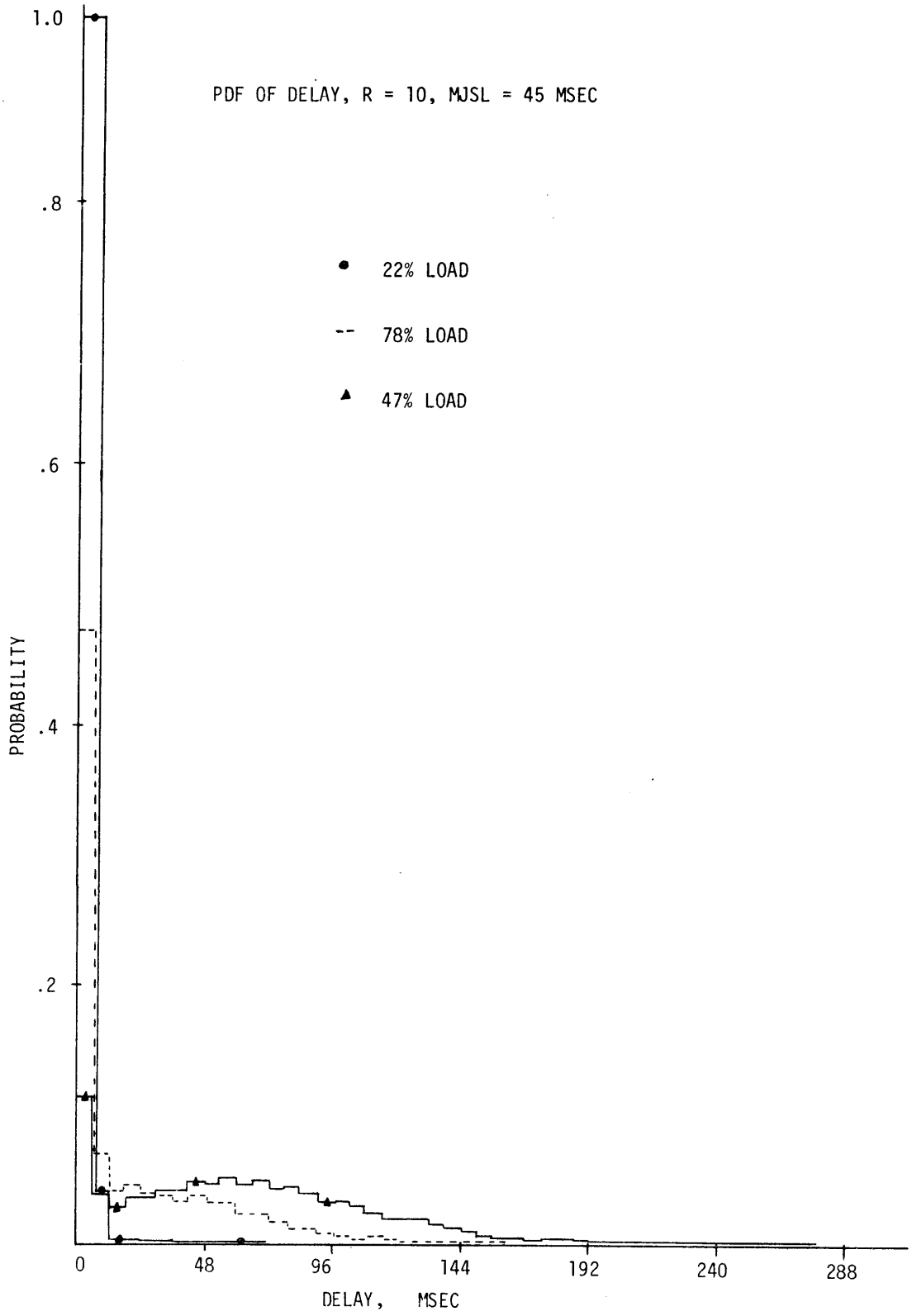

PDF OF DELAY, R = 10, MJSL = 45 MSEC
22% LOAD
78% LOAD
47% LOAD
PROBABILITY
1.0
.8
.6
.4
.2
0
48
96
144
192
240
288
DELAY, MSEC

Figure 6.11A.

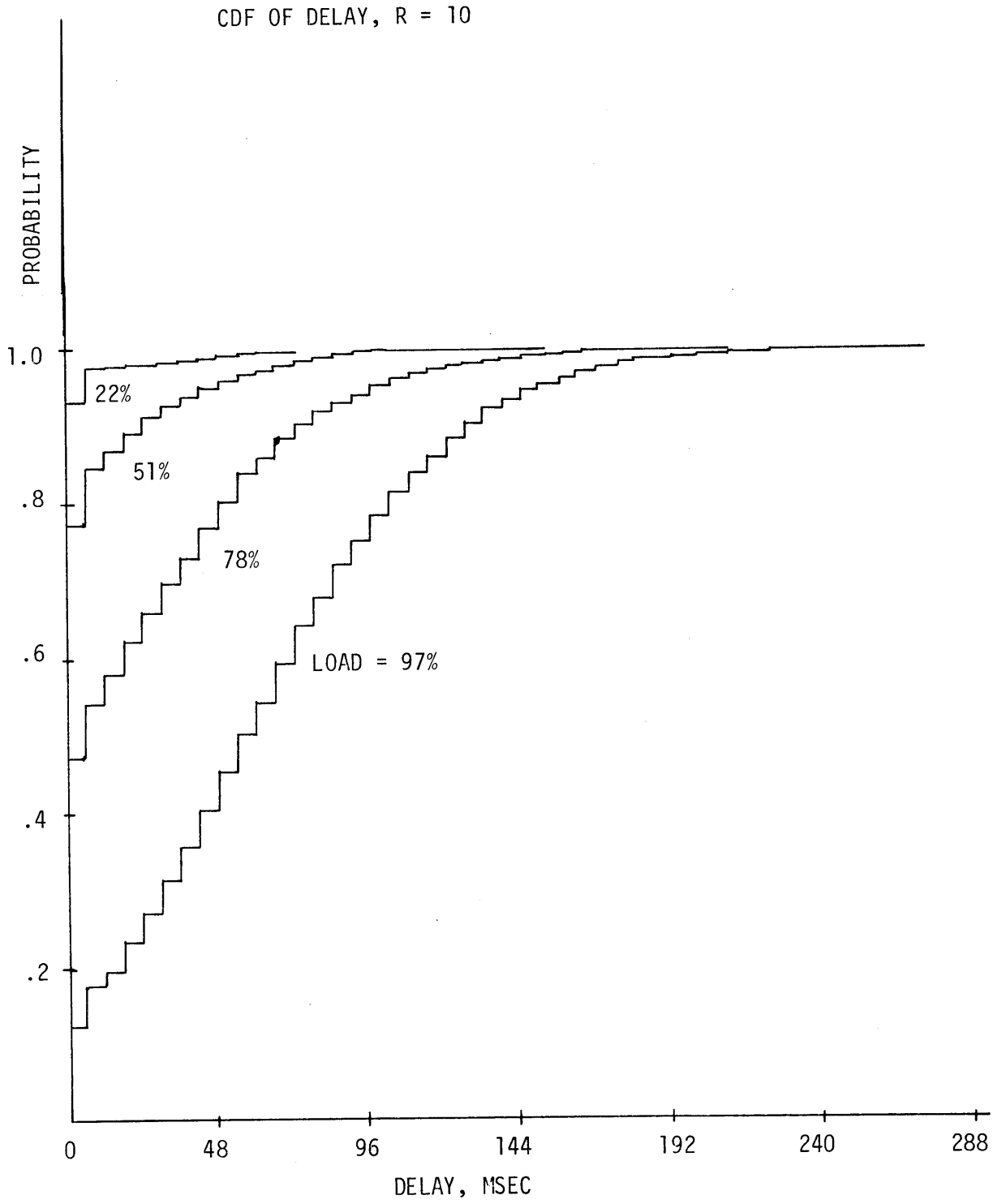

CDF OF DELAY, R = 10
PROBABILITY
1.0
.8
.6
.4
.2
22%
51%
78%
LOAD = 97%
0
48
96
144
192
240
288
DELAY, MSEC

Figure 6.11B.

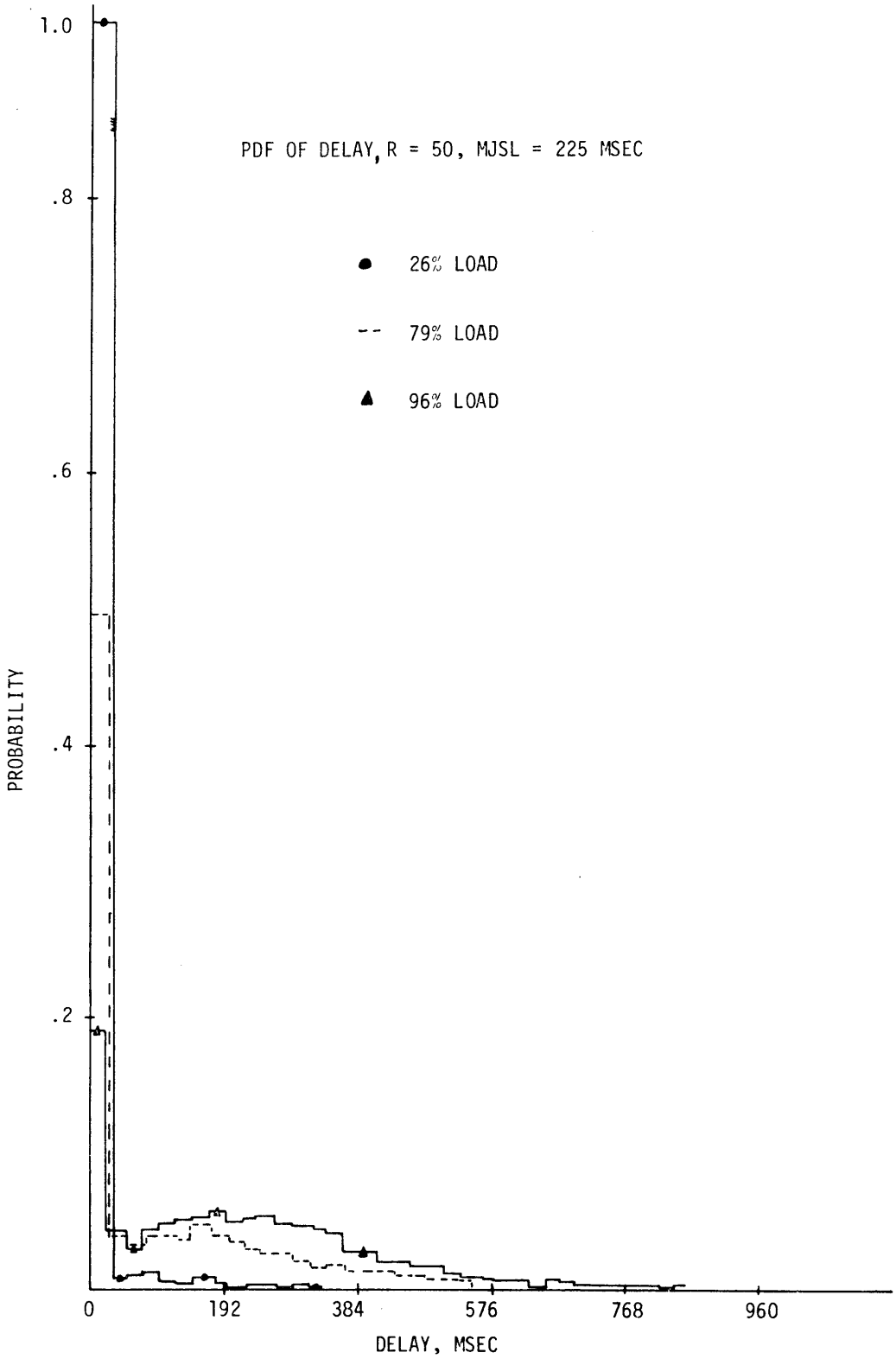

PDF OF DELAY, R = 50, MJSL = 225 MSEC
26% LOAD
79% LOAD
96% LOAD
PROBABILITY
1.0
.8
.6
.4
.2
0
192
384
576
768
960
DELAY, MSEC

Figure 6.12A.

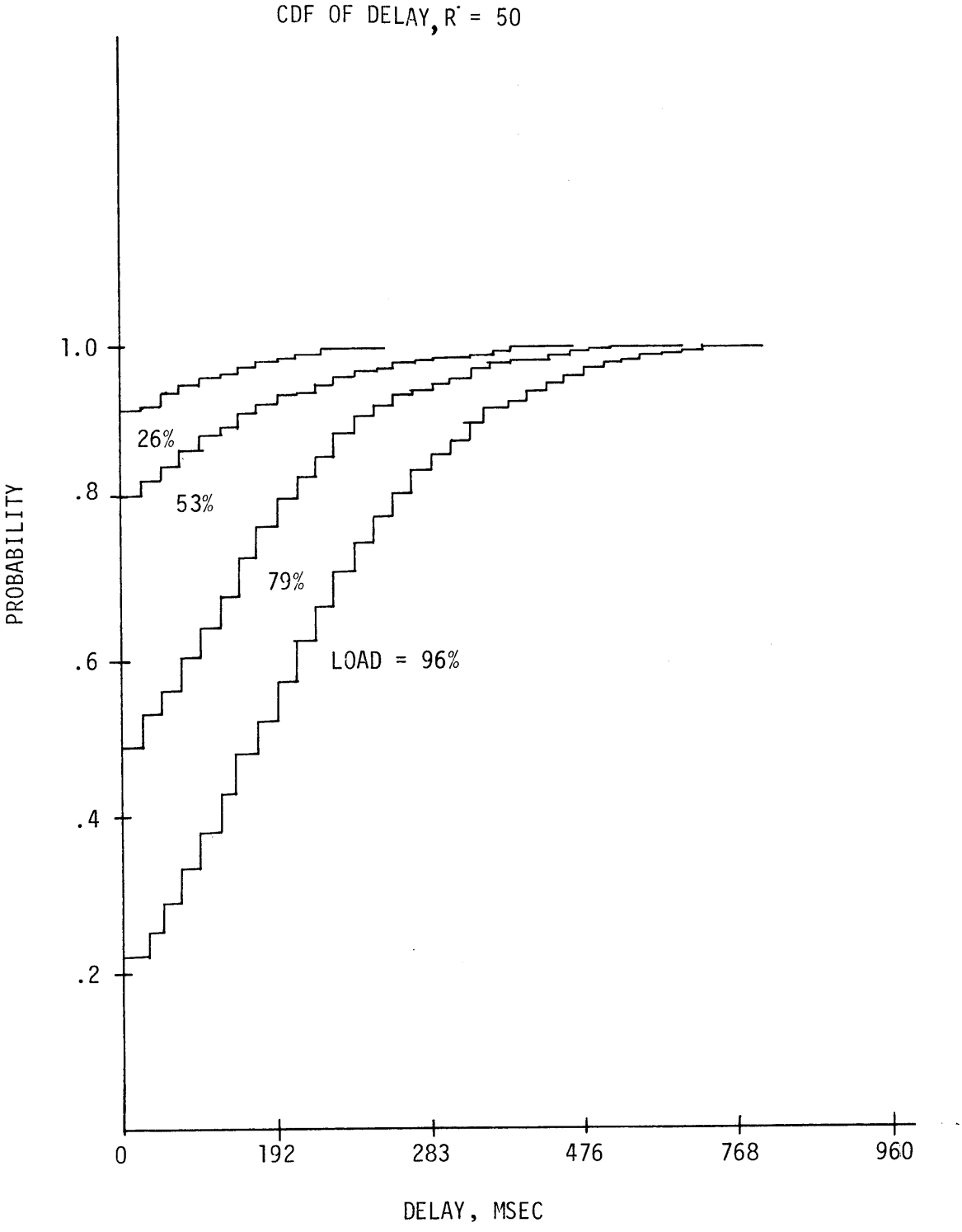

CDF OF DELAY, R = 50
PROBABILITY
1.0
.8
.6
.4
.2
26%
53%
79%
LOAD = 96%
0
192
283
476
768
960
DELAY, MSEC

Figure 6.12B.

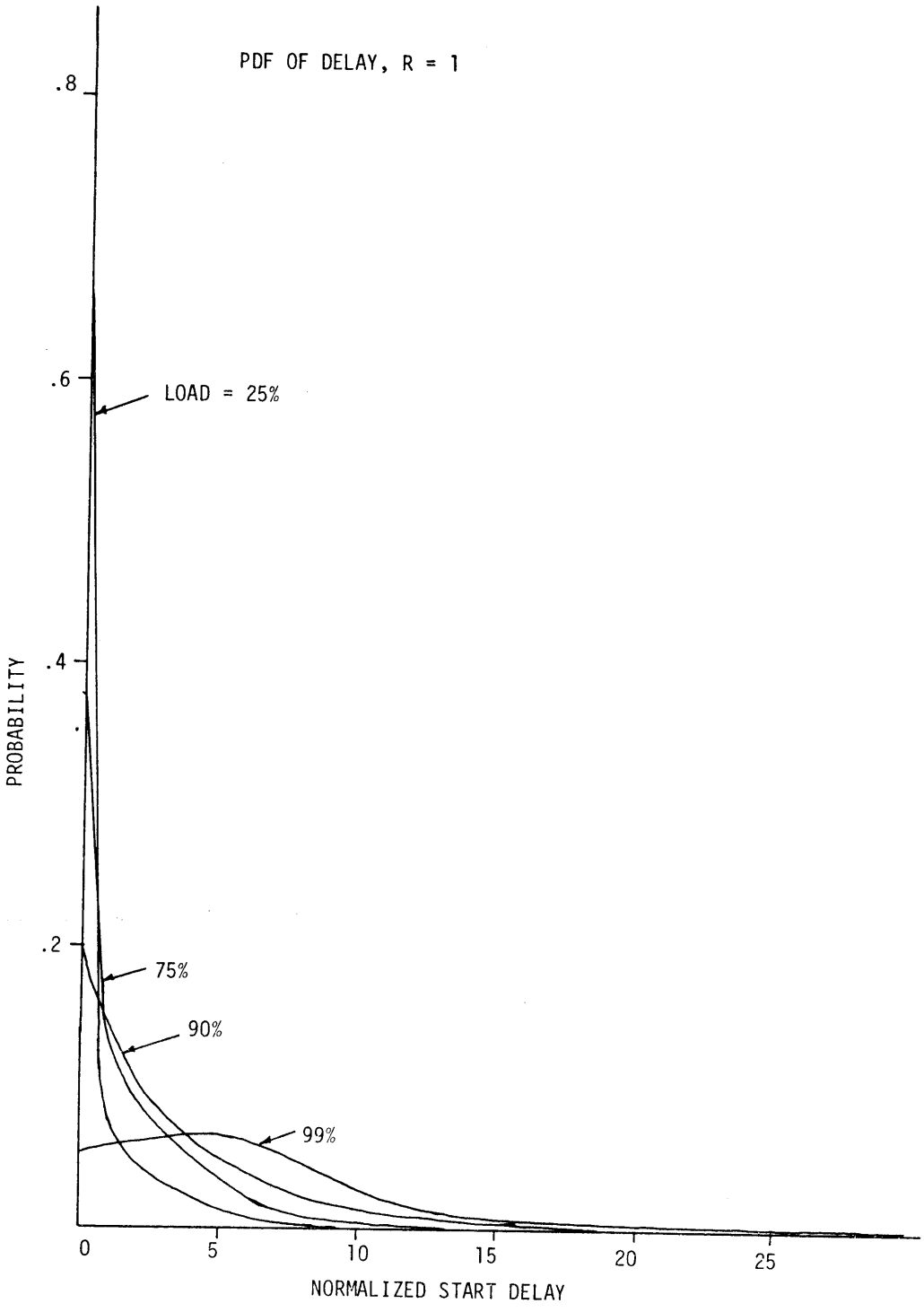

PDF OF DELAY, R = 1
.8
.6
.4
.2
PROBABILITY
LOAD = 25%
75%
90%
99%
0
5
10
15
20
25
NORMALIZED START DELAY

Figure 6.13.

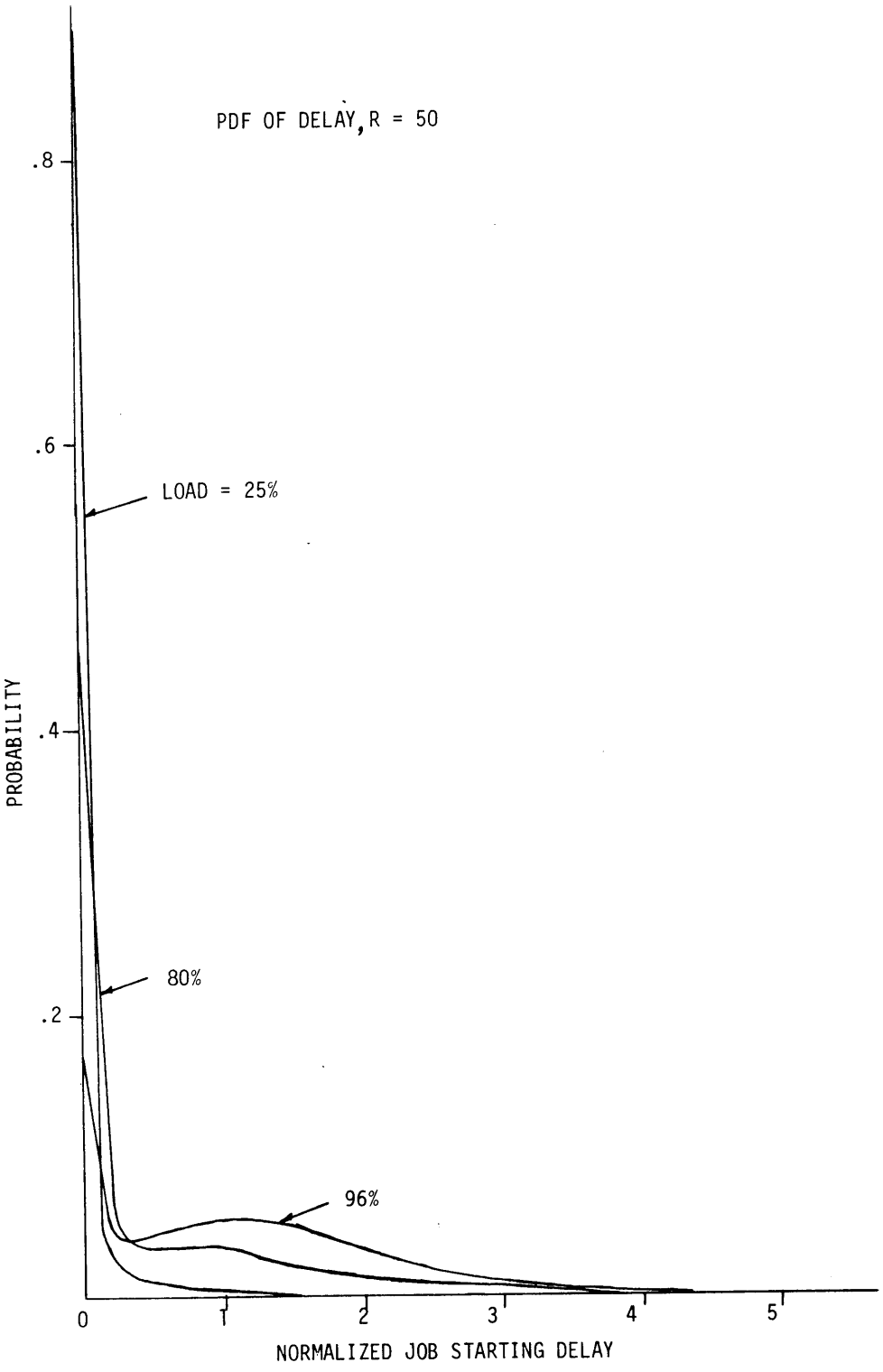

PDF OF DELAY, R = 50
.8
.6
.4
.2
PROBABILITY
LOAD = 25%
80%
96%
0
1
2
3
4
5
NORMALIZED JOB STARTING DELAY

Figure 6.14.

in fact, increases slightly before gradually decreasing with a long tail. A monotonically decreasing function implies that the probability of a job starting without a delay is higher than with any other value of delay. The distribution function for R=50 is slightly more complex as shown in Figure 6.14. The high load curves have three distinct phases; a rapid initial decline followed by a plateau and then a final asymptotic descent.

Let us turn our attention now to results regarding maximum job starting delay. In principle, the probability of any delay, no matter how large, occurring in practice is non-zero. This was somewhat verified in the experiments by incresing the run time in one case to more than an hour. The maximum delay continued to creep upward although at an increasingly slower rate. The theoretical maximum delay, therefore, is always infinite. However, it is instructive to take a look at various percentiles of delay. Figure 6.15, for example, shows for R=1 the 80 percentile delay as a function of the system load. That is, 80 per cent of all delays fall below this function. The 80 percentile delay doubles, for example, as the load is increased from 50 to 75 per cent. The 99 percentile curve, however, has a slightly different nature. It is seen that the same increase in the load factor results only in a 50 per cent increase in the 99 percetile delay. This has some interesting consequences. For example, the penalty to be paid in going from a low to a high load factor operating point is not as great as the mean delay function implies. Figure 6.15 also shows points corresponding to the 99.99 percentile curve. The scattering of the points is due to the small number of samples obtained. At any rate, the function tends to show a linear behavior in the 5 to 95% load range. In other words, cost incurred in terms of worst case delays increases only linearly with the system load. This has also been found to be true for the higher values of R.

Figure 6.16 shows the probability of the job starting delay exceeding a given value as a function of the system load for R=1. It is seen that this probability approaches a linear function for zero delay. That is, the per cent of jobs that start on time decreases linearly with the load. This figure

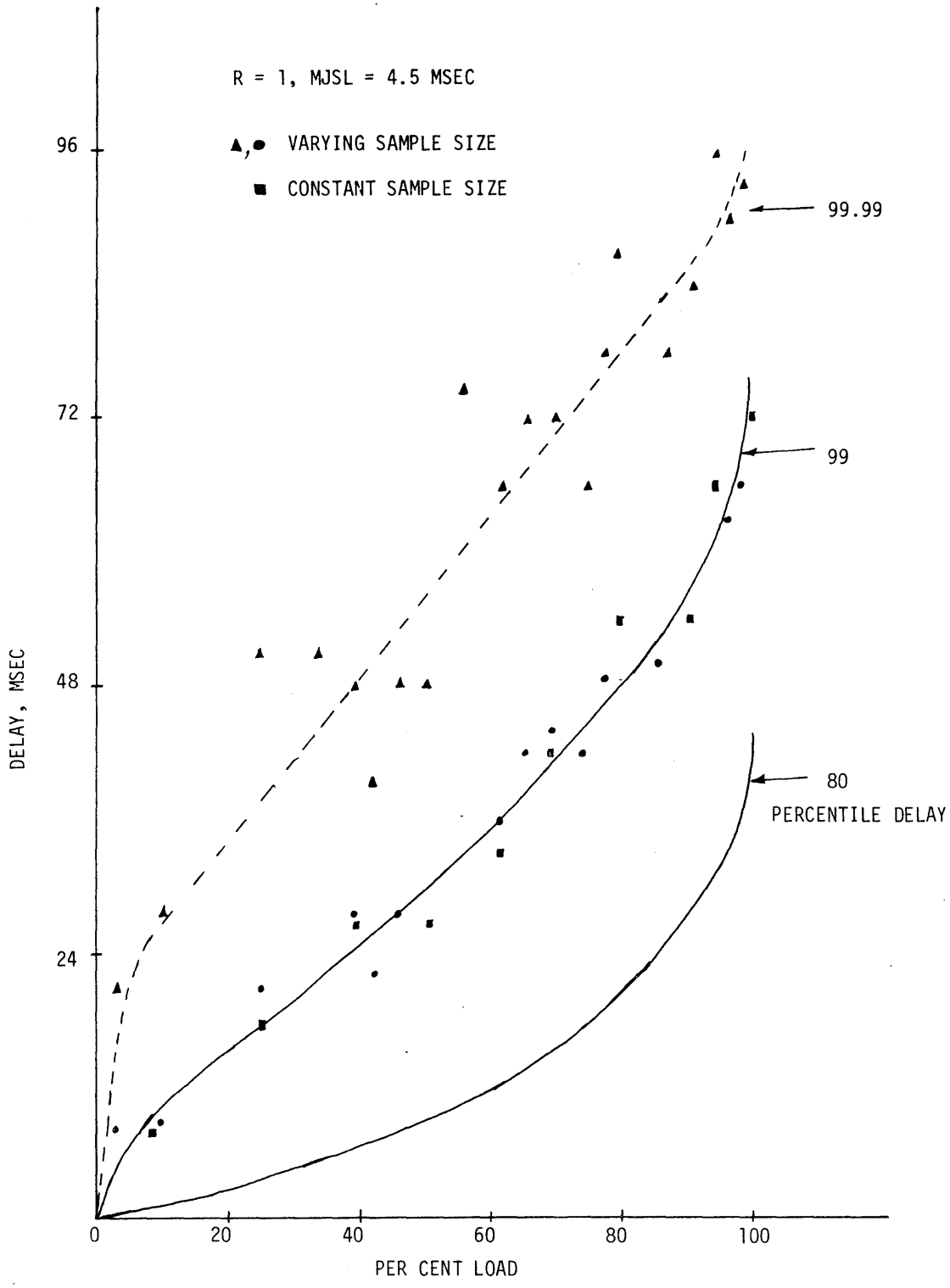

R = 1, MJSL = 4.5 MSEC
▲,● VARYING SAMPLE SIZE
■ CONSTANT SAMPLE SIZE
99.99
99
80
PERCENTILE DELAY
DELAY, MSEC
96
72
48
24
0
20
40
60
80
100
PER CENT LOAD

Figure 6.15.

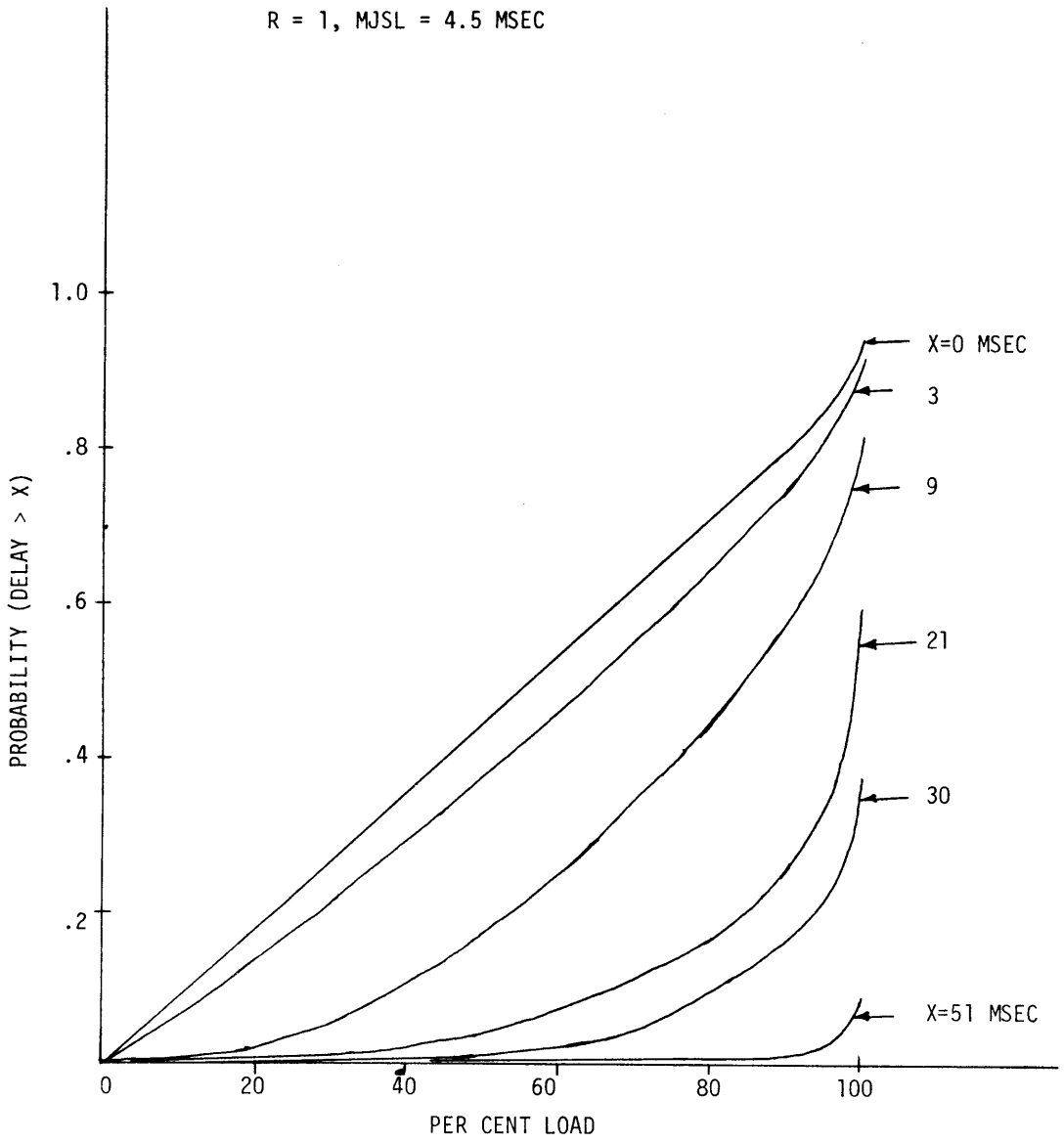

R = 1, MJSL = 4.5 MSEC
PROBABILITY (DELAY > X)
1.0
.8
.6
.4
.2
0
20
40
60
80
100
PER CENT LOAD
X=0 MSEC
3
9
21
30
X=51 MSEC

Figure 6.16.

indicates that large system loads may be quite acceptable in real time systems if moderate delays are permitted. Notice, for example, that if a 51 msec starting delay is permissible, then a system load of 85% will produce delays in excess of 51 msec only 1% of the time. Zero-delay curves for other values of R are shown in Figure 6.17.

6.4 FCFS Versus SJF

It was pointed out in Chapter V with the help of an example that for a multiprocessor the Shortest-Job-First job dispatch strategy will reduce the mean delay slightly, while widening the spread of the delay and increasing the Executive overheads. The results shown in Figs 6.18 through 6.22 tend to confirm this.

Figure 6.18 compares the normalized mean job starting delay for R=1 for the two strategies as a function of the system load while Fig. 6.19 shows the probability distribution function for fifty per cent load for the same value of R. It is seen that the delay for some jobs is reduced, thereby increasing the probability of short delays. However this is done at the expense of the longer jobs, thereby increasing the probability of longer delays. The range of delays is increased considerably while the mean delay is reduced only slightly. Also for R=1 the processor time devoted to the Executive is up to two per cent more (depending on the load factor) for SJF than for FCFS.

The same results for R=10 are shown in Figs. 6.20 and 6.21. Although the PDF's for the two strategies are very close together the SJF curve begins at a higher probability value and extends to much higher delays. The average delay values are also approximately equal for the two scheduling schemes. Increasing the variance of the job lengths does not impact this significantly, as evidenced by Figs. 6.22 and 6.23.

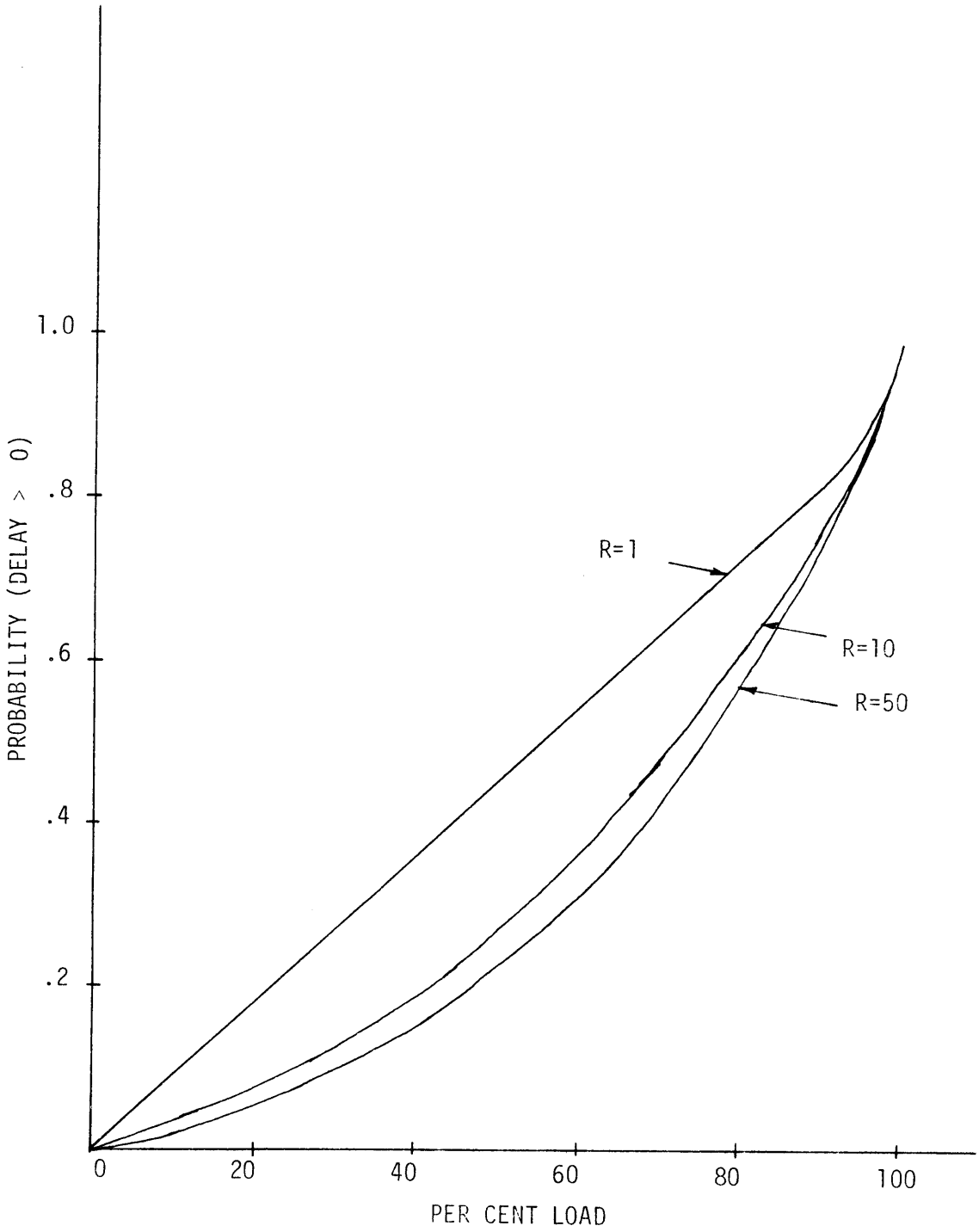

1.0
.8
.6
.4
.2
PROBABILITY (DELAY > 0)
R=1
R=10
R=50
0
20
40
60
80
100
PER CENT LOAD

Figure 6.17.

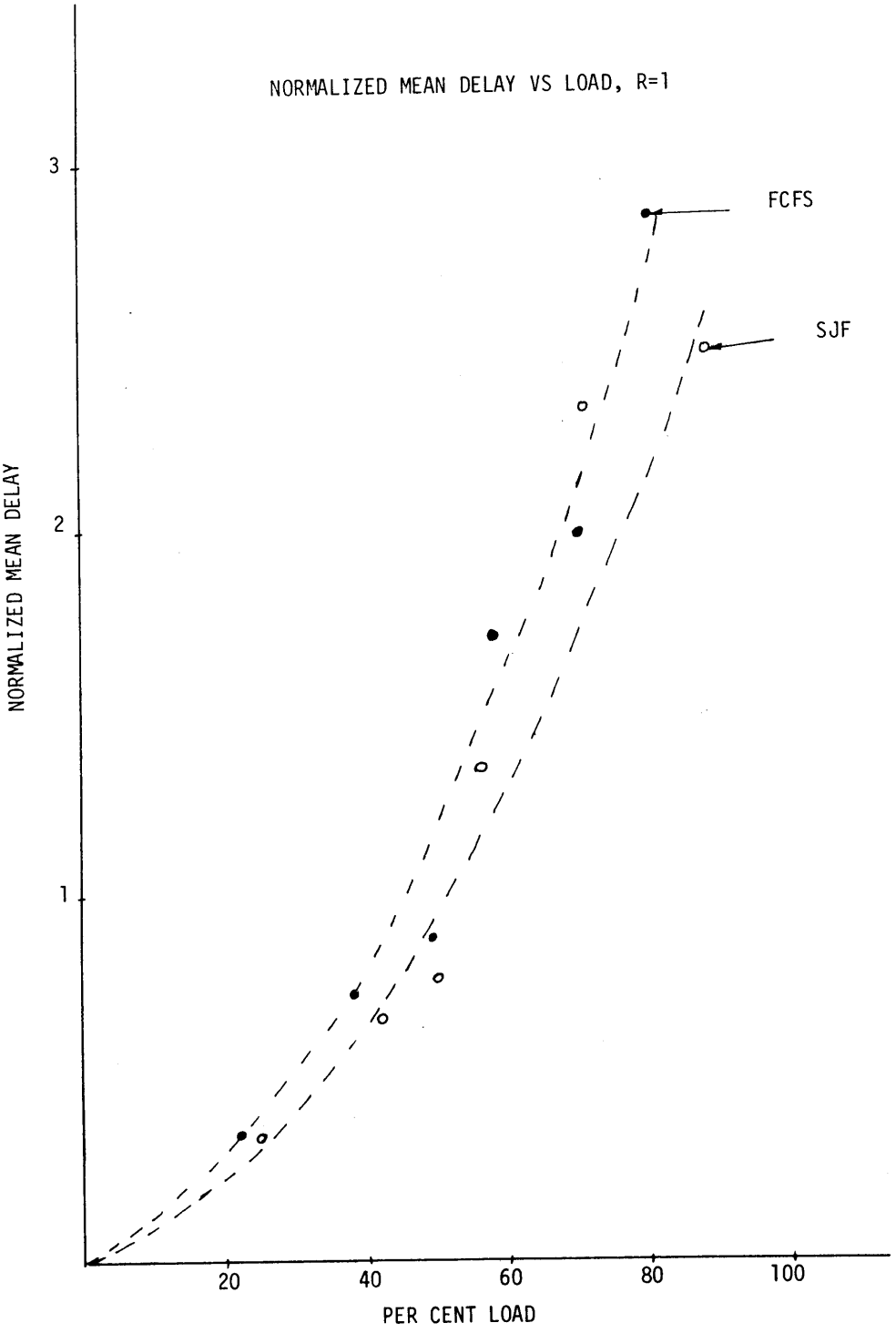

NORMALIZED MEAN DELAY VS LOAD, R=1
NORMALIZED MEAN DELAY
3
2
1
FCFS
SJF
20
40
60
80
100
PER CENT LOAD

Figure 6.18

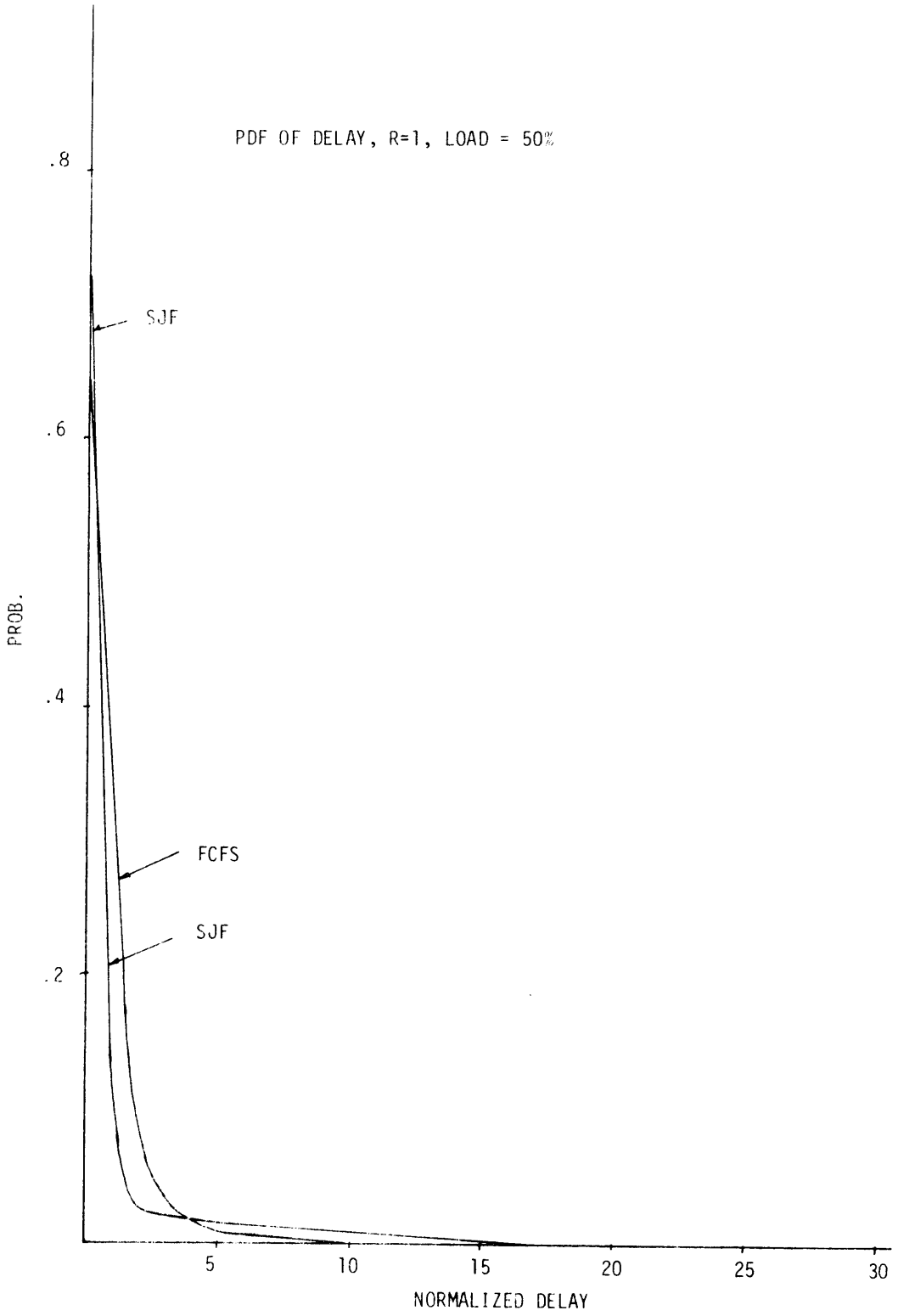

PDF OF DELAY, R=1, LOAD = 50%
.8
.6
.4
.2
PROB.
SJF
FCFS
SJF
5
10
15
20
25
30
NORMALIZED DELAY

Figure 6.19

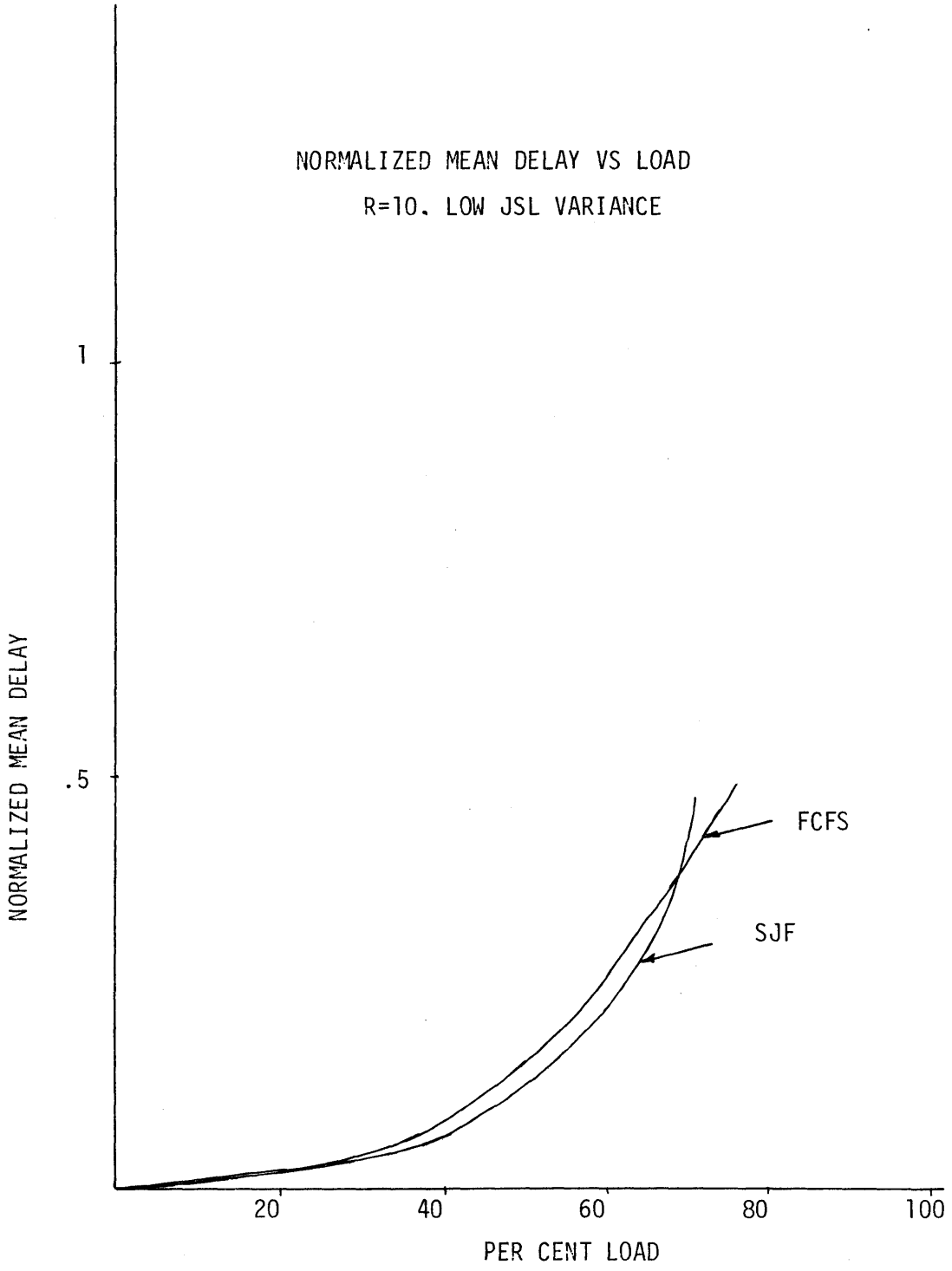

NORMALIZED MEAN DELAY VS LOAD
R=10. LOW JSL VARIANCE
NORMALIZED MEAN DELAY
1
.5
FCFS
SJF
20
40
60
80
100
PER CENT LOAD

Figure 6.20

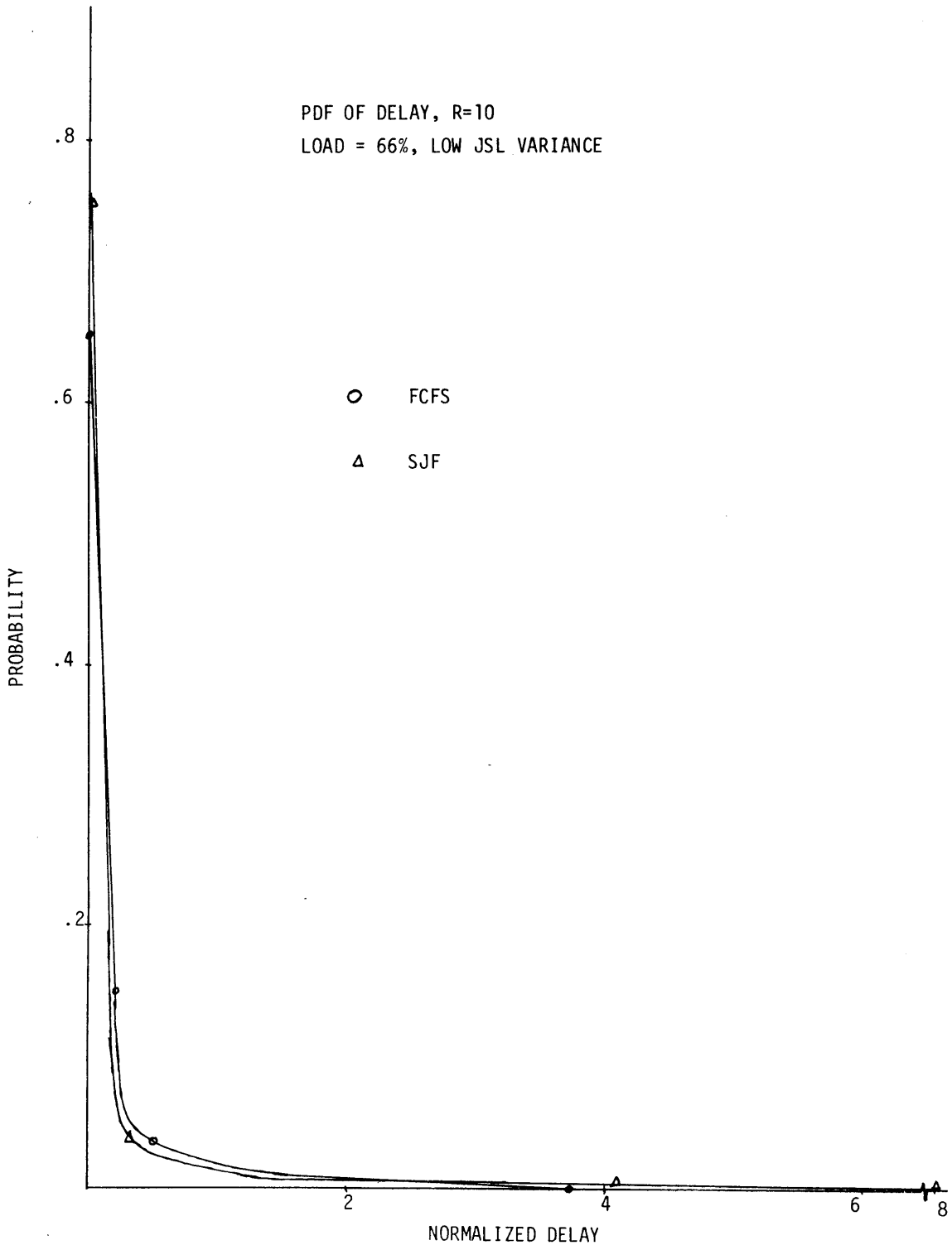

PDF OF DELAY, R=10
LOAD = 66%, LOW JSL VARIANCE
FCFS
SJF
PROBABILITY
.8
.6
.4
.2
2
4
6
8
NORMALIZED DELAY

Figure 6.21

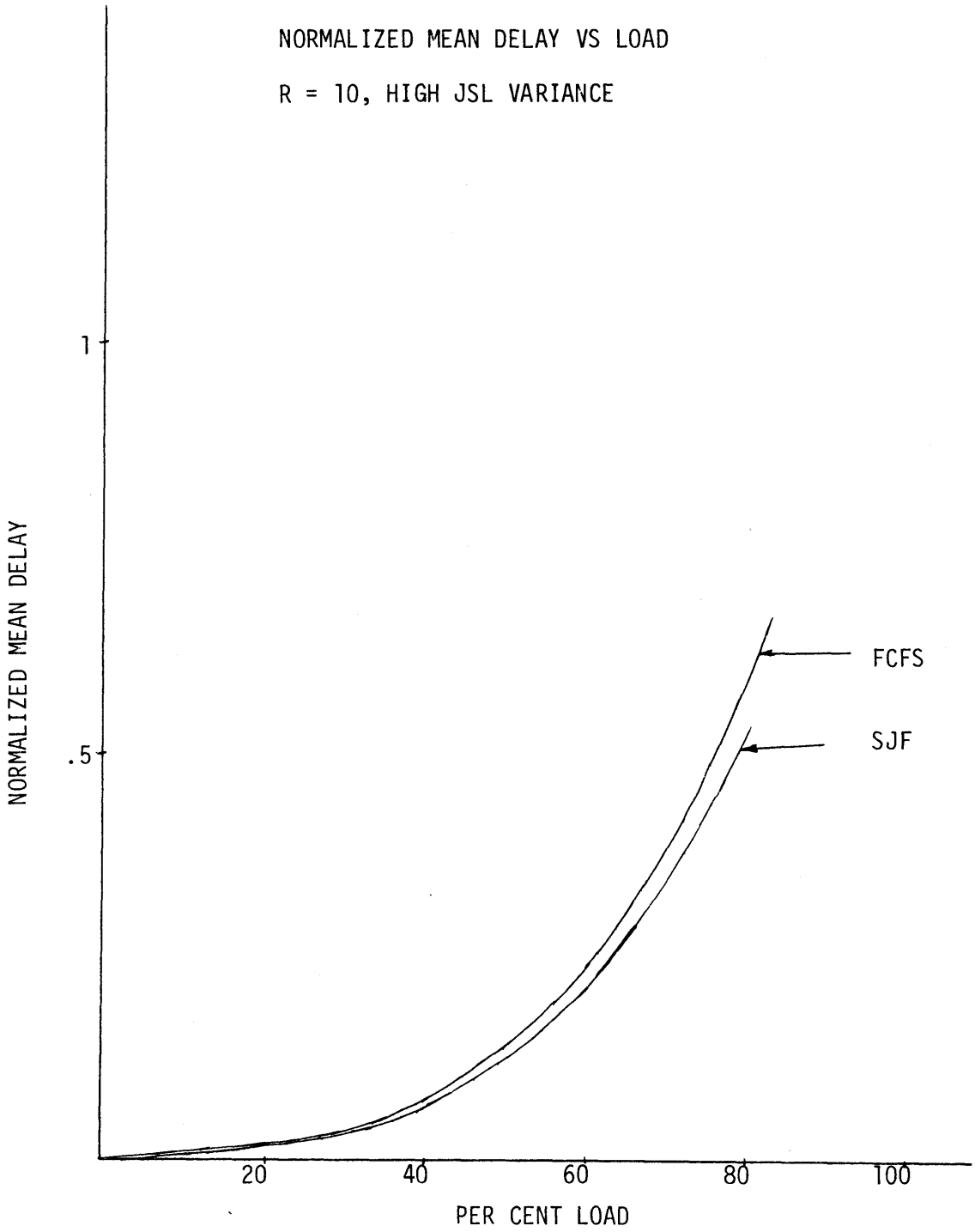

NORMALIZED MEAN DELAY VS LOAD
R = 10, HIGH JSL VARIANCE
NORMALIZED MEAN DELAY
1
.5
FCFS
SJF
20
40
60
80
100
PER CENT LOAD

Figure 6.22

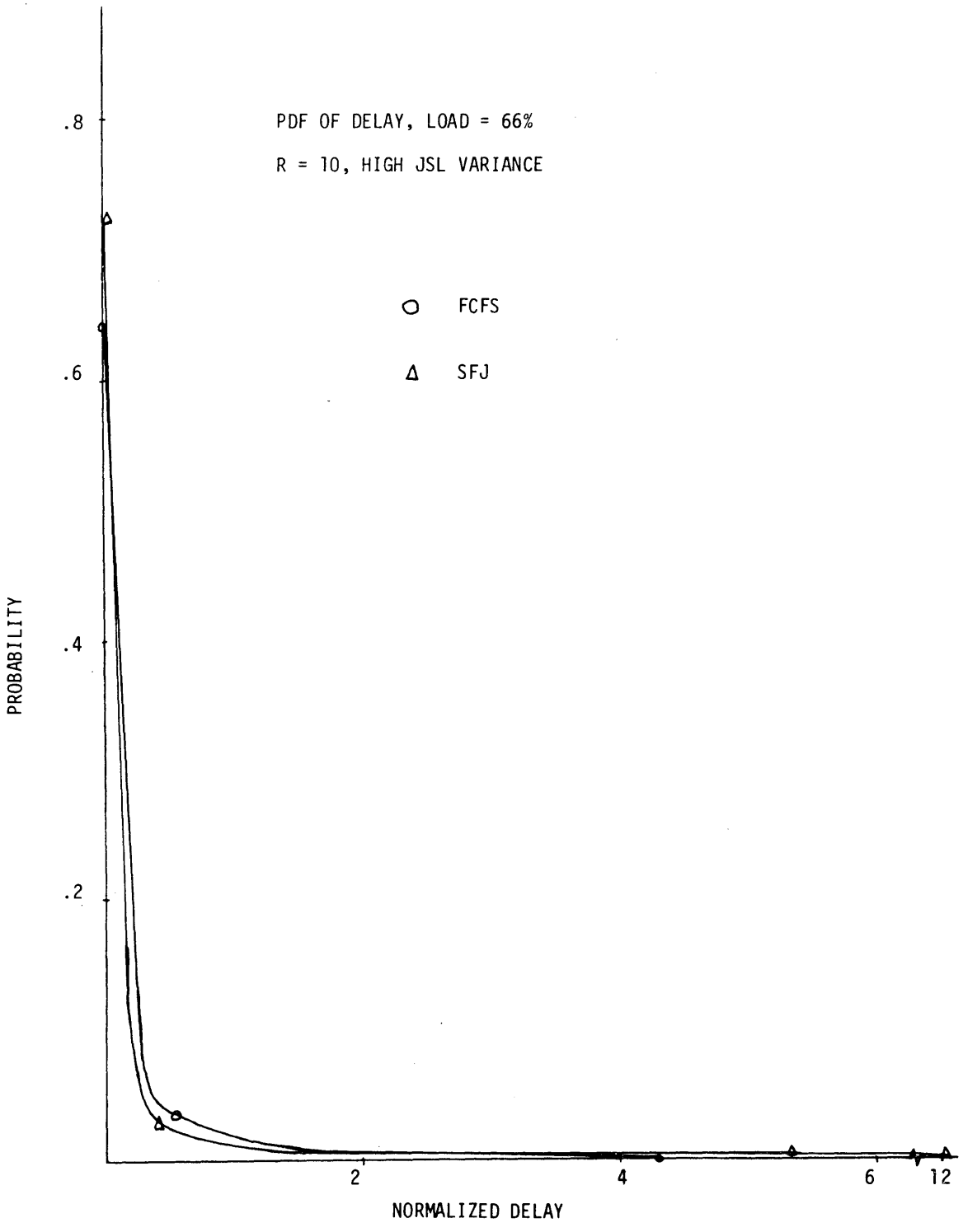

PDF OF DELAY, LOAD = 66%
R = 10, HIGH JSL VARIANCE
FCFS
SFJ
.8
.6
.4
.2
PROBABILITY
2
4
6
12
NORMALIZED DELAY

Figure 6.23

These results show that more jobs start on schedule with SJF, but at the expense of other jobs which are delayed an inordinately long time. Any gains in the average delay are likely to be nullified for a system with a higher number of processors.

6.5 Summary

This chapter has decribed the simulation results concerning the system utilization and the job start delay statistics. There are two points that do not directly concern the results but are relevant to the experimental procedure and therefore should be mentioned here briefly. First of all it was found that such measurements as the per cent processor time devoted to different functions assumed their steady state values within several tens of seconds of run time. Therefore it would be unnecessary to run the simulation any longer if only those variables were being measured. However the mean delay took up to several hundreds of seconds to stabilize. In this respect the delay statistics were also sensitive to how the waitlist was initialized. Therefore a different initial waitlist was used for each value of R. Finally, if the run time is held constant then the sample size or the number of jobs completed during a single run will be determined by the job dispatch rate, that is, the load factor. To investigate the impact of varying sample size on the results one set of experiments was run with a constant sample size. This was done by varying the run time with the load factor. The result is shown in Figure 6.15. In this figure the squares correspond to the 99 percentile delay curve when the sample size is held constant. It is seen that the circles which represent the 99 percentile curve when the run time is held constant closely approximates the fixed sample size curve.

Once again to summarize the results of this chapter, interesting characteristics were discovered concerning the frequency distribution of job starting delay as well as the maximum job starting delay. Also two

job dispaching strategies, First-Come-First-Serve and Shortest-Job-First, were compared from the performance viewpoint. It was found that the SJF offers little advantage over FCFS while increasing the maximum delays sharply.

CHAPTER VII

THEORETICAL MODELS AND RESULTS

7.1 Introduction

The performance evaluation experiments conducted on the CERBERUS multiprocessor differ in one important respect from the usual multiprocessor simulation studies in that rather than simulating multiprocessor conditions on a uniprocessor they were actually conducted on a multiprocessor. As such the experiments are more accurately described as a combination of simulation, hardware monitoring and software monitoring. Nonetheless, as in any other simulation study, it is necessary to validate the present experimental results.

Two different approaches were considered for the validation problem: a queueing network model and a Markov process model. It was found easier to incorporate variables such as system load and performance criteria such as job start delay into the Markov process models. The Markov process approach was therefore chosen to obtain system utilization and job start delay statistics.

First of all a simple birth-and-death Markov process was used to model a situation in which the executive length is shortened to such a point that it becomes negligibly small. In such a case the ratio of mean job length to the executive length tends to infinity, i.e., $R=\infty$. This simple model was used to obtain theoretical results concerning the mean delay and the probability of starting a job without any delay as a function of the system

load. The model was then enlarged and refined to take into account the fact that the Executive takes a non-zero time to run. Results depicting the impact of the executive run time and the ratio R were derived from this model.

7.2 A Simple Analytical Model

For the case when $R=\infty$, it is easy to derive an expression for the steady state probabilities for a system with m processors, and then specialize the results to the three-processor case. Consider therefore a queue with m independent servers (processors). The job arrivals (customers in the queue) are assumed to follow a Poisson process with a mean arrival rate of λ jobs per unit time. The service time (running time of jobs) is assumed to be an exponentially distributed random variable with a mean value of $\frac{1}{\mu}$. The state transition diagram of such a system is shown in Figure 7.1.

There are n customers in the system, including the ones being served, in state S_n. Let the steady state probability of the system being in state S_n be P_n. In the steady state, then, the rate of arrival into state S_n equals the rate of departure from that state. Therefore,

$$
\begin{aligned}
P_o\lambda &= P_1\mu \\
P_o\lambda + 2\mu P_2 &= P_1(\lambda+\mu) \\
P_1\lambda + 3\mu P_3 &= P_2(\lambda+2\mu) \\
P_{m-2}\lambda + m\mu P_m &= P_{m-1}\left[\lambda + (m-1)\mu\right] \\
P_{m-1}\lambda + m\mu P_{m+1} &= P_m\left[\lambda + m\mu\right] \\
P_{n-1}\lambda + m\mu P_{n+1} &= P_n\left[\lambda + m\mu\right]
\end{aligned}
\qquad (7.1)
$$

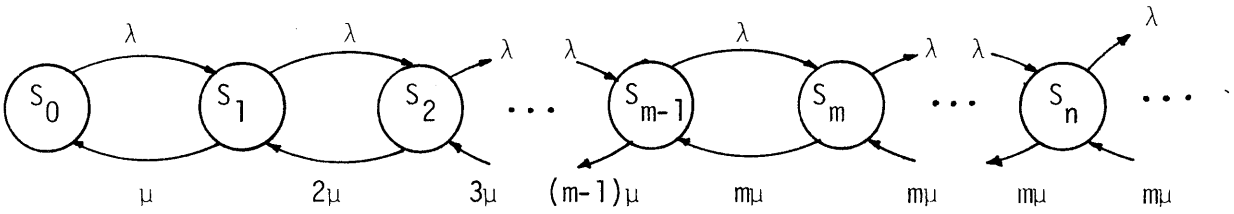


Figure 7.1.

In addition, $\sum_{i=0}^{\infty} P_i = 1$ (7.2)

Solving these equations for the steady state probabilities [41], we obtain

$$P_n = T^{-1} \frac{(m\rho)^n}{n!} \qquad n < m$$

$$= T^{-1} \frac{m^m}{m!} \rho^n \qquad n \geq m$$

$$\text{where} \quad T = 1 + m\rho + \frac{(m\rho)^2}{2!} + \ldots + \frac{(m\rho)^{m-1}}{(m-1)!} + \frac{(m\rho)^m}{m!(1-\rho)}$$

$$\text{and} \quad \rho = \frac{\lambda}{m\mu} \qquad (7.3A)$$

The parameter ρ is the system load, defined as the ratio of the mean job arrival rate, λ, to the mean job completion rate, $m\mu$, of the system. Substituting the value three for m in Equation 7.3A we obtain the following expression for the steady state probabilities of the three processor system.

$$P_n = T_3^{-1} \frac{(3\rho)^m}{n!} \qquad n < 3$$

$$= T_3^{-1} \frac{9\rho^n}{2} \qquad n \geq 3$$

$$\text{where} \quad T_3 = \frac{2+4\rho+3\rho^2}{2(1-\rho)} \qquad (7.3B)$$

The probability of a job starting without a delay is the probability of finding at least one free processor when the job arrives. This is given by $P_0 + P_1 + P_2$.

i.e.,
$$\text{Prob. (delay = 0)} = 1 - \frac{9\rho^3}{2+4\rho+3\rho 2}$$

$$\text{Prob. (delay > 0)} = \frac{9\rho^3}{2+4\rho+3\rho^2} \qquad (7.4)$$

This probability function is shown in Figure 7.2 as a function of the system load.

Now, in order to obtain the mean delay we must first obtain an expression for the distribution function of the delay. The probability that there are n jobs in the system when a new job arrives is P_n. That is, the probability that the new job upon arrival finds the system in state S_n is P_n. The waiting time of a new job when there are two or fewer customers in the system equals zero. When there are n customers (n>2) in the system the waiting time of the new job equals the sum of the service times of n-2 jobs in the queue. Now, the service time of each job is an exponentially distributed random variable with a mean value of $1/\mu$ time units. The probability distribution function (pdf) of the service time equals $\mu e^{-\mu t}$. To obtain the total waiting time or delay we must sum n-2 such random variables. The sum may be obtained by convolving their pdf's in the time domain or equivalently multiplying their Laplace transforms in the frequency domain [42]. Since there are three independent servers in the system, the pdf of the service time becomes $3\mu e^{-3\mu t}$. Its Laplace transform is

$R = \infty$

$$\frac{9\rho^3}{2 + 4\rho + 3\rho^2}$$

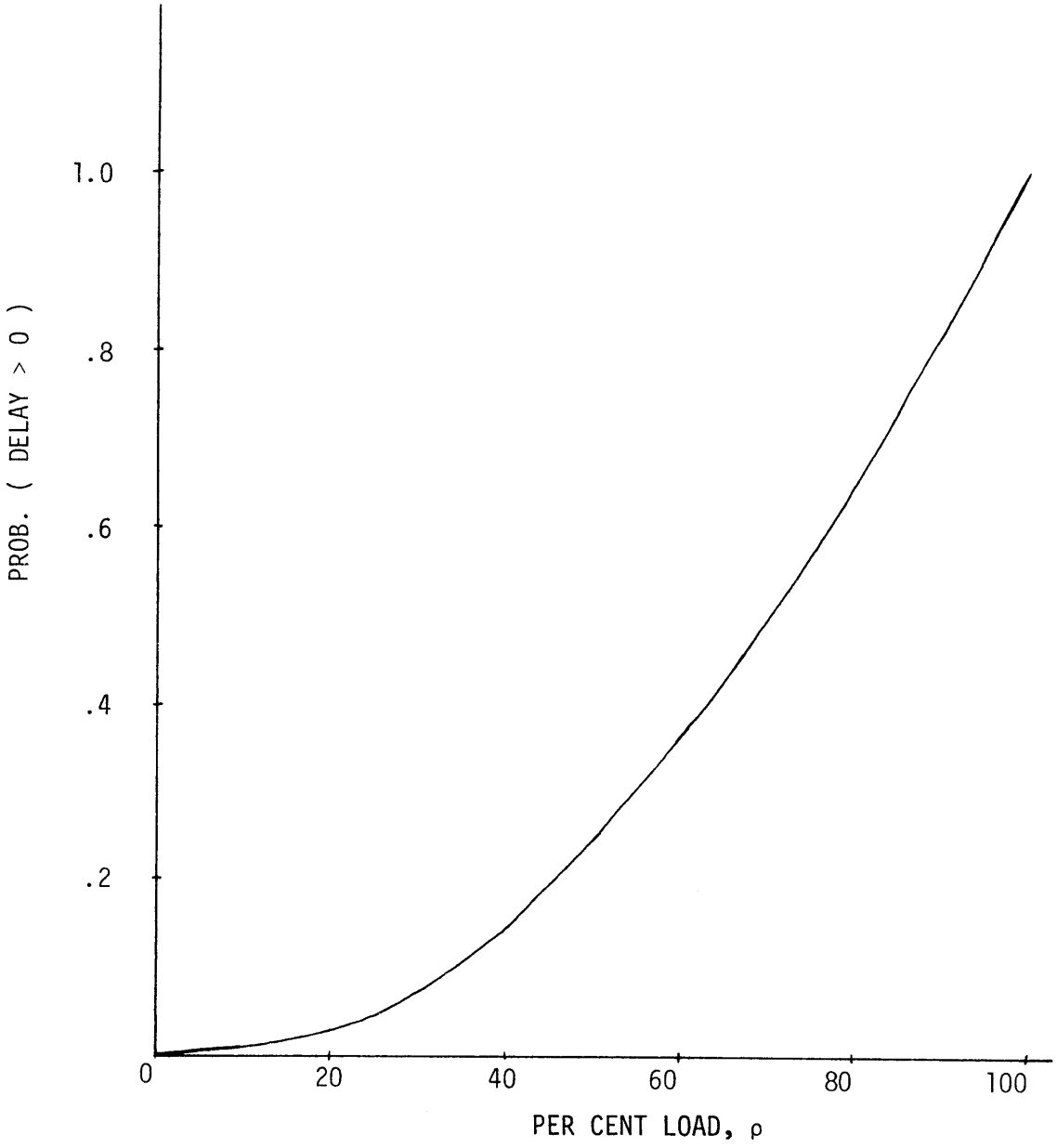


Figure 7.2

$\frac{3\mu}{3\mu+s}$. The product of n-2 such terms is $\left(\frac{3\mu}{3\mu+s}\right)^{n-2}$. Therefore the Laplace transform of the delay equals

$$\sum_{n=3}^{\infty} \left(\frac{3\mu}{3\mu+s}\right)^{n-2} P_n.$$

With Equations 7.3,

$$\mathcal{L}\ (\text{PDF OF DELAY}) = \sum_{n=3}^{\infty} \frac{9\rho^n}{2} \left(\frac{3\mu}{3\mu+s}\right)^{n-2} S^{-1} \qquad (7.5)$$

$$\text{or} \qquad \text{PDF} = \sum_{n=3}^{\infty} \frac{9\rho^n}{2} (3\mu)^{n-2} \frac{t^{n-3}}{(n-3)!} e^{-3\mu t} T_3^{-1}$$

$$= \frac{9\rho^3}{2+4\rho+3\rho^2} 3\mu(1-\rho) e^{-3\mu(1-\rho)t} \qquad (7.6)$$

$$\text{Therefore, mean delay} = \frac{3\rho^3}{(2+4\rho+3\rho^2)(1-\rho)} \text{ mean job length} \qquad (7.7)$$

The normalized mean delay is shown in Figure 7.3 as a function of the system load.

Equation 7.6 is an exact analytical expression for the pdf of the job starting delay for a system modeled by Figure 7.4. The method usedto obtain such an expression for more complex models is outlined below.

1. Assume that the new job upon becoming due to run finds the system in state S_n. This state will be denoted by $S_{initial}$ or S_i. The system state when the new job actually gets started will be denoted by S_{final} or S_f. In

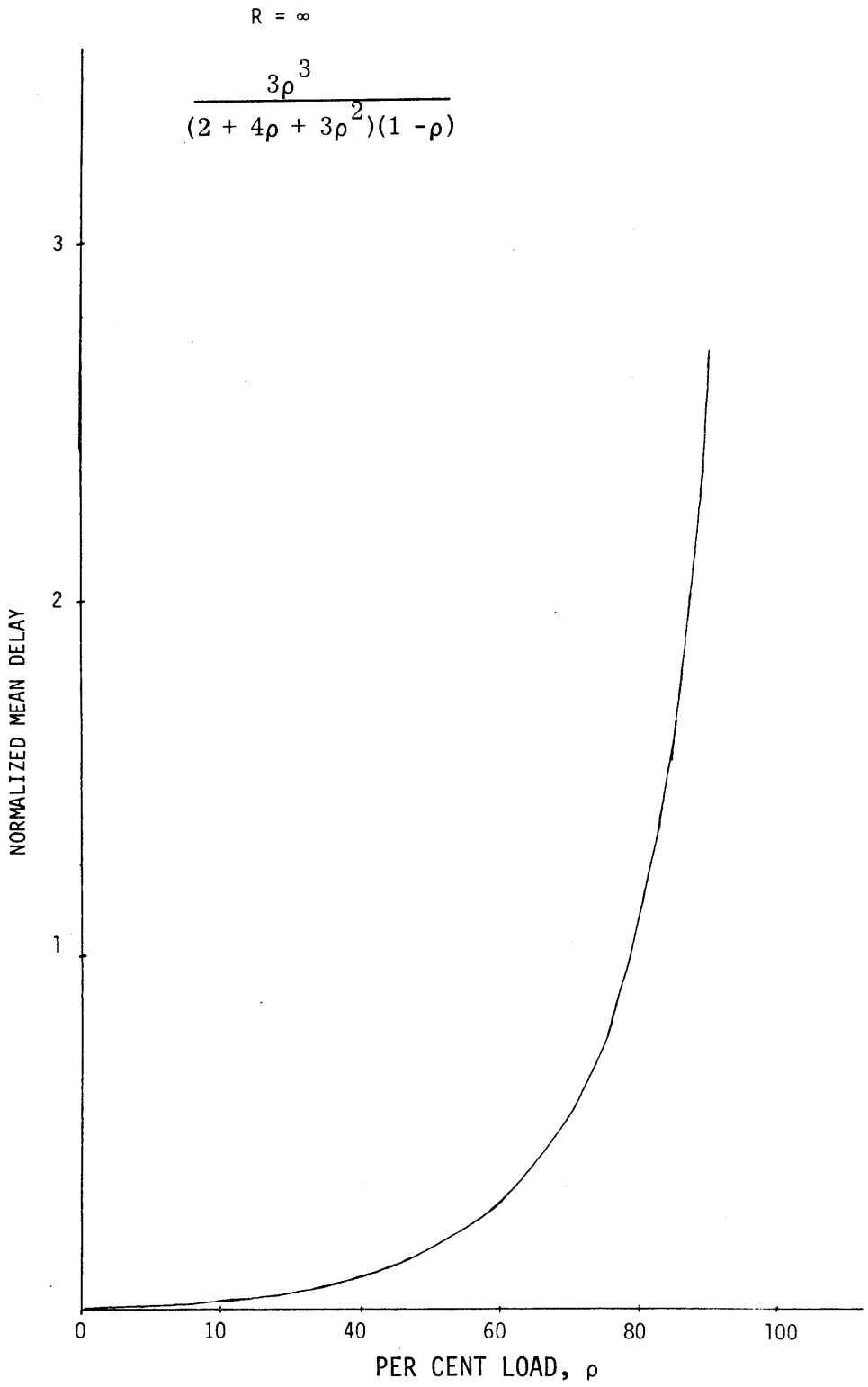

R = ∞
$\frac{3\rho^3}{(2 + 4\rho + 3\rho^2)(1 - \rho)}$
NORMALIZED MEAN DELAY
3
2
1
0
10
40
60
80
100
PER CENT LOAD, ρ

Figure 7.3

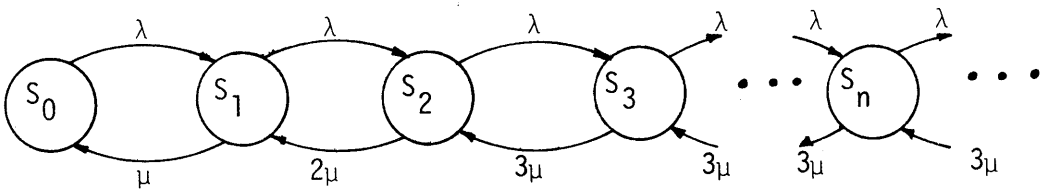


Figure 7.4

Figure 7.4 S_f is S_2. In more complex systems there may be more than one final state. It is assumed here that the system moves from S_i to S_f along a certain path as the successive jobs in the queue are completed. That is, any state transitions due to further job arrivals are ignored. The first step is to identify all such paths from S_i to S_f.

2. The second step is to obtain the pdf of each state transition in the S_i-S_f path. Once again in the present example the pdf is identical for all the transitions.

3. Next, obtain the Laplace transform of each pdf.

4. The fourth step is to obtain the product of all the Laplace transforms in the S_i-S_f path and weigh it with the probability factor P_n, the probability of the system being in state S_i. The resulting expression is one term of an infinite series. In practice the series would be finite due to the finite number of states in the truncated Markov model.

Each initial state contributes one such term to the series. If, however, there are multiple paths between S_i and S_f, say m, then there would be m terms corresponding to that initial state in the series. Figure 7.5 illustrates this point. Figure 7.5 shows part of a system with an initial and a final state and two different paths between the two states. The pdf of the rate of transition from state S_i into S_1 is exponential and is given by ae^{-at}. Its Laplace transform is $\frac{a}{s+a}$. Other transitions and their rates are also shown in the figure. In this example the system would go from S_i to S_f via S_1 with the probability $\frac{a}{a+b}$ and via S_2 with the probability $\frac{b}{a+b}$. Therefore the two terms corresponding to S_i would be $\frac{a}{a+b}\left(\frac{a}{s+a}\right)^2 P_i$ and $\frac{b}{a+b}\left(\frac{b}{s+b}\right)^2 P_i$.

5. The sum of the series gives the Laplace transform of the pdf of the delay.

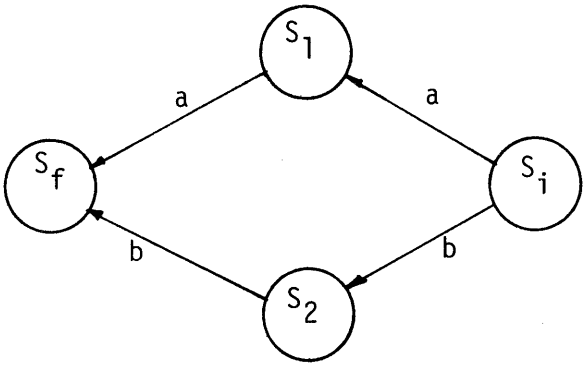


Figure 7.5

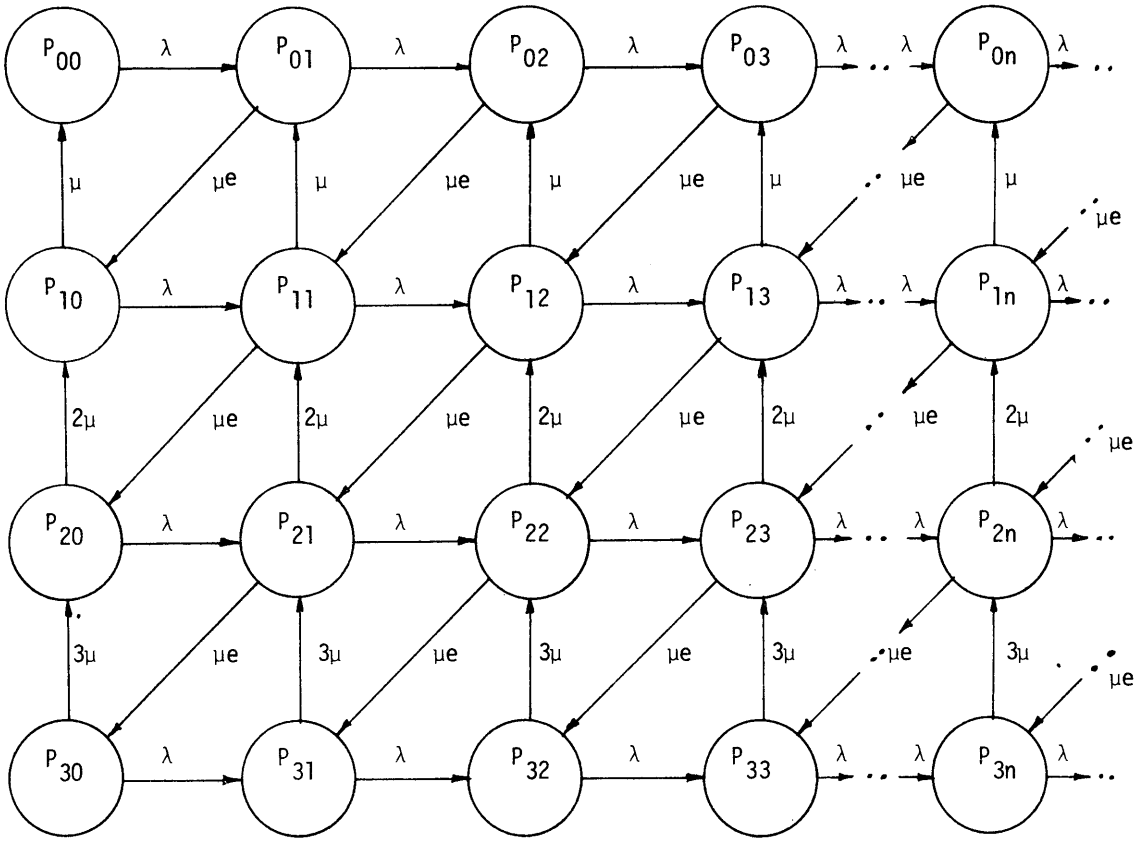
6. The last step is to invert the Laplace transform. Steps 5 and 6 may of course be interchanged. That is, the series may be summed after inverting the Laplace transforms.

7.3 Exponential Distribution Markov Model

Figures 7.2 and 7.3 are the theoretical curves for $R=\infty$. In order to incorporate the parameter R into the model and to see the impact of the executive length on the system performance, the state transition diagram of Figure 7.4 was expanded to the diagrams shown in Figures 7.6A and 7.6B. Here $P_{m,n}$ is the probability of finding the system in state $S_{m,n}$ with m jobs running and n jobs waiting. Before beginning a job a processor that is free must run the Executive. If another processor is already running the Executive then it must wait until the executive program becomes available. This gives rise to new states as shown in Figure 7.6. The run times of the Executive and the job steps are assumed to be exponentially distributed random variables with mean values of $\frac{1}{\mu_e}$ and $\frac{1}{\mu}$ respectively. Hence the name 'Exponential Distribution Markov Model'.

The steady state equations for the transition to and from each state are listed as Equations 7.8.

If we approximate the infinite state birth-and-death Markov process of Figure 7.6 by a finite state process with N states, we would then have N+1 equations in N unknowns. Thus one of the equations is redundant. In other words one of the equations is a linear combination of others. As a matter of fact, each of the state transition equations is simply a sum of the rest of the equations. Therefore by dropping any one of the state transition equations we obtain a set of N linearly independent equations in N unknowns, that is, the steady state probabilities. A standard Gauss-Jordan reduction was used to obtain the numerical solution of this set on an IBM 360 computer. The equations were solved for a range of load factors for

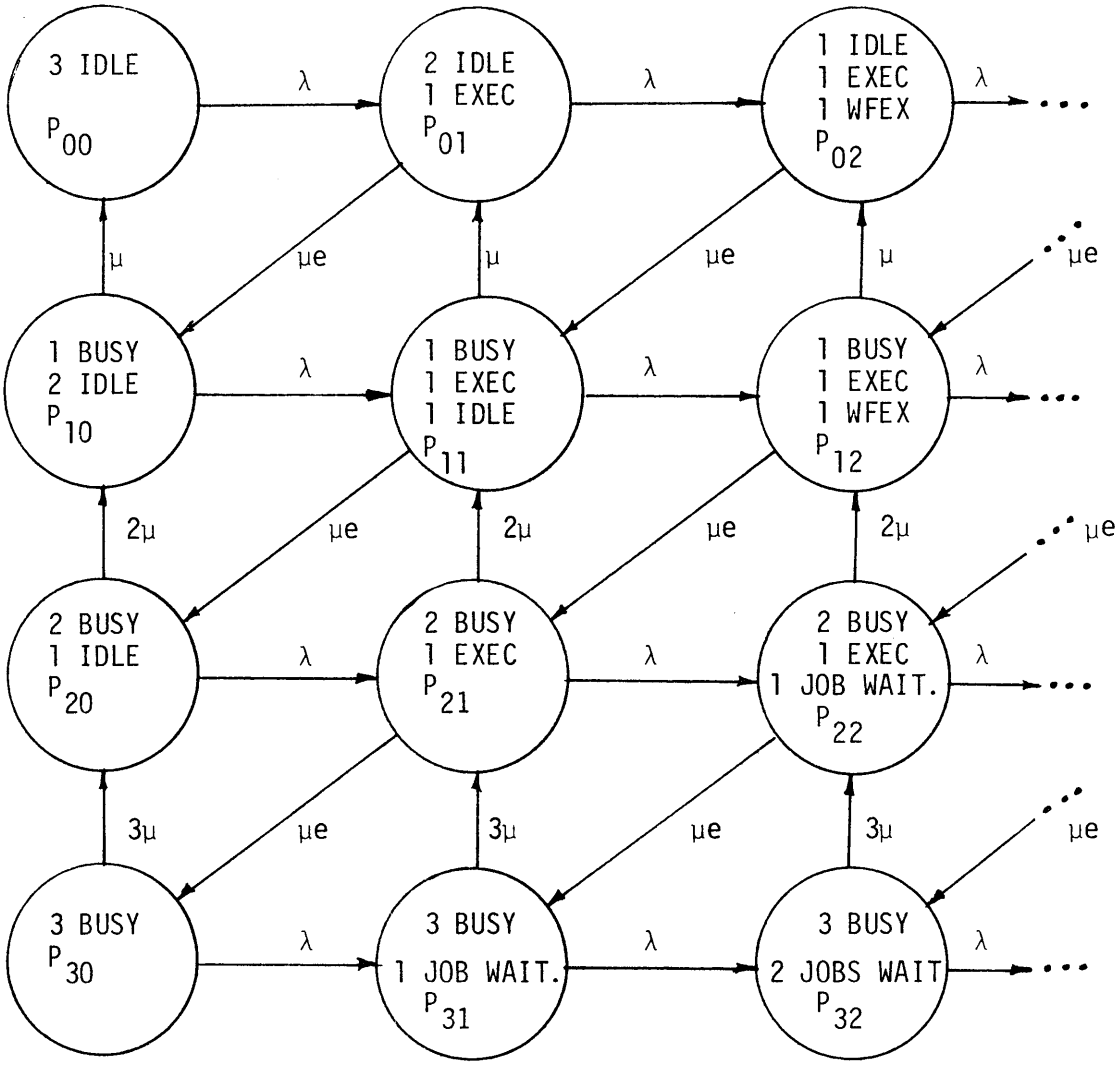


P_{mn} = Probability (m jobs running, n jobs waiting)

$\frac{1}{\lambda}$ = Mean job inter-arrival time

$\frac{1}{\mu}$ = Mean job length (run time), $\frac{1}{\mu e}$ = Mean Executive run time

Figure 7.6A



BUSY: Running a job

EXEC: Running the Executive

WFEX: Waiting for the Executive

Figure 7.6B

$$P_{00} = \frac{\mu}{\lambda} P_{10}$$

$$P_{10} = \frac{2\mu}{\lambda+\mu} P_{20} + \frac{\mu_e}{\lambda+\mu} P_{01}$$

$$P_{20} = \frac{3\mu}{\lambda+2\mu} P_{30} + \frac{\mu_e}{\lambda+2\mu} P_{11}$$

$$P_{30} = \frac{\mu_e}{\lambda+3\mu} P_{21}$$

$$P_{01} = \frac{\lambda}{\lambda+\mu_e} P_{00} + \frac{\mu}{\lambda+\mu_e} P_{11}$$

$$P_{11} = \frac{\lambda}{\lambda+\mu_e+\mu} P_{10} + \frac{2\mu}{\lambda+\mu_e+\mu} P_{21} + \frac{\mu_e}{\lambda+\mu_e+\mu} P_{02}$$

$$P_{21} = \frac{\lambda}{\lambda+\mu_e+2\mu} P_{20} + \frac{3\mu}{\lambda+\mu_e+2\mu} P_{31} + \frac{\mu_e}{\lambda+\mu_e+2\mu} P_{12}$$

$$P_{31} = \frac{\lambda}{\lambda+3\mu} P_{30} + \frac{\mu_e}{\lambda+3\mu} P_{22}$$

$$\vdots$$

$$P_{0n} = \frac{\lambda}{\lambda+\mu_e} P_{0,n-1} + \frac{\mu}{\lambda+\mu_e} P_{1n}$$

$$P_{1n} = \frac{\lambda}{\lambda+\mu_e+\mu} P_{1,n-1} + \frac{2\mu}{\lambda+\mu_e+\mu} P_{2n} + \frac{\mu_e}{\lambda+\mu_e+\mu} + P_{0,n+1}$$

$$P_{2n} = \frac{\lambda}{\lambda+\mu_e+2\mu} P_{2,n-1} + \frac{3\mu}{\lambda+\mu_e+2\mu} P_{3n} + \frac{\mu_e}{\lambda+\mu_e+2\mu} P_{1,n+1}$$

$$P_{3n} = \frac{\lambda}{\lambda+3\mu} P_{3,n-1} + \frac{\mu_e}{\lambda+3\mu} P_{2,n+1}$$

Also $$\sum_{i=0}^{\infty} (P_{0i} + P_{1i} + P_{2i} + P_{3i}) = 1 \qquad (7.8A)$$

For the finite state system with 4n states, the last four equations are:

$$
\begin{aligned}
P_{0n} &= \frac{\lambda}{\mu_e} P_{0,n-1} + \frac{\mu}{\mu_e} P_{1n} \\
P_{1n} &= \frac{\lambda}{\mu_e+\mu} P_{1,n-1} + \frac{2\mu}{\mu+\mu_e} P_{2n} \\
P_{2n} &= \frac{\lambda}{\mu_e+2\mu} P_{2,n-1} + \frac{3\mu}{\mu_e+2\mu} P_{3n} \\
P_{3n} &= \frac{\lambda}{3\mu} P_{3,n-1}
\end{aligned}
\qquad (7.8B)
$$

the three values of R investigated experimentally. The resulting steady state probabilities were then used to compute the job starting delay statistics as well as the per cent processor time devoted to job step computation, running the executive program and waiting for the executive.

The infinite state Markov process was initially approximated by a 40-state process. The accuracy of the results thus obtained was not acceptable for moderate load factors. The criterion of accuracy was chosen to be the value of the steady state probability of the last four states, P_{37} -P_{40} in the present case, measured to four decimal places. The results would be acceptable if these were all zero. The number of states was therefore increased by 20 each time until the required accuracy was obtained for load factors in the range of 80-90 per cent. All the results to be presented in this chapter have been obtained from a 120-state Markov process model.

For the purposes of delay computation it is assumed that a job is in effect started the moment it is extracted from the job queue by the executive program. For example, if the system is in state $S_{1,0}$ when a new job arrives (see Fig. 7.6B) it will be assumed that the new job is started without any delay even though the executive program has to run before the job can begin. This is how the job start delays were computed in the experiments.

Referring to Fig. 7.6 once again, it is seen that there are three final states, viz. S_{01}, S_{11} and S_{21}. S_{01}, however, can not be reached without going through S_{11} which itself is a final state. The number of final states therefore reduces to two. It is obvious from the same figure that the number of paths between an initial and one of the final states is numerous. The multiplicity of paths increases geometrically as one moves farther away from the final states. Consider for instance the initial state S_{22}. There are two different paths between S_{22} and S_{21} via S_{31} and S_{12} respectively. Also there is a third path between S_{22} and S_{11} via S_{12} and S_{02}. Following the step-by-step procedure of obtaining the delay pdf as outlined earlier

we see that each path from S_i to S_f will contribute one term to the series. The three terms, weighed appropriately, are

$$S_{22}\text{-}S_{31}\text{-}S_{21} : \frac{\mu_e}{\mu_e+s} \cdot \frac{3\mu}{3\mu+s} \cdot \frac{\mu_e}{\mu_e+2\mu} \cdot P_{21}$$

$$S_{22}\text{-}S_{12}\text{-}S_{21} : \frac{2\mu}{2\mu+s} \cdot \frac{\mu_e}{\mu_e+s} \cdot \frac{2\mu}{2\mu+\mu_e} \cdot \frac{\mu_e}{\mu_e+\mu} \cdot P_{21}$$

$$S_{22}\text{-}S_{12}\text{-}S_{02}\text{-}S_{11} : \frac{2\mu}{2\mu+s} \cdot \frac{\mu}{\mu+s} \cdot \frac{\mu_e}{\mu_e+s} \cdot \frac{2\mu}{2\mu+\mu_e} \cdot \frac{\mu}{\mu+\mu_e} \cdot P_{21}$$

The next step is to repeat the above procedure for all initial states up to $S_{3,29}$ which is the 120th state. Inverting each term and summing all the terms in the time domain leads to an expression for the pdf of the job starting delay.

This procedure, however, is too long and tedious to be of any practical value. Some approximations are therefore in order. First of all it is not necessary to consider all the paths between an initial state and a final state since some paths are much more likely to be followed than others. For instance, for R=10 ($\frac{\mu_e}{\mu} = 10$) the relative likelihood of the three paths originating from S_{22} is as follows:

$$S_{22}\text{-}S_{31}\text{-}S_{21} : \frac{10}{12}$$

$$S_{22}\text{-}S_{12}\text{-}S_{21} : \frac{2}{12} \cdot \frac{10}{11}$$

$$S_{22}\text{-}S_{12}\text{-}S_{02}\text{-}S_{11} : \frac{2}{12} \cdot \frac{1}{11}$$

Evidently the third path is at least an order of magnitude less probable than the other two. Hence it may be ignored. A second approximation may be made by replacing the transition rate between states, which is actually a random variable, by its mean value. Such an approximation considerably simplifies the mathematics of obtaining the delay pdf. In particular, it obviates the need for doing the algebraic manipulations in the frequency domain. Therefore it is not necessary to compute the Laplace transforms and their inverses. To illustrate this point further as well as to compare the exact and approximate results, consider the simplified Markov model of Fig. 7.4. The exact probability distribution function for this system is given by Equation 7.6. The exact cumulative distribution function (cdf) for the same system obtained by integrating Eqn. 7.6 becomes

$$\begin{aligned} \text{cdf(delay)} &= \text{prob. (delay} \leq t) \\ &= 1 - \frac{9\rho^3}{2+4\rho+3\rho^2} \cdot e^{-3\mu t(1-\rho)} \\ \text{or} \quad \text{Prob. (delay} \leq d) &= 1 - \frac{9\rho^3}{2+4\rho+3\rho^2} \cdot e^{-3d(1-\rho)} \qquad (7.9) \\ \text{where} \quad d &= \text{normalized delay} \end{aligned}$$

The approximate analysis on the other hand results in the following discrete cdf.

$$\begin{aligned} \text{Prob. (delay} \leq \tfrac{1}{3}) &= \sum_{i=0}^{3} P_i \\ \text{Prob. (delay} \leq \tfrac{2}{3}) &= \sum_{i=0}^{4} P_i \\ \text{etc.} \quad &\vdots \end{aligned} \qquad (7.10)$$

The numerical values of the exact and approximate cdfs for three different load factors are tabulated in Tables 7.1-7.3. The approximate probabilities are about 2 to 5 per cent higher than the exact figures. The resulting errors are small enough to justify the use of the approximate procedure for obtaining the delay statistics. The results for the Markov model of Figure 7.6, obtained by the approximate analysis, are presented in Figures 7.7 through 7.16.

7.4 Markov Model Results

It is seen from Figures 7.7 through 7.16 that the fidelity of the results obtained from the second model is significantly better than that obtained from the first one. In particular, each case of R is identified separately. Second, with the exception of the maximum delay, the overall characteristics and nature of the theoretical curves is very similar to the experimental ones. There are discrepancies in the numerical values, but these can generally be explained. For instance, the Markov model did not include the 'wait for memory bus' and 'fetch instructions from the memory' states of the system. Therefore the theoretical values for throughput efficiency are found to be much higher. However when 'bus wait and use' percentages of processor time are included in the experimental throughput efficiency figures, the differences become small. This is shown in Figure 7.8. Figure 7.9 compares for R=10 the theoretical and the experimental values of the processor time devoted to the job step computation, the executive program and waiting for the executive. In this set of experiments the jobs were assumed to reside in each processor memory. That is, there was no memory bus use to fetch instructions.

Figure 7.10 shows the normalized mean delay as a function of the processor load for the three values of R. The corresponding experimental curves are shown in the same figure. Figure 7.11 compares the absolute values of the mean delay. Figure 7.12 shows the per cent of jobs that do

TABLE 7.1

LOAD = 25.0% NORMALIZED DELAY	CDF	
	APPRX.	EXACT
0.0	0.9550	0.9550
0.33	0.9890	0.9792
0.67	0.9972	0.9962
1.00	0.9993	0.9954
1.33	0.9998	0.9972
1.67	1.0000	0.9994
2.00	1.0000	0.9995
2.33	1.0000	0.9998
2.67	1.0000	0.9999
3.00	1.0000	0.9999
3.33	1.0000	1.0000
3.67	1.0000	1.0000
4.00	1.0000	1.0000

TABLE 7.2

LOAD = 50.0% NORMALIZED DELAY	CDF	
	APPRX.	EXACT
0.0	0.7632	0.7632
0.33	0.8816	0.8583
0.67	0.9402	0.9129
1.00	0.9704	0.9472
1.33	0.9852	0.9679
1.67	0.9926	0.9806
2.00	0.9963	0.9882
2.33	0.9981	0.9928
2.67	0.9991	0.9957
3.00	0.9995	0.9974
3.33	0.9998	0.9994
3.67	0.9999	0.9994
4.00	0.9999	0.9994

TABLE 7.3

LOAD = 75.0% NORMALIZED DELAY	CDF	
	APPRX.	EXACT
0.0	0.4322	0.4322
0.33	0.5742	0.5579
0.67	0.6296	0.6556
1.00	0.7695	0.7318
1.33	0.8204	0.7911
1.67	0.8653	0.8373
2.00	0.8990	0.8733
2.33	0.9242	0.9013
2.67	0.9432	0.9232
3.00	0.9574	0.9492
3.33	0.9680	0.9554
3.67	0.9760	0.9637
4.00	0.9829	0.9717

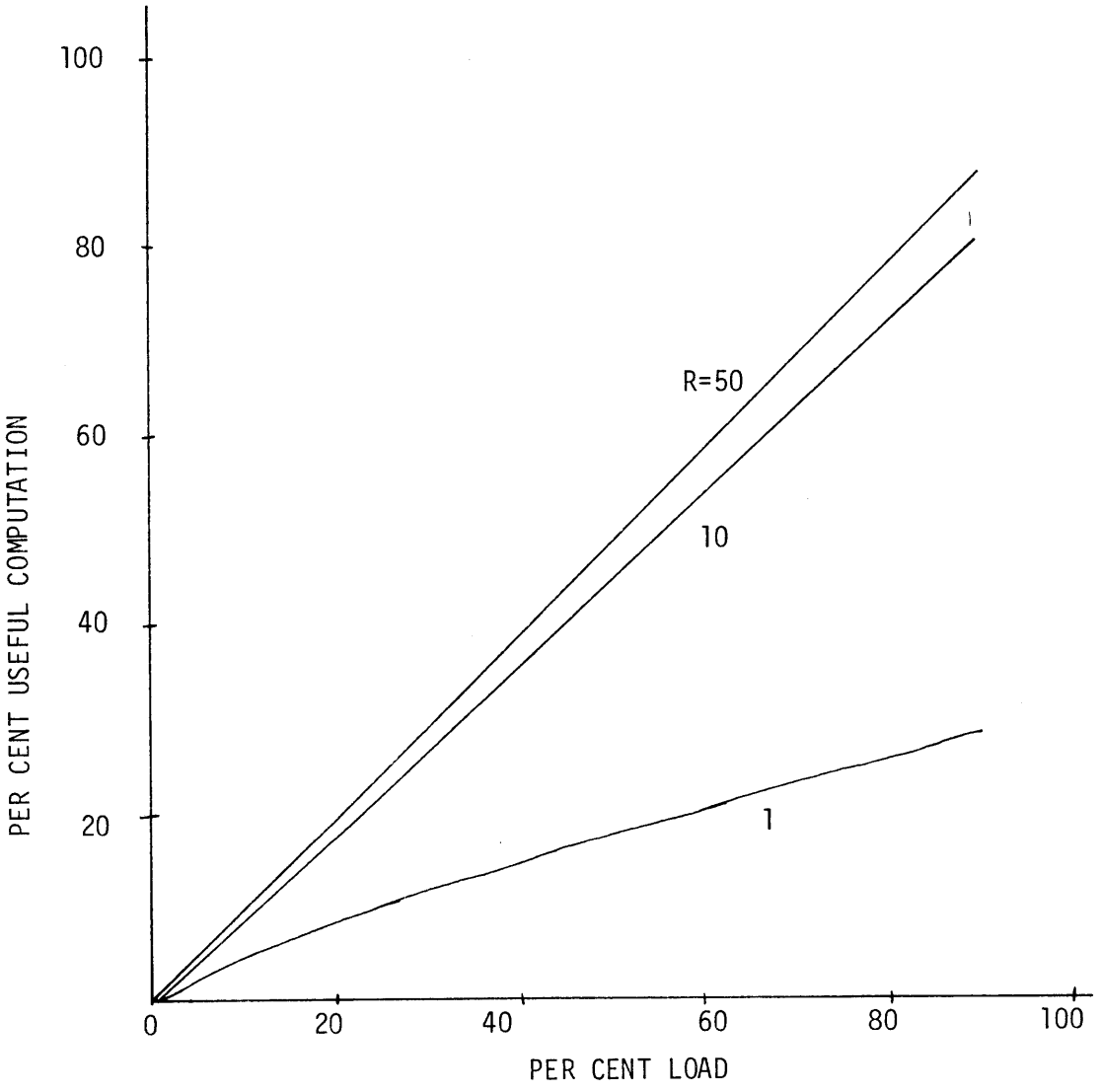

100
80
60
40
20
PER CENT USEFUL COMPUTATION
R=50
10
1
0
20
40
60
80
100
PER CENT LOAD

Figure 7.7

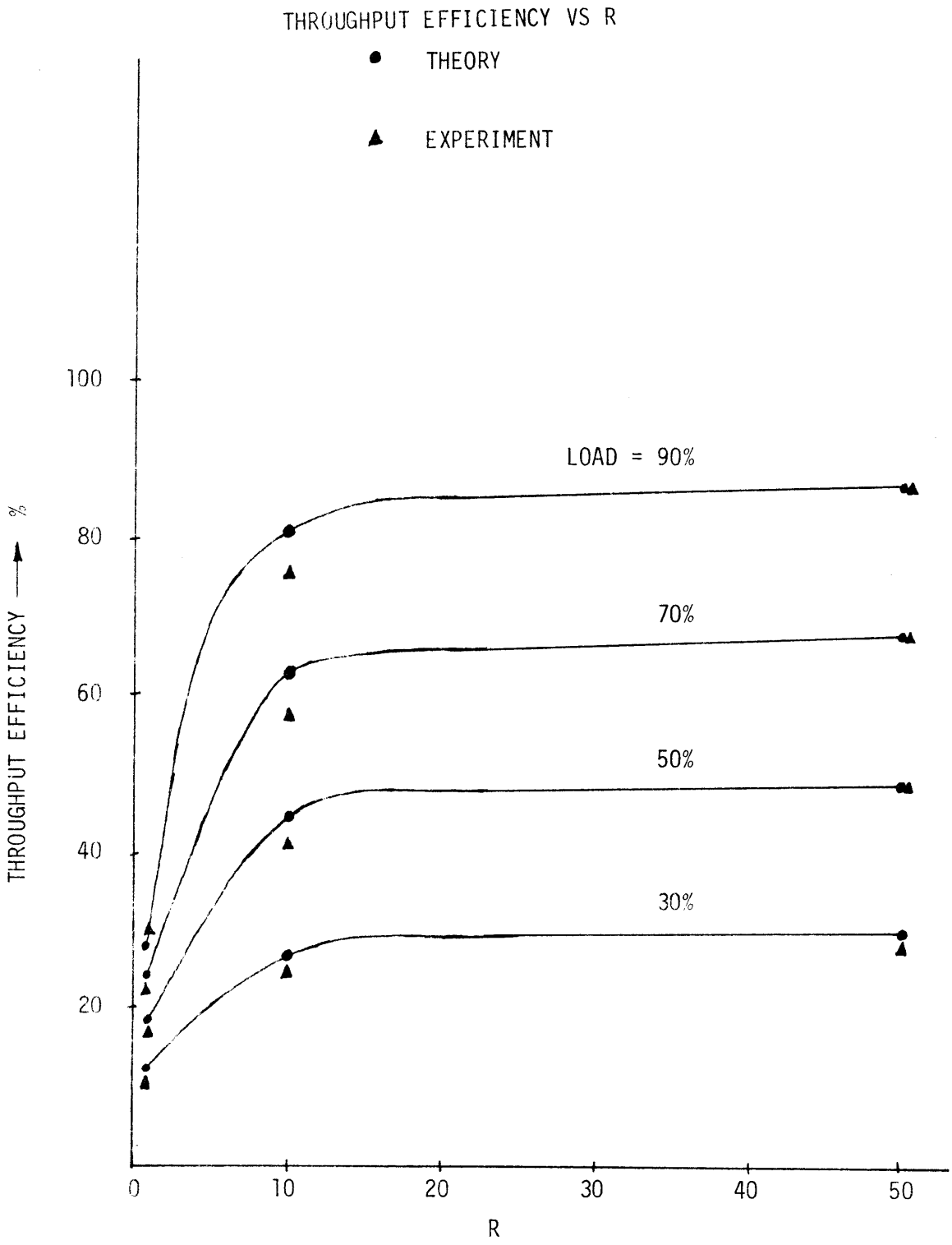

THROUGHPUT EFFICIENCY VS R
THEORY
EXPERIMENT
LOAD = 90%
70%
50%
30%
THROUGHPUT EFFICIENCY ——▲ %
100
80
60
40
20
0
10
20
30
40
50
R

Figure 7.8

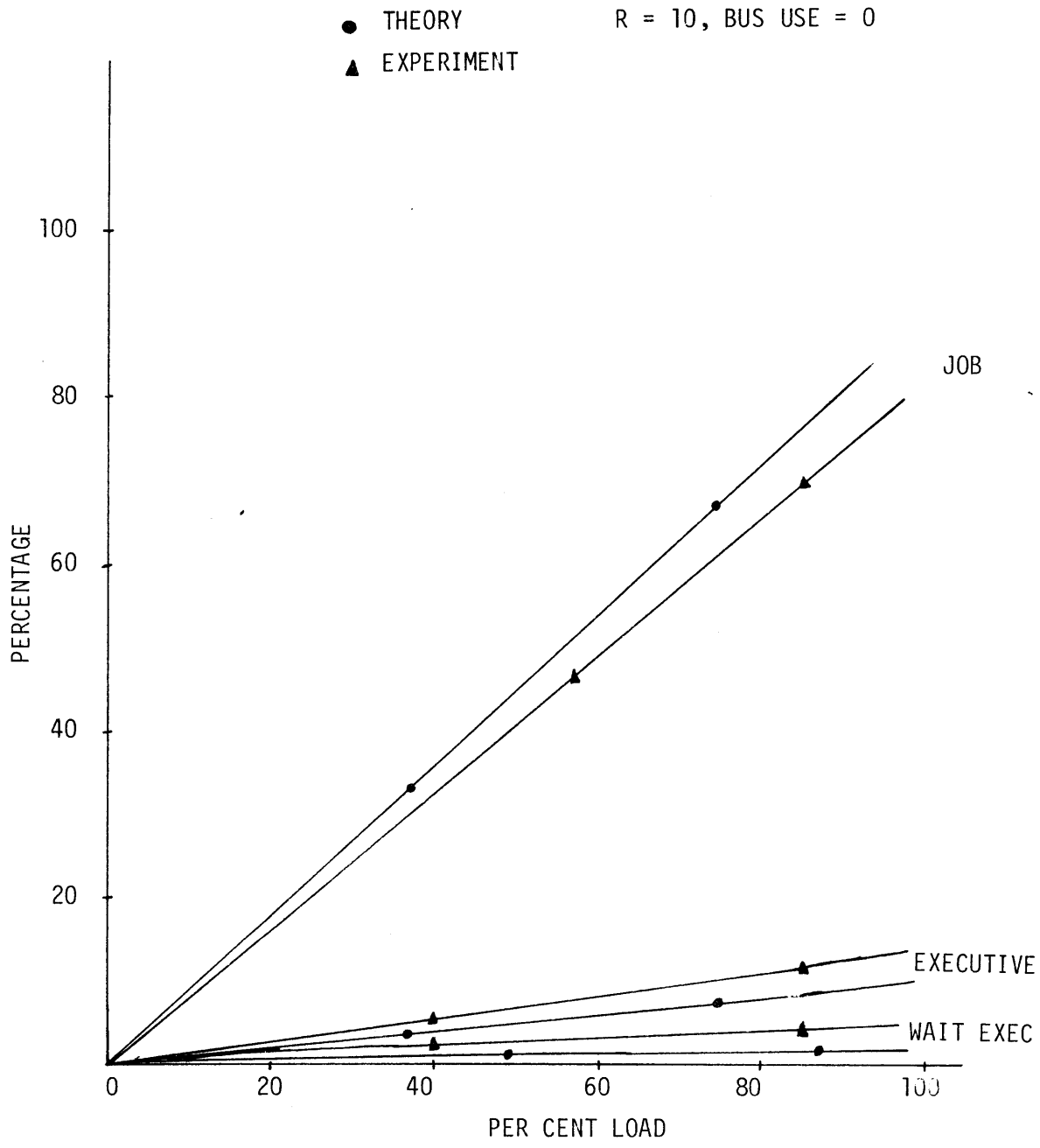

THEORY
EXPERIMENT
R = 10, BUS USE = 0
100
80
60
40
20
PERCENTAGE
JOB
EXECUTIVE
WAIT EXEC
0
20
40
60
80
100
PER CENT LOAD

Figure 7.9

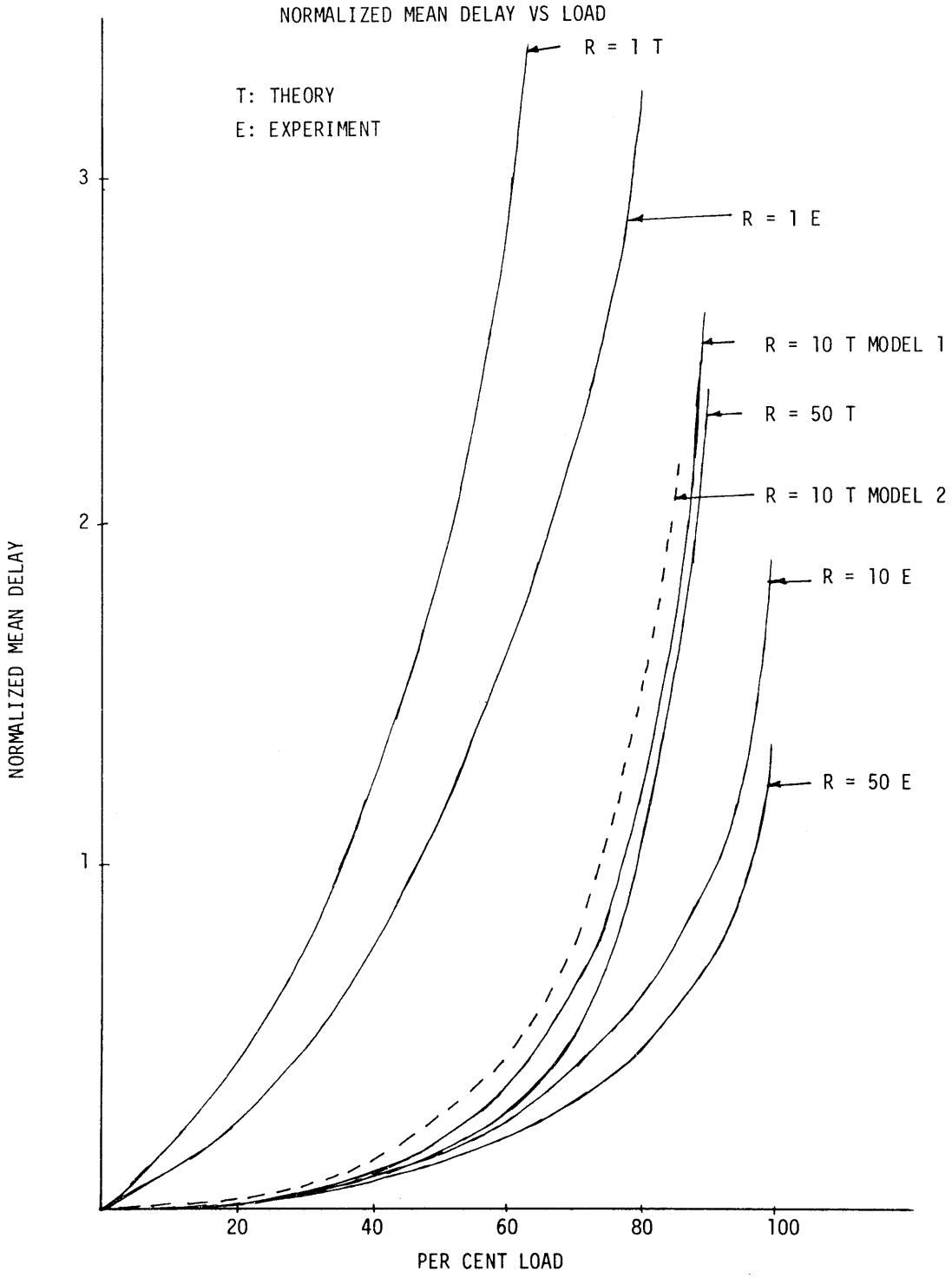

NORMALIZED MEAN DELAY VS LOAD
T: THEORY
E: EXPERIMENT
R = 1 T
R = 1 E
R = 10 T MODEL 1
R = 50 T
R = 10 T MODEL 2
R = 10 E
R = 50 E
NORMALIZED MEAN DELAY
3
2
1
20
40
60
80
100
PER CENT LOAD

Figure 7.10

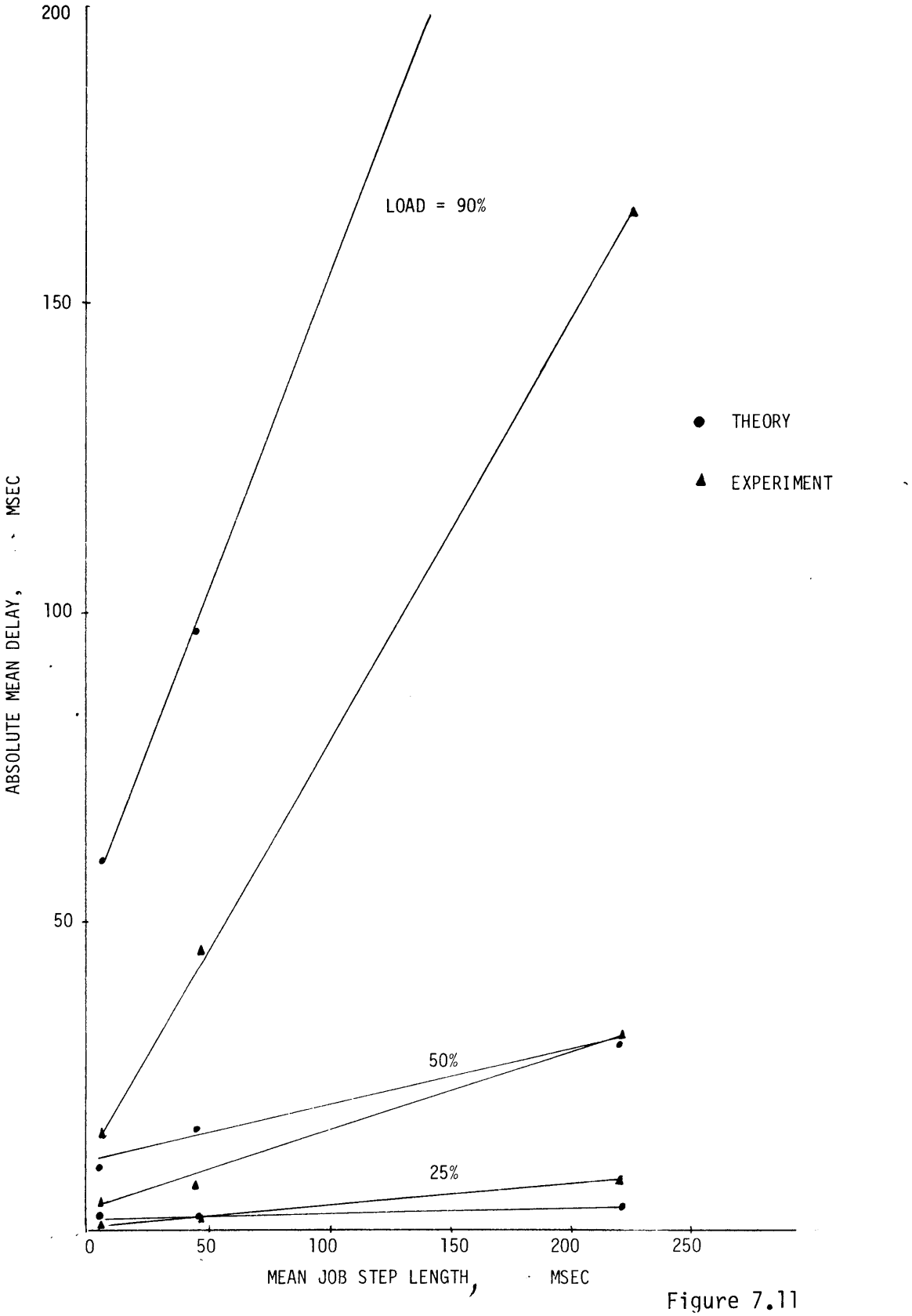

200
150
100
50
0
50
100
150
200
250
LOAD = 90%
50%
25%
THEORY
EXPERIMENT
ABSOLUTE MEAN DELAY, MSEC
MEAN JOB STEP LENGTH, MSEC

Figure 7.11

T: THEORY

E: EXPERIMENT

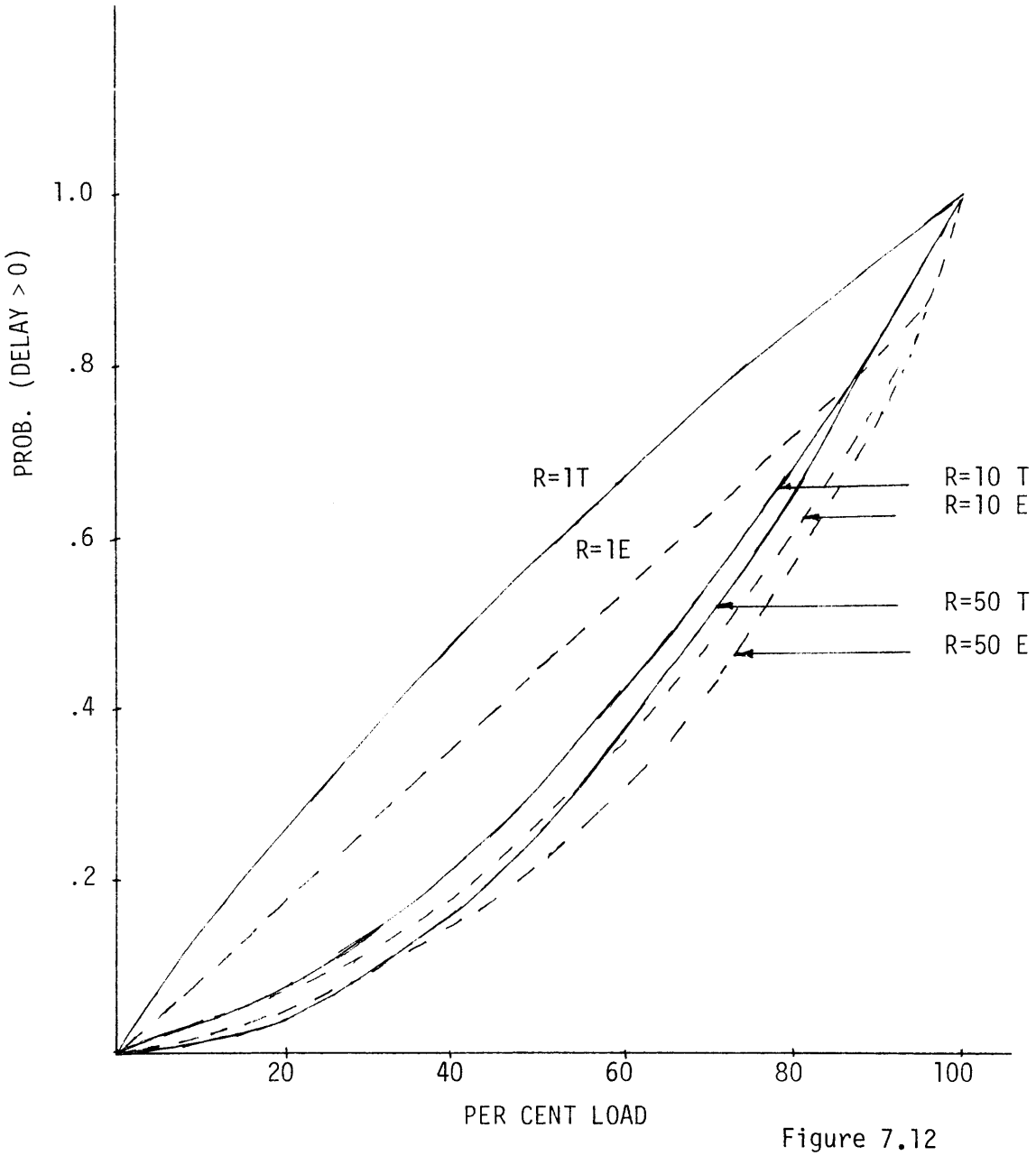


Figure 7.12

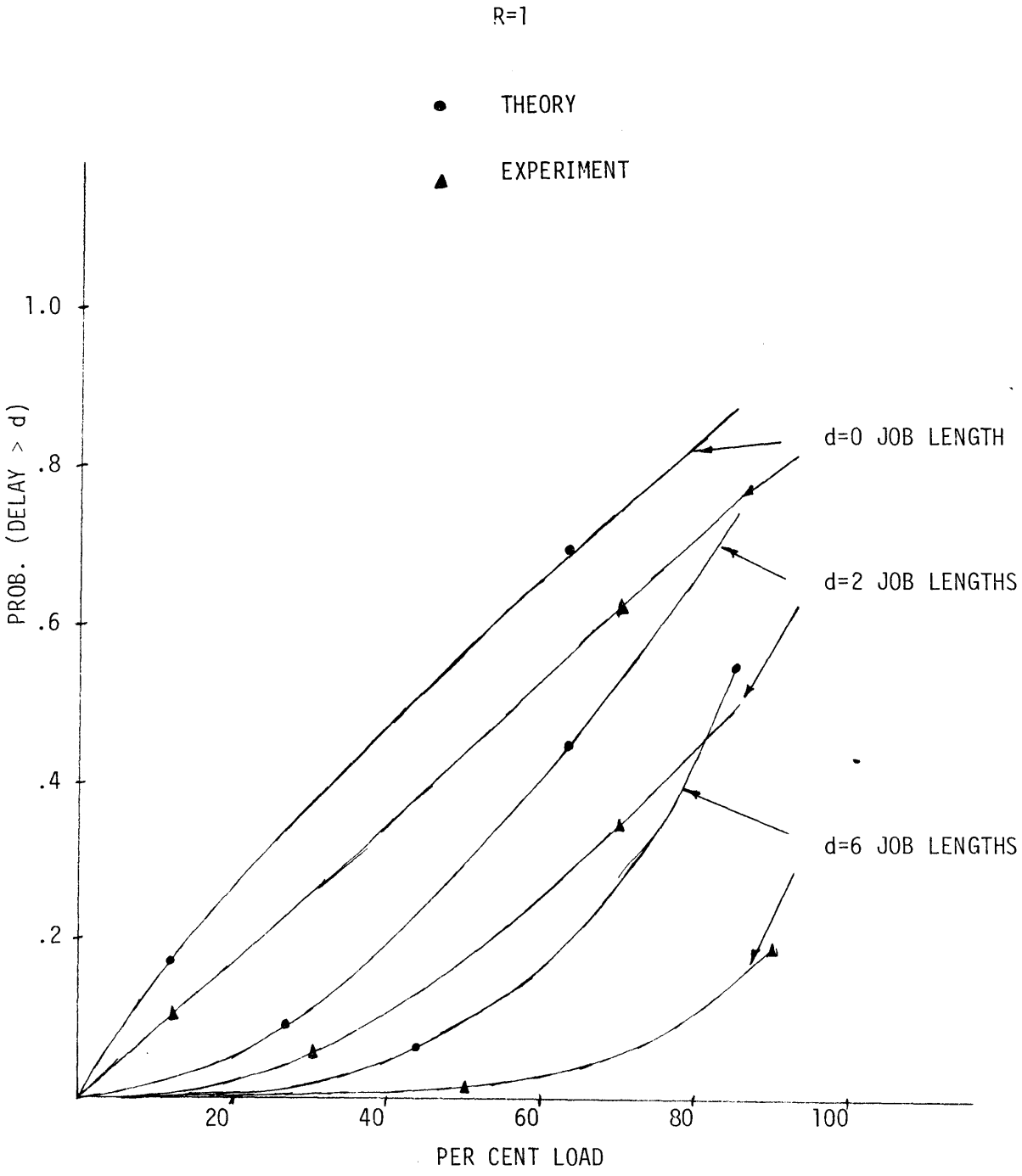

R=1
THEORY
EXPERIMENT
1.0
.8
.6
.4
.2
PROB. (DELAY > d)
d=0 JOB LENGTH
d=2 JOB LENGTHS
d=6 JOB LENGTHS
20
40
60
80
100
PER CENT LOAD

Figure 7.13

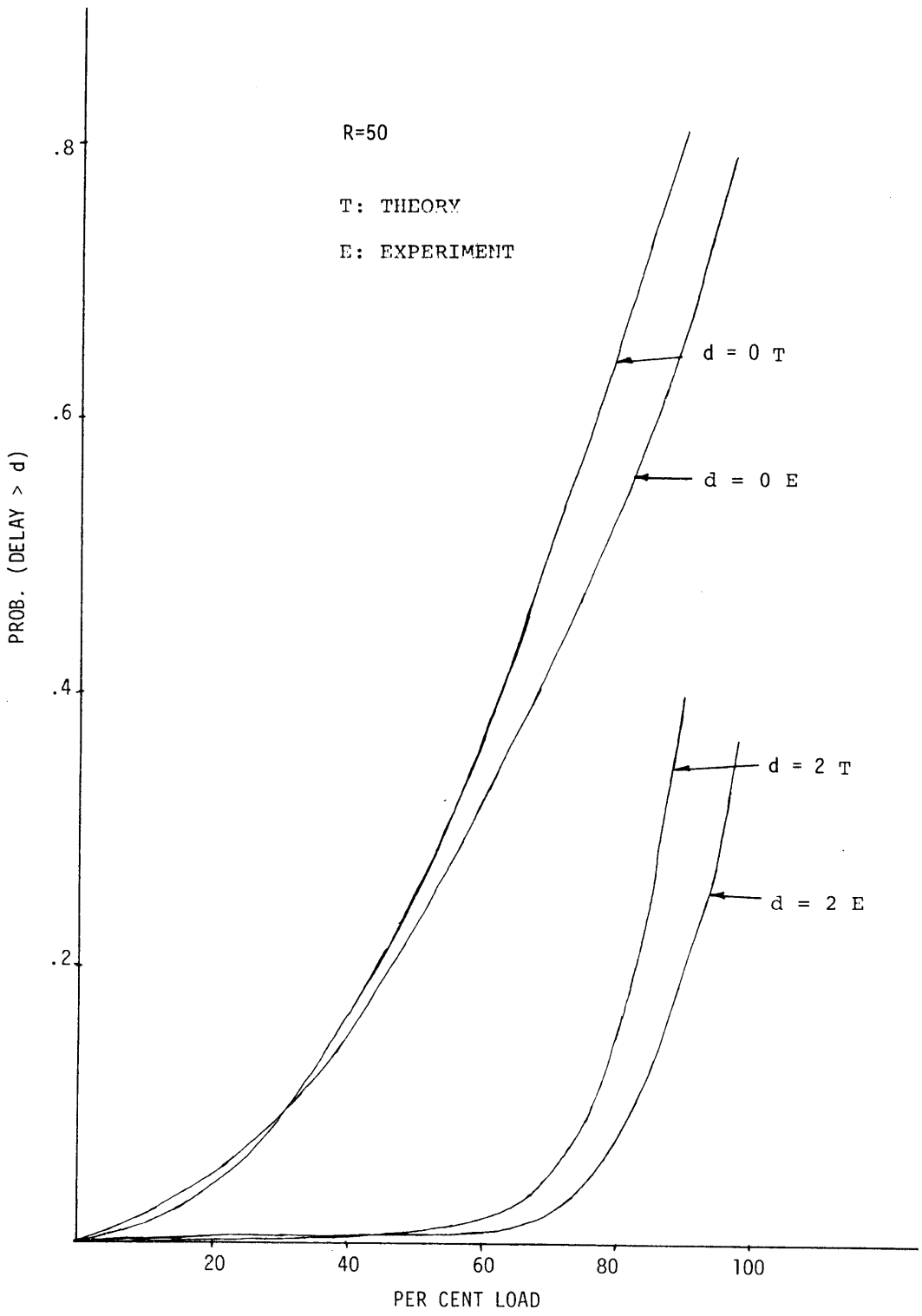

R=50
T: THEORY
E: EXPERIMENT
d = 0 T
d = 0 E
d = 2 T
d = 2 E
PROB. (DELAY > d)
.8
.6
.4
.2
20
40
60
80
100
PER CENT LOAD

Figure 7.14

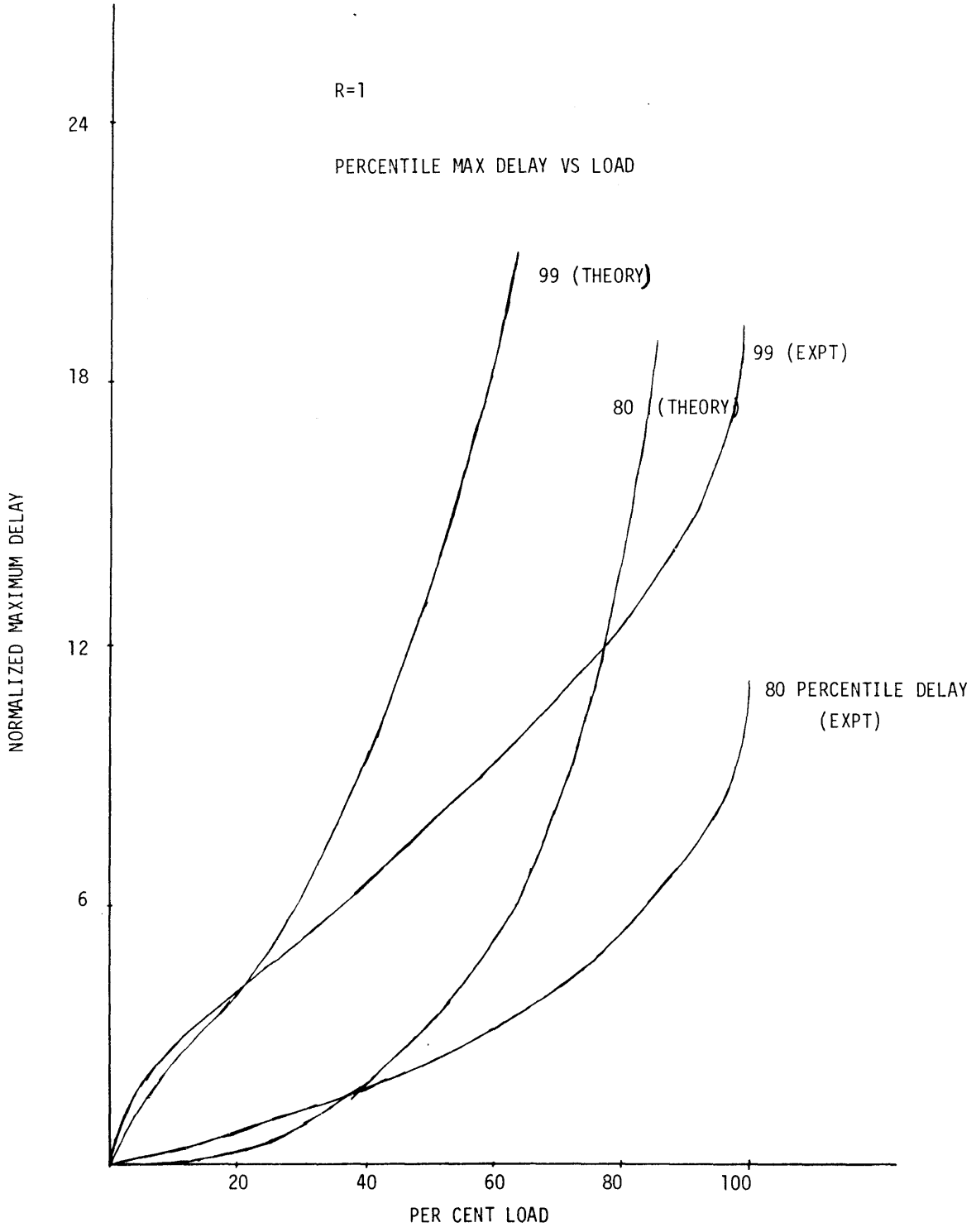
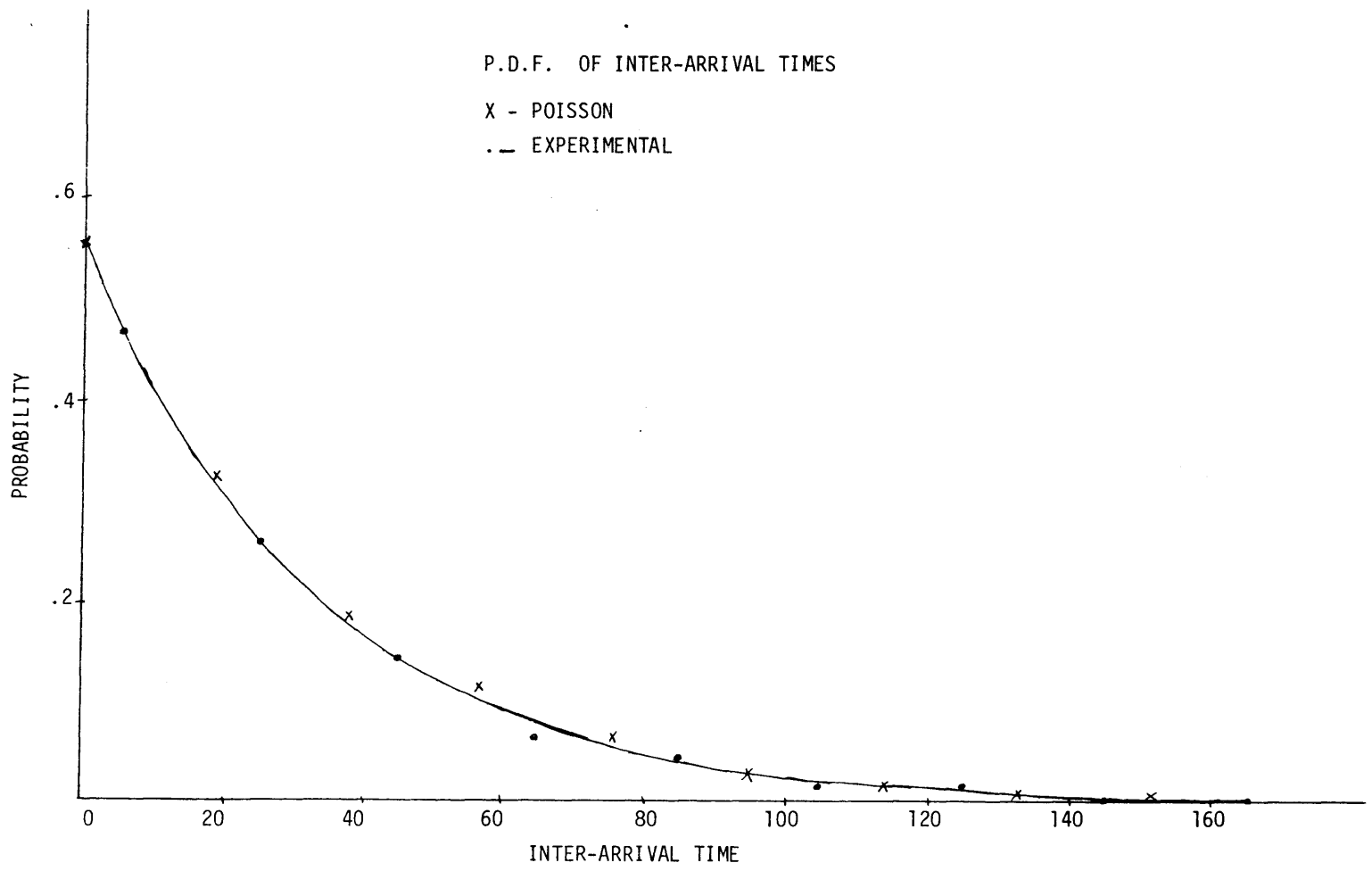

R=1
PERCENTILE MAX DELAY VS LOAD
99 (THEORY)
99 (EXPT)
80 (THEORY)
80 PERCENTILE DELAY
(EXPT)
NORMALIZED MAXIMUM DELAY
24
18
12
6
20
40
60
80
100
PER CENT LOAD

Figure 7.15

Fig. 7.16


P.D.F. OF INTER-ARRIVAL TIMES
X - POISSON
. — EXPERIMENTAL
PROBABILITY
.6
.4
.2
0
20
40
60
80
100
120
140
160
INTER-ARRIVAL TIME

not start on-time, that is, the per cent of jobs that are delayed. The probability of exceeding a certain delay for R=1 and 50 is shown in the next two figures. Figure 7.15 shows the percentile delays for R=1. The experimental values of these functions have also been superimposed in the respective figures for ease of comparison. It is evident from such a comparison that the theoretical delay values are higher than the corresponding simulation figures. The reason behind this difference is the following. In the Markov model the job arrivals and completions were modeled by exponentially distributed random variables. Figure 7.16 shows the experimental probability distribution function of the job inter-arrival time. This is seen to be in accordance with the model used. However the job length, that is , the job service time, has a much smaller variance than an exponential distribution implies. The impact of the variance of the service time on the queue length and hence the job waiting time can be considerable. This can be illustrated by a simple example as follows.

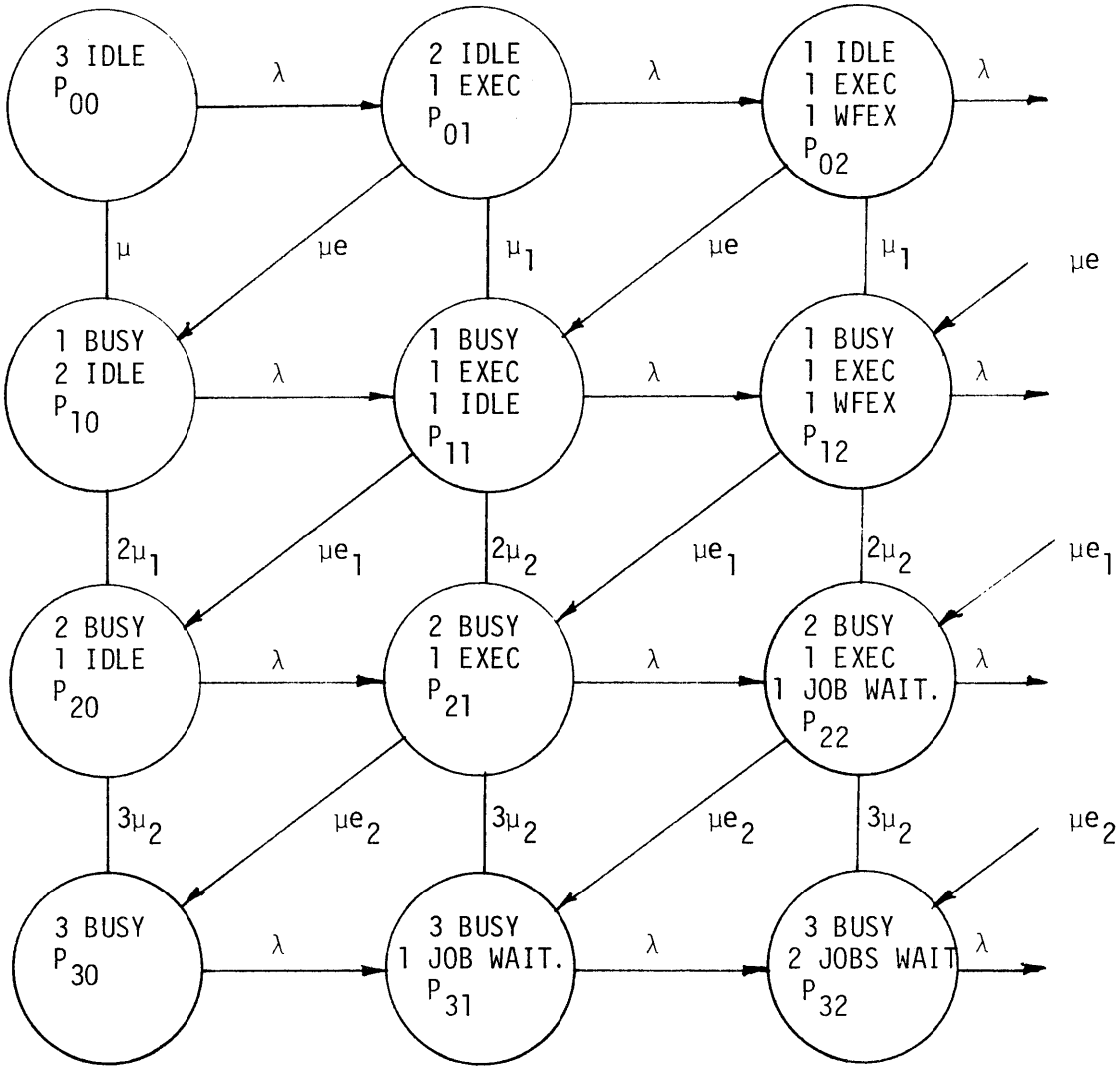
Consider a single server facility at which the arrival of customers is a Poisson process, with an average rate of λ arrivals per unit time. Let the service time be represented by an exponentially distributed random variable with a mean of $1/\mu$ time units. It can be shown that the mean waiting time of a customer at such a hypothetical service facility is given by $\frac{\mu}{\mu - \lambda}$ time units or $\frac{\rho}{1-\rho}$ mean job lengths. If however the service time were constant (zero variance) at $1/\mu$ time units the mean waiting time reduces to $\rho(1-\rho/2)/(1-\rho)$ [41]. That is, the mean waiting time decreases by a factor of $(1-\rho/2)$. Consequently, the discrepancies between the experimental and the theoretical delay may be attributed to the variance of job service times.

The only area of real disagreement between the results is then the percentile of maximum delay. The shape of the percentile delay functions, as shown in Figure 7.15 for example, near the origin is similar to the experimental curves. Beyond that, the functions for 99 and 99.9 percentile delays do become almost linear but the trend is not as strong as it is in the experimental figures.

As pointed out earlier, the present model does not include the 'bus use' and 'wait for bus' states of the system. The Markov model was therefore altered slightly to include the effects of the memory bus contention. The bus is used to fetch instructions and data for the Executive and the job steps. If at any time more than one processor wants to access the memory, all but one processor will be denied access and delayed. There is no bus contention if only one processor is running the Executive or a job and the other two are idle. There is some interference when two processors are busy and more when all are busy. As a result the job and the executive completion rates become smaller. The following procedure was used to compute the new job and executive completion rates.

It was found from the simulation experiments that on the average the processor time spent running the job, using the bus, and waiting for the bus, is approximately in the ratio of 2/3, 1/6 and 1/6. These are only the average values for all Rs and load factors. For higher values of R, the job fraction is slightly higher and the bus wait is slightly lower, and vice versa. In the Markov model the 'job' and the 'bus use' states are combined into a single state called 'busy'. Using the experimentally found average ratios it is seen that a processor spends 1/5th of the 'busy' state time using the bus. As a result, bus interference reduces the job and executive completion rates by 4 per cent ($0.2^2 = 0.04$) when any two processors are busy and by 13 per cent ($0.2^3 + 3X\ 0.2^2 = 0.128$) when all the processors are busy.

With the modified state transition rates (see Fig. 7.17) the Markov model was once again solved to obtain the steady state probabilities for a range of load factors. Qualitatively there was no difference between the results obtained from the two models. Quantitatively, however, the job start delays were slightly higher in the second model. This is evidently due to memory interference. Figure 7.18, for example, shows the 99.9 percentile maximum delay for the two models. Figure 7.10 compares the normalized mean delay for R=10 as obtained from the two models.



BUSY: Running a job

EXEC: Running the Executive

WFEX: Waiting for the Executive

$\mu e_1 = 0.96\ \mu e$, $\mu e_2 = 0.87\ \mu e$

$\mu_1 = 0.96\ \mu$, $\mu_2 = 0.87\ \mu$

Fig. 7.17.

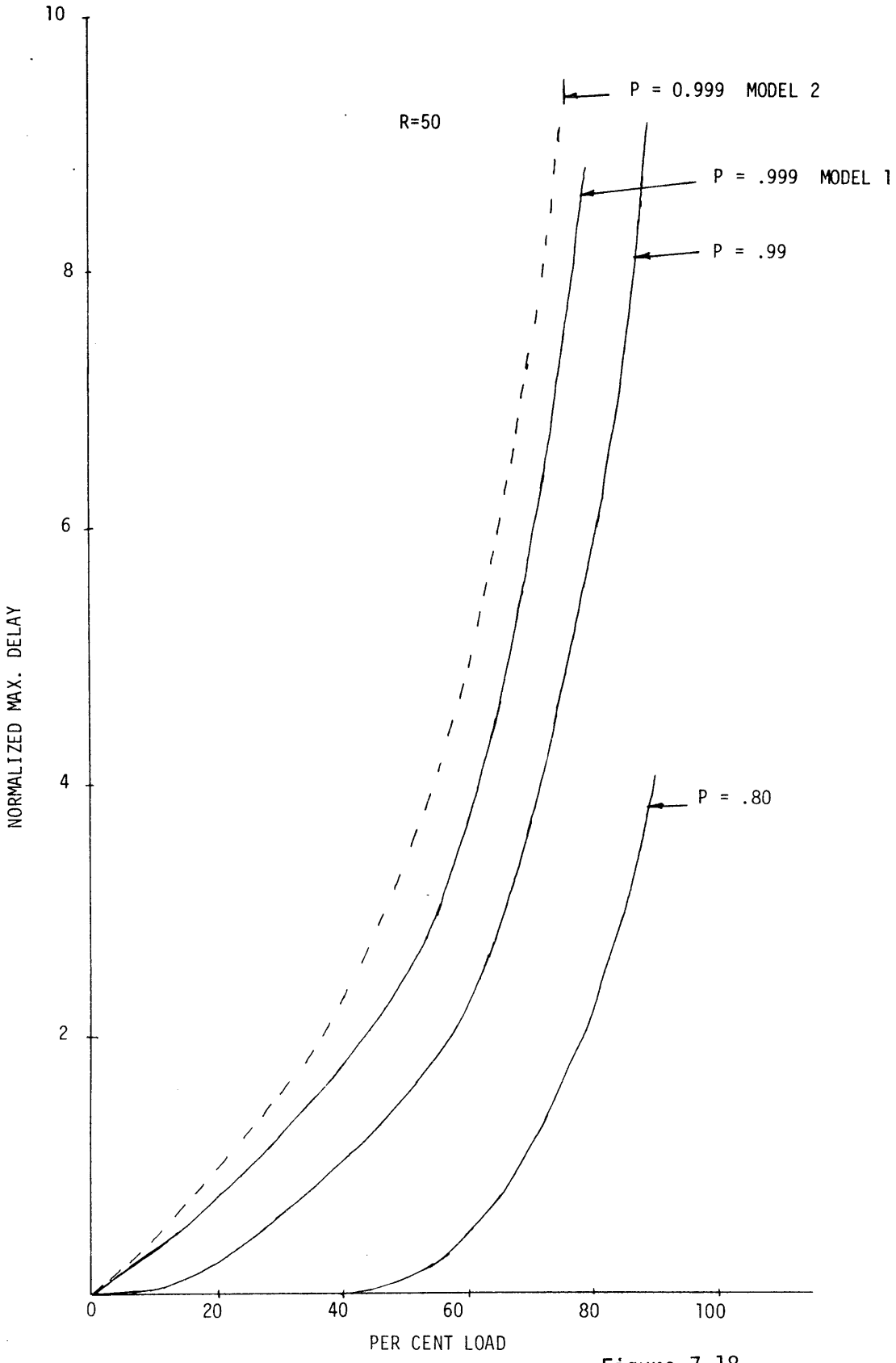

10
8
6
4
2
0
20
40
60
80
100
NORMALIZED MAX. DELAY
PER CENT LOAD
R=50
P = 0.999 MODEL 2
P = .999 MODEL 1
P = .99
P = .80

Figure 7.18

7.5 Erlang Model and Results

It is evident from the discussion of the last section that although the overall qualitative characteristics of the theoretical and the experimental results are similar, there are some real differences in the numerical values. It was also illustrated with the help of an example in the same section that the differences in the assumed and the actual frequency distributions of job lengths could possibly be blamed for the descrepancies in the results. In particular, the probability distribution function of job lengths in the simulation experiments is Gaussian, while the Markov model is based on an exponential distribution. To test the hypothesis put forward above, a new more complex model of the experimental situation was developed. This is described in what follows.

It is well known that by definition the transition rate from one state to another in a Markov process is an exponentially distributed random variable. Therefore it would seem that we are limited to this kind of frequency distribution when using a Markov model. However sometimes it is possible to introduce new states between two given states, thereby breaking up a single transition into a series of proportionately faster transitions. The new distribution of the transition rate between the old two states is the convolution of a number of exponential functions which represent the new transitions. The convolution of n exponentials results in a frequency distribution known as an n-Erlang function. As n approaches infinity, the n-Erlang distribution approaches the normal distribution (the central limit theorem) [42]. The new Markov model with 3-Erlang job length and Executive length distributions is shown in Figs. 7.19A and 7.19B. The reasons for choosing the third order distribution are:

1. It is the minimum acceptable order for matching with the Gaussian function.

3-ERLANG MARKOV MODEL

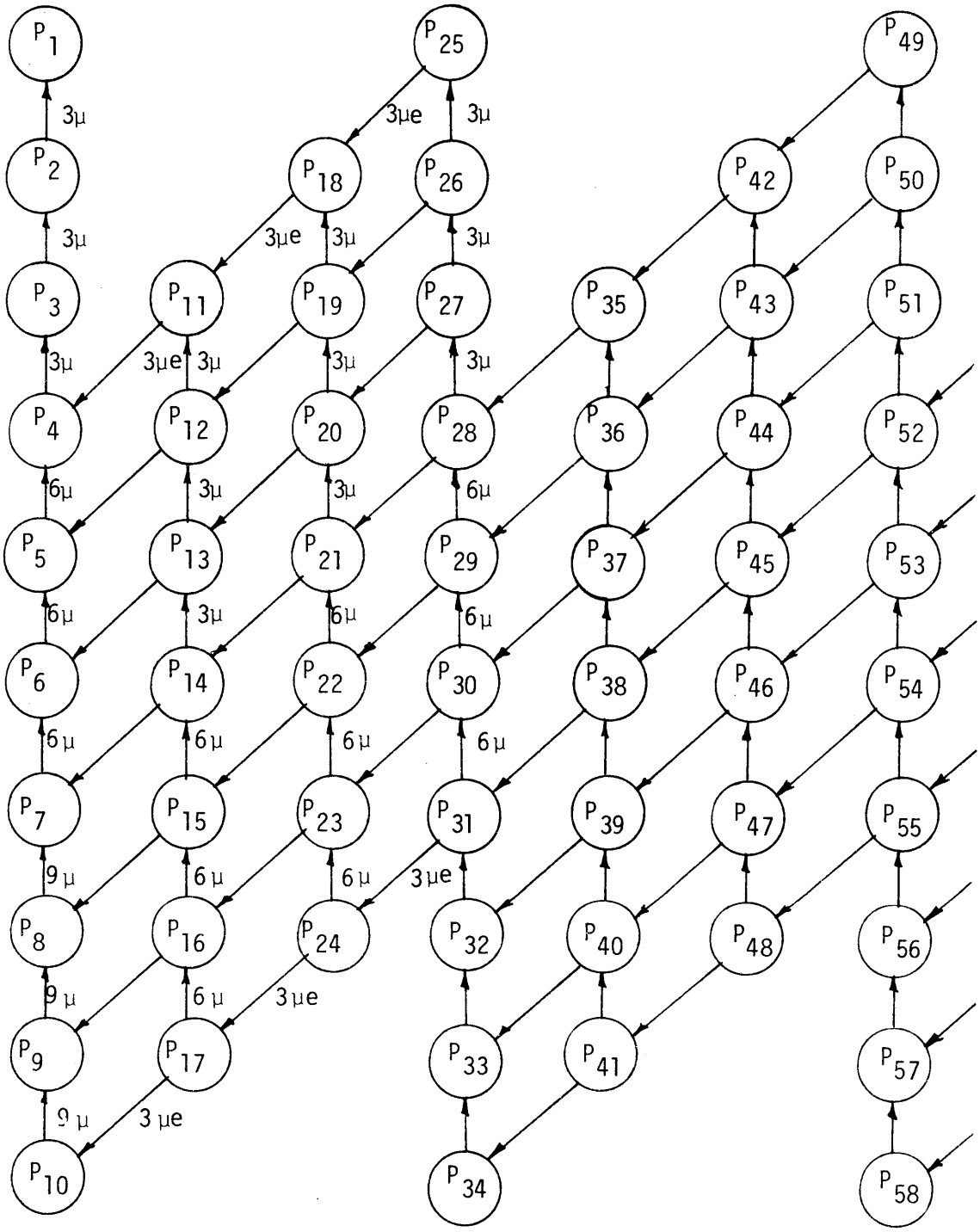


Fig. 7.19A. Transitions due to Job and Executive Completions.

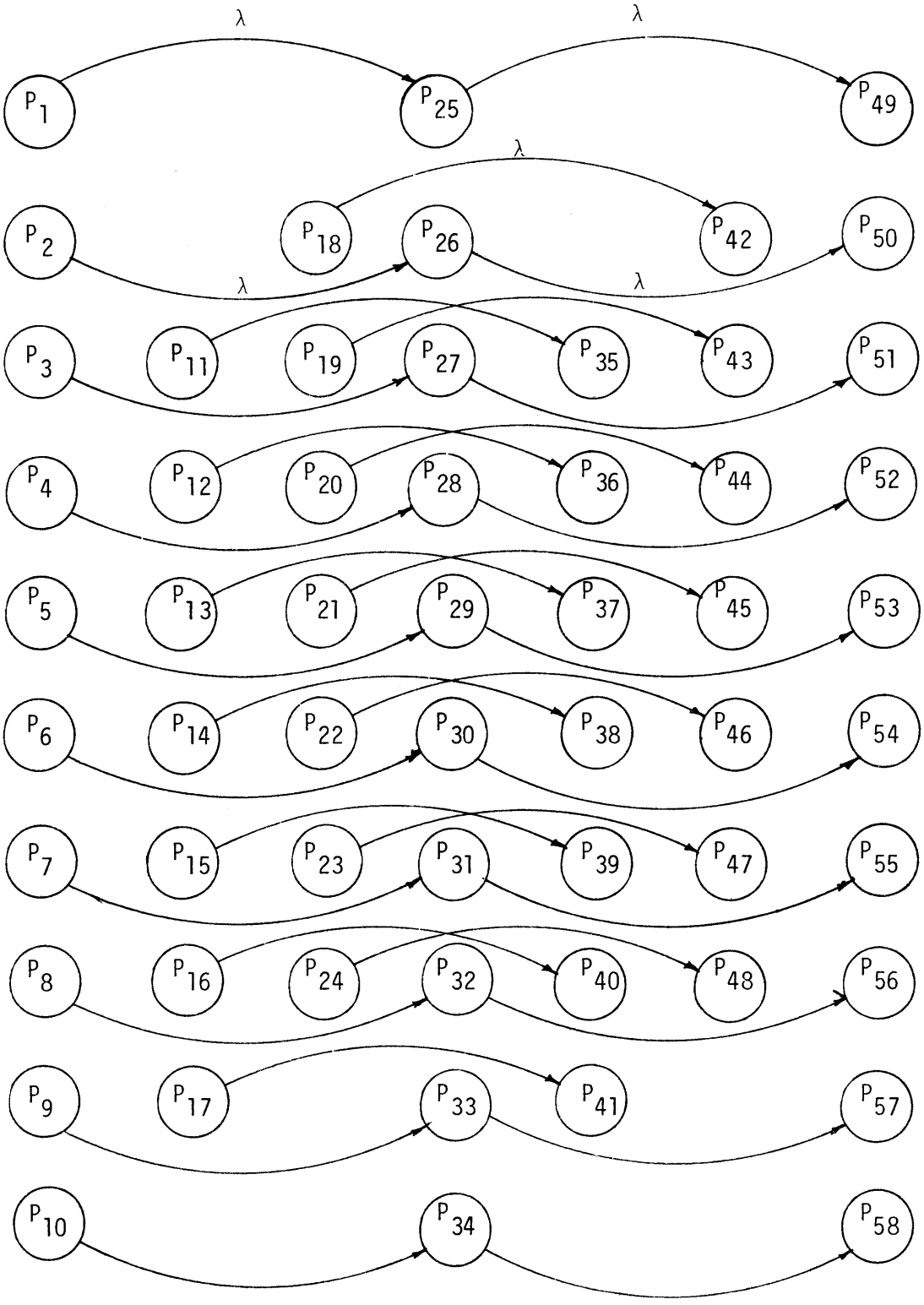


Fig. 7.19B. Transitions due to Job Arrivals.

2. A higher order will most likely not result in a significant change in results.

3. Assuming that the maximum number of states that can be handled on the digital computer for the numerical solution of the model is fixed, a higher Erlang distribution would result in fewer significant states in the model.

Comparing the Erlang model (Fig. 7.19) to the Exponential model (Fig. 7.6B) it is seen that the states P1 (3 Idle), P4 (1 Busy, 2 Idle), P7 (2 Busy, 1 Idle) and P10 (3 Busy) of the new model correspond to states P00, P10, P20 and P30 respectively of the old model. In the old model a job completion was represented by a single transition such as from state P30 (3 Busy) to state P20 (2 Busy, 1 Idle). The interval between these transitions was an exponentially distributed random variable with a mean value of $1/3\ \mu$ time units. In the Erlang model the two corresponding states are P7 and P10 respectively. In addition there are two more states, P8 and P9, between P7 and P10. A job completion is therefore signified by the sum of three independent transitions (P10 to P9, P9 to P8, and P8 to P7). Each transition signifies completion of one third of a job. The intervals between these transitions are exponentially distributed random variables with a mean of $1/9\ \mu$ time units. It is seen, therefore, that the transition interval from P10 to P7 is a random variable with a mean value of $1/3\ \mu$ and a 3-Erlang frequency distribution. The same discussion applies to transitions corresponding to completions of the Executive. As a result the number of states in the new model is increased six-fold.

The steady state equations for the 3-Erlang Markov model are shown as Eqns. 7.11 and 7.12. The model was truncated at 226 states which implies a maximum queue length of waiting jobs of nine. The accuracy criterion described previously showed the model to be valid for loads up to 65 to 70

$$P_1 = \frac{3\mu}{\lambda}P_2$$

$$P_2 = \frac{3\mu}{3\mu+\lambda}P_3$$

$$P_3 = \frac{3\mu}{3\mu+\lambda}P_4$$

$$P_4 = \frac{6\mu}{3\mu+\lambda}P_5 + \frac{3\mu_e}{3\mu+\lambda}P_{11}$$

$$P_5 = \frac{6\mu}{6\mu+\lambda}P_6 + \frac{3\mu_e}{6\mu+\lambda}P_{12}$$

$$P_6 = \frac{6\mu}{6\mu+\lambda}P_7 + \frac{3\mu_e}{6\mu+\lambda}P_{13}$$

$$P_7 = \frac{9\mu}{6\mu+\lambda}P_8 + \frac{3\mu_e}{6\mu+\lambda}P_{14}$$

$$P_8 = \frac{9\mu}{9\mu+\lambda}P_9 + \frac{3\mu_e}{9\mu+\lambda}P_{15}$$

$$P_9 = \frac{9\mu}{9\mu+\lambda}P_{10} + \frac{3\mu_e}{9\mu+\lambda}P_{16} \qquad (7.11)$$

$$P_{10} = \frac{3\mu_e}{9\mu+\lambda}P_{17}$$

$$P_{11} = \frac{3\mu}{3\mu_e+\lambda}P_{12} + \frac{3\mu_e}{3\mu_e+\lambda}P_{18}$$

$$P_{12} = \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{13} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{19}$$

$$P_{13} = \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{14} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{20}$$

$$P_{14} = \frac{6\mu}{3\mu+3\mu_e+\lambda}P_{15} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{21}$$

$$P_{15} = \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{16} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{22}$$

$$P_{16} = \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{17} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{23}$$

(continued . . .)

$$P_{17} = \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{24}$$

$$P_{18} = \frac{3\mu}{3\mu_e+\lambda}P_{19} + \frac{3\mu_e}{3\mu_e+\lambda}P_{25}$$

$$P_{19} = \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{20} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{26}$$

$$P_{20} = \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{21} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{27}$$

$$P_{21} = \frac{6\mu}{3\mu+3\mu_e+\lambda}P_{22} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{28}$$

$$P_{22} = \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{23} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{29}$$

$$P_{23} = \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{24} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{30}$$

$$P_{24} = \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{31}$$

$$P_{25} = \frac{\lambda}{3\mu_e+\lambda}P_1 + \frac{3\mu}{3\mu_e+\lambda}P_{26} \qquad (7.11)$$

$$P_{26} = \frac{\lambda}{3\mu+3\mu_e+\lambda}P_2 + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{27}$$

$$P_{27} = \frac{\lambda}{3\mu+3\mu_e+\lambda}P_3 + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{28}$$

$$P_{28} = \frac{\lambda}{3\mu+3\mu_e+\lambda}P_4 + \frac{6\mu}{3\mu+3\mu_e+\lambda}P_{29} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{35}$$

$$P_{29} = \frac{\lambda}{6\mu+3\mu_e+\lambda}P_5 + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{30} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{36}$$

$$P_{30} = \frac{\lambda}{6\mu+3\mu_e+\lambda}P_6 + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{31} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{37}$$

$$P_{31} = \frac{\lambda}{6\mu+3\mu_e+\lambda}P_7 + \frac{9\mu}{6\mu+3\mu_e+\lambda}P_{32} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{38}$$

$$P_{32} = \frac{\lambda}{9\mu+\lambda}P_8 + \frac{9\mu}{9\mu+\lambda}P_{33} + \frac{3\mu_e}{9\mu+\lambda}P_{39}$$

(continued. . .)

$$
\begin{aligned}
P_{33} &= \frac{\lambda}{9\mu+\lambda}P_9 + \frac{9\mu}{9\mu+\lambda}P_{34} + \frac{3\mu_e}{9\mu+\lambda}P_{40} \\
P_{34} &= \frac{\lambda}{9\mu+\lambda}P_{10} + \frac{3\mu_e}{9\mu+\lambda}P_{41} \\
P_{35} &= \frac{\lambda}{3\mu_e+\lambda}P_{11} + \frac{3\mu}{3\mu_e+\lambda}P_{36} + \frac{3\mu_e}{3\mu_e+\lambda}P_{42} \\
P_{36} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{12} + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{37} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{43} \\
P_{37} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{13} + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{38} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{44} \\
P_{38} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{14} + \frac{6\mu}{3\mu+3\mu_e+\lambda}P_{39} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{45} \\
P_{39} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{15} + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{40} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{46} \\
P_{40} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{16} + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{41} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{47} \\
P_{41} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{17} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{48} \\
P_{42} &= \frac{\lambda}{3\mu_e+\lambda}P_{18} + \frac{3\mu}{3\mu_e+\lambda}P_{43} + \frac{3\mu_e}{3\mu_e+\lambda}P_{49} \\
P_{43} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{19} + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{44} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{50} \\
P_{44} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{20} + \frac{3\mu}{3\mu+3\mu_e+\lambda}P_{45} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{51} \\
P_{45} &= \frac{\lambda}{3\mu+3\mu_e+\lambda}P_{21} + \frac{6\mu}{3\mu+3\mu_e+\lambda}P_{46} + \frac{3\mu_e}{3\mu+3\mu_e+\lambda}P_{52} \\
P_{46} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{22} + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{47} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{53} \\
P_{47} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{23} + \frac{6\mu}{6\mu+3\mu_e+\lambda}P_{48} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{54} \\
P_{48} &= \frac{\lambda}{6\mu+3\mu_e+\lambda}P_{24} + \frac{3\mu_e}{6\mu+3\mu_e+\lambda}P_{55}
\end{aligned}
\tag{7.11}
$$

$$P_{n+11} = \frac{\lambda}{3\mu_e} P_{n-13} + \frac{\mu}{\mu_e} P_{n+12} + P_{n+18}$$

$$P_{n+12} = \frac{\lambda}{3\mu+3\mu_e} P_{n-12} + \frac{\mu}{\mu+\mu_e} P_{n+13} + \frac{\mu_e}{\mu+\mu_e} P_{n+19}$$

$$P_{n+13} = \frac{\lambda}{3\mu+3\mu_e} P_{n-11} + \frac{\mu}{\mu+\mu_e} P_{n+14} + \frac{\mu_e}{\mu+\mu_e} P_{n+20}$$

$$P_{n+14} = \frac{\lambda}{3\mu+3\mu_e} P_{n-10} + \frac{2\mu}{\mu+\mu_e} P_{n+15} + \frac{\mu_e}{\mu+\mu_e} P_{n+21}$$

$$P_{n+15} = \frac{\lambda}{6\mu+3\mu_e} P_{n-9} + \frac{2\mu}{2\mu+\mu_e} P_{n+16} + \frac{\mu_e}{2\mu+\mu_e} P_{n+22}$$

$$P_{n+16} = \frac{\lambda}{6\mu+3\mu_e} P_{n-8} + \frac{2\mu}{2\mu+\mu_e} P_{n+17} + \frac{\mu}{2\mu+\mu_e} P_{n+23}$$

$$P_{n+17} = \frac{\lambda}{6\mu+3\mu_e} P_{n-7} + \frac{\mu_e}{2\mu+\mu_e} P_{n+24}$$

$$P_{n+18} = \frac{\lambda}{3\mu_e} P_{n-6} + \frac{\mu}{\mu_e} P_{n+19} + P_{n+25} \qquad (7.12)$$

$$P_{n+19} = \frac{\lambda}{3\mu+3\mu_e} P_{n-5} + \frac{\mu}{\mu+\mu_e} P_{n+20} + \frac{\mu_e}{\mu+\mu_e} P_{n+26}$$

$$P_{n+20} = \frac{\lambda}{3\mu+3\mu_e} P_{n-4} + \frac{\mu}{\mu+\mu_e} P_{n+21} + \frac{\mu_e}{\mu+\mu_e} P_{n+27}$$

$$P_{n+21} = \frac{\lambda}{3\mu+3\mu_e} P_{n-3} + \frac{2\mu}{\mu+\mu_e} P_{n+22} + \frac{\mu_e}{\mu+\mu_e} P_{n+28}$$

$$P_{n+22} = \frac{\lambda}{6\mu+3\mu_e} P_{n-2} + \frac{2\mu}{2\mu+\mu_e} P_{n+23} + \frac{\mu_e}{2\mu+\mu_e} P_{n+29}$$

(continued . . .)

$$P_{n+23} = \frac{\lambda}{6\mu+3\mu_e}P_{n-1} + \frac{2\mu}{2\mu+\mu_e}P_{n+24} + \frac{\mu_e}{2\mu+\mu_e}P_{n+30}$$

$$P_{n+24} = \frac{\lambda}{6\mu+3\mu_e}P_{n} + \frac{\mu_e}{2\mu+\mu_e}P_{n+31}$$

$$P_{n+25} = \frac{\lambda}{3\mu_e}P_{n+1} + \frac{\mu}{\mu_e}P_{n+26}$$

$$P_{n+26} = \frac{\lambda}{3\mu+3\mu_e}P_{n+2} + \frac{\mu}{\mu+\mu_e}P_{n+27}$$

$$P_{n+27} = \frac{\lambda}{3\mu+3\mu_e}P_{n+3} + \frac{\mu}{\mu+\mu_e}P_{n+28}$$

$$P_{n+28} = \frac{\lambda}{3\mu+3\mu_e}P_{n+4} + \frac{2\mu}{\mu+\mu_e}P_{n+29}$$

$$P_{n+29} = \frac{\lambda}{6\mu+3\mu_e}P_{n+5} + \frac{2\mu}{2\mu+\mu_e}P_{n+30} \qquad (7.12)$$

$$P_{n+30} = \frac{\lambda}{6\mu+3\mu_e}P_{n+6} + \frac{2\mu}{2\mu+\mu_e}P_{n+31}$$

$$P_{n+31} = \frac{\lambda}{6\mu+3\mu_e}P_{n+7} + \frac{3\mu}{2\mu+\mu_e}P_{n+32}$$

$$P_{n+32} = \frac{\lambda}{9\mu}P_{n+8} + P_{n+33}$$

$$P_{n+33} = \frac{\lambda}{9\mu}P_{n+9} + P_{n+34}$$

$$P_{n+34} = \frac{\lambda}{9\mu}P_{n+10}$$

per cent. The range of applicability of the model can be increased by increasing the number of states. However beyond 70 per cent load the number of states required to maintain the same accuracy increases rapidly. As a result the matrix that must be inverted to obtain the numerical solution grows even faster (as a square of the number of states). Given enough time and resources this can be accomplished. On the other hand it may not be necessary to do so if the theoretical and the experimental results show good agreement for the range 0-70 per cent.

Figure 7.20 compares the normalized mean delay as obtained from this model to the experimental results as well as results from the exponential distribution model. For R=1, 10 and 50 the agreement is excellent considering the numerous sources of error such as

1. Not all system states are represented in the model.

2. After the markov model has been solved for steady state probabilities, more approximations are made to obtain the job start delay statistics.

Figure 7.21 shows the 80, 99 and 99.9 percentile delays as functions of the system load for R=1. The sharp initial slope and near linearity of the 99.9 percentile delay is evident from this figure. For the other two percentiles also, the results are in close agreement for loads approaching seventy per cent.

The next three figures compare the frequency distribution of delay as obtained from the experiments, the exponential model and the Erlang model. The cases of R=1 (load factors 25 and 75 per cent) and R=10 (load factor 50 per cent) are shown. In all the three cases the experimental and the Erlang model results are quite close, showing good agreement.

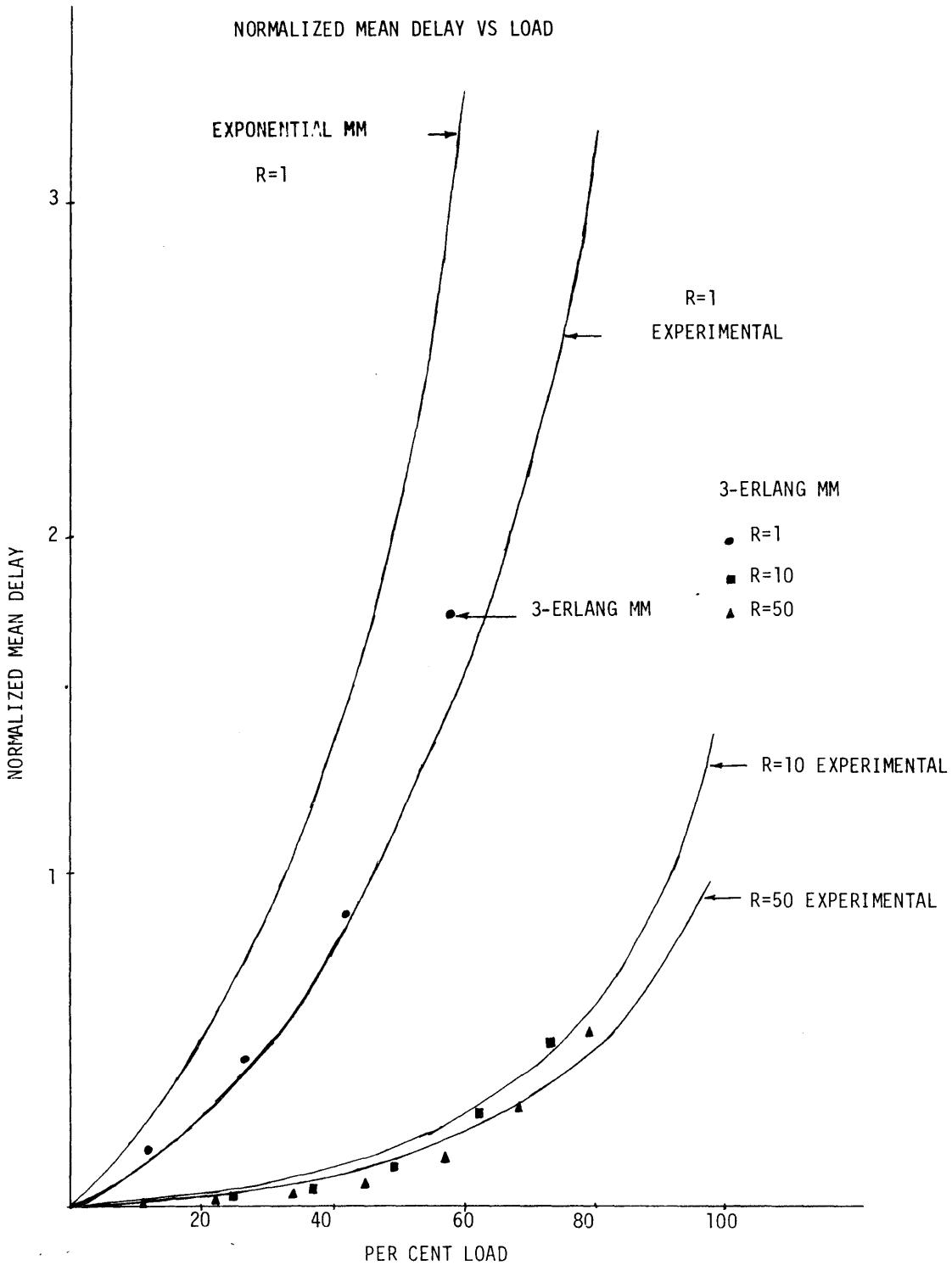

NORMALIZED MEAN DELAY VS LOAD
EXPONENTIAL MM
R=1
R=1
EXPERIMENTAL
3-ERLANG MM
3-ERLANG MM
R=1
R=10
R=50
R=10 EXPERIMENTAL
R=50 EXPERIMENTAL
NORMALIZED MEAN DELAY
3
2
1
20
40
60
80
100
PER CENT LOAD

Figure 7.20

ig. 7.21

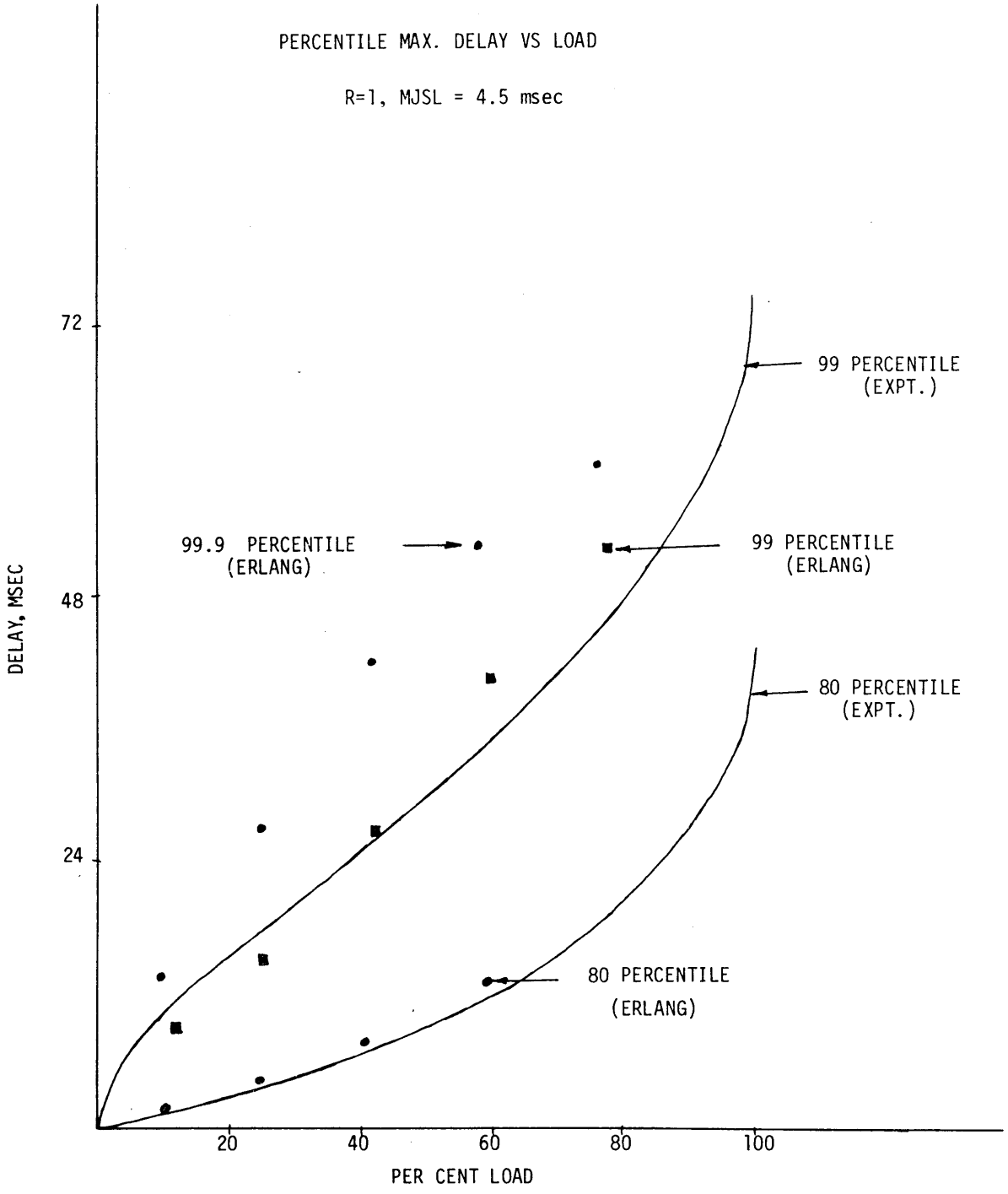


Figure 7.21

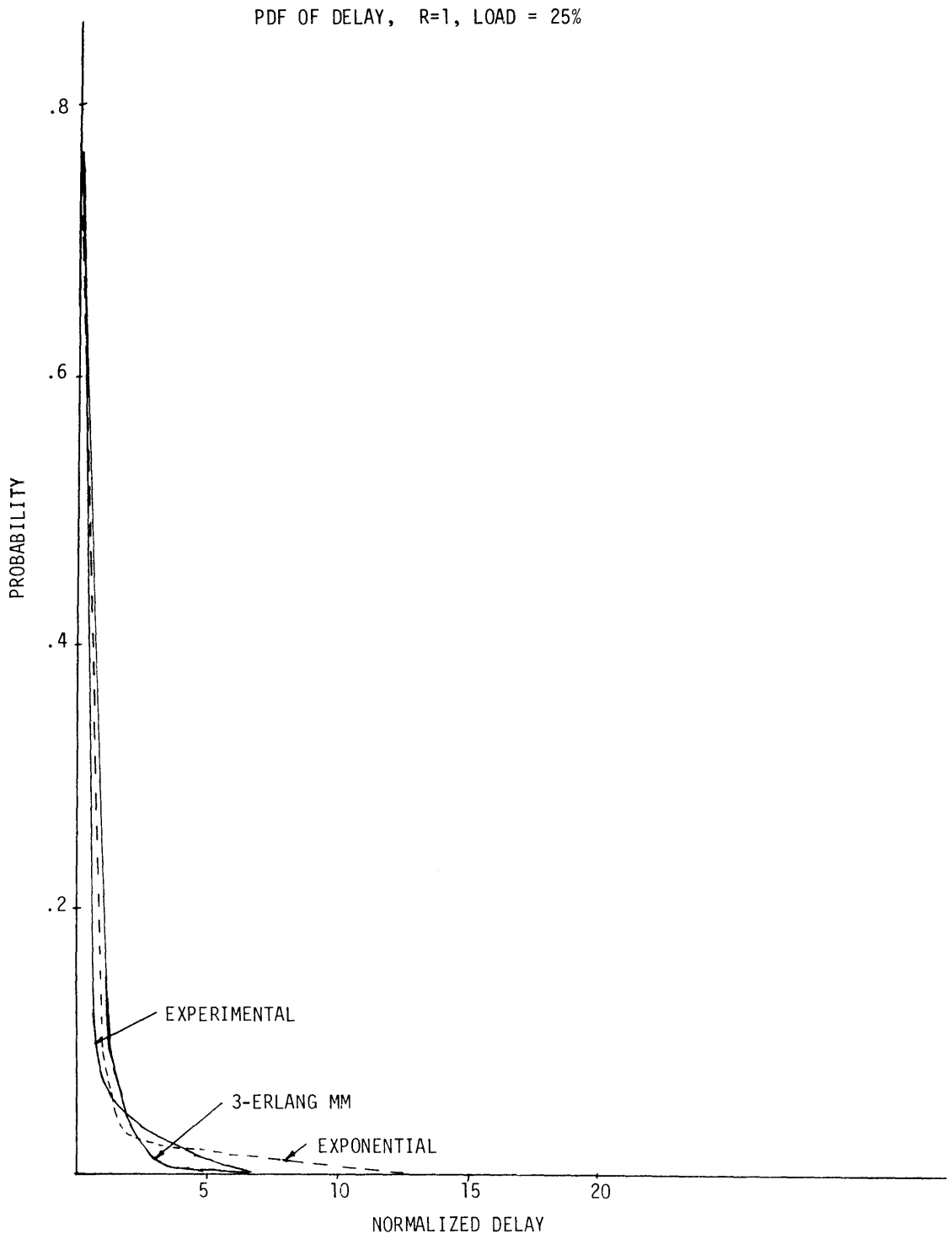

PDF OF DELAY, R=1, LOAD = 25%
.8
.6
.4
.2
PROBABILITY
EXPERIMENTAL
3-ERLANG MM
EXPONENTIAL
5
10
15
20
NORMALIZED DELAY

Figure 7.22

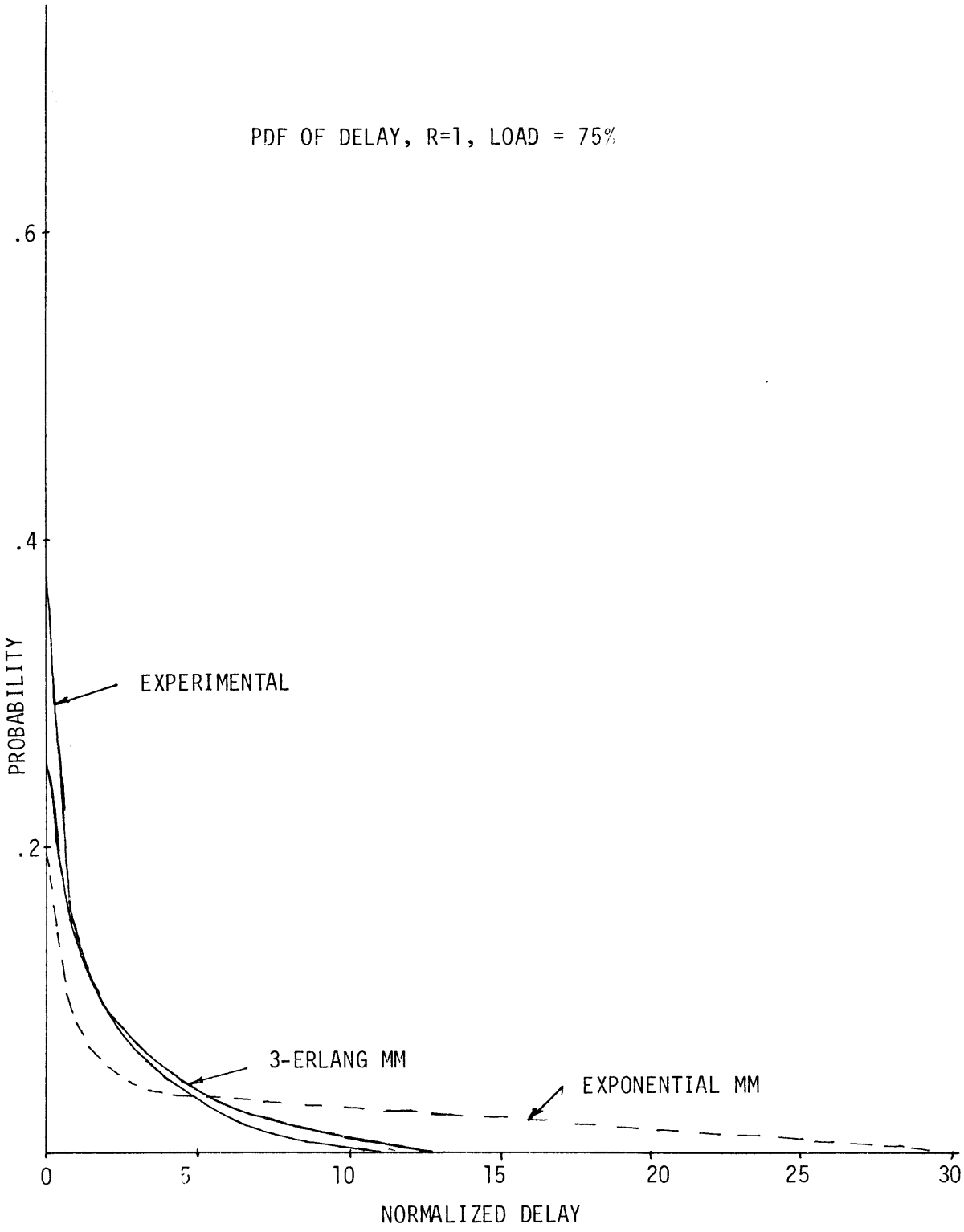

PDF OF DELAY, R=1, LOAD = 75%
.6
.4
.2
PROBABILITY
EXPERIMENTAL
3-ERLANG MM
EXPONENTIAL MM
0
5
10
15
20
25
30
NORMALIZED DELAY

Figure 7.23

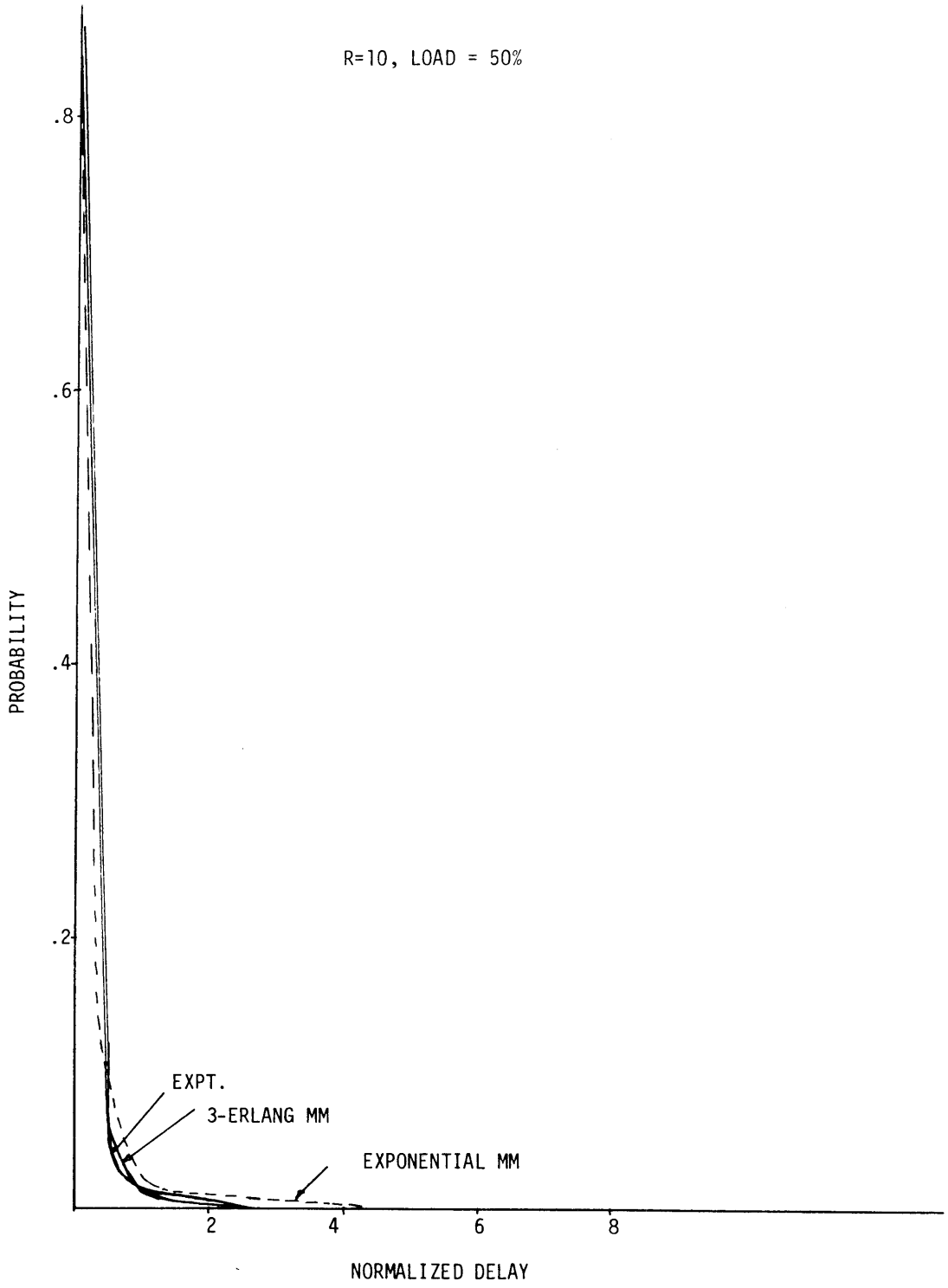

R=10, LOAD = 50%
.8
.6
.4
.2
PROBABILITY
EXPT.
3-ERLANG MM
EXPONENTIAL MM
2
4
6
8
NORMALIZED DELAY

Figure 7.24

7.6 Summary

To summarize the work reported in this chapter, three Markov models of increasing complexity were developed and solved numerically. The results concerning the processor occupancy times and job start delays were derived and compared to the experimentally found results. The 3-Erlang Markov model was found to be in close agreement with the experiments.

CHAPTER VIII

CONCLUSION

8.1 Introduction

The objective of this thesis has been to analyze the performance of a multiprocessor computer system in a real time environment. This objective was established as a result of the growing number of real time situations where multiprocessors are being used or will be put to use in the future. The importance of the area of performance evaluation research is attested to by the large number of papers published in this field [47]. Even though the general area of performance evaluation is receiving a great deal of attention, the specific field chosen for this thesis has been explored only superficially. During the course of this work, more results have been published regarding the behavior of multiprocessor computer systems [43,44,45]. However none of these addresses real time applications. Therefore their performance criteria are different from the ones developed here. In particular, the usual emphasis is on system throughput, while in this thesis job response time has been used as the major performance criterion. It is interesing to note, though, that the mathematical tools used by some of the authors [43,44] are the same as those used in this thesis viz., the Markov chain techniques. This will be discussed further in the next section.

To accomplish the objective of evaluating the performance of a multiprocessor, a three-processor system was instrumented to facilitate performance monitoring through hardware. Software was developed to

simulate a parameterized real-time workload through a number of repetitive pseudo-jobs, also to help monitor system performance, and finally to perform data reduction of the results. An Executive program was written to dispatch jobs, to accept job requests and to perform other operating system functions. Two different strategies were used to dispatch jobs. The two sets of results obtained with the two strategies were analyzed and compared.

Theoretical models were developed to validate the results of the experiments. These models range from a simple analytical model with a closed form solution to a complex Markov chain model with more than 200 states, which was solved numerically on a digital computer. The following section summarizes the major results and their significance.

8.2 Results and Their Significance

The results of the performance evaluation experiments have been described in detail in Chapter six. Chapter seven contains the results obtained from the mathematical models of the system under investigation. A comparison of the two is also contained in the same chapter.

The first set of results concerns the throughput efficiency as a function of R, the ratio of the average job length to the executive length. When the job length and the executive length are comparable (R=1), the multiprocessor throughput efficiency is approximately forty per cent of its value for R=10, (i.e. when the executive length is an order of magnitude shorter than the job length). Further shortening of the Executive produces diminishing returns. A value of about ten for R may then be considered to have an acceptably small throughput penalty.

The efficiency of the executive program becomes an important issue when job step lengths are shortened to a point comparable to the executive

length. First of all, the executive program itself consumes a large proportion of the available resources. Second, the impact of the executive's lockout of other processors waiting for a new job step is significant. With three processors, executive lockout accounts for about 27 per cent of all processor time at full load for R=1. This would be even higher if there were more processors in the system. When job steps are longer, executive lockout time is less of an issue. When R=10, it drops to three per cent at full load, and when R=50 it is down to one half of one per cent.

The second set of results concerns the average delay encountered by a job before starting. The mean job starting delay increases without bound as the load approaches saturation, in agreement with Rosenberg [35] and Mallach [28]. The normalized mean delay, that is, the ratio of absolute mean delay to the job step length, decreases as the ratio of the mean job step length to executive running time is increased. This is due to the fact that the higher the ratio R, the lower is the probability of executive access conflicts between processors. Once again, as in the case of throughput efficiency the law of diminishing returns applies strongly beyond R=10.

The third set of results concerns the maximum job starting delay. In principle, the probability of any delay, no matter how large, occuring in practice is non-zero. Therefore it would be futile to talk of maximum delay as such. However it is still meaningful to define various percentiles of delay. For example, if the 99 percentile delay is x then 99 per cent of all delays are less than or equal to x. For the 80 percentile function (R=1) the delay doubles as the load is increased from 50 to 75 per cent (see Fig. 6.15). The same increase in the load factor results only in a 50 per cent increase in the 99 percentile delay. Going still further, the 99.99 percentile function shows an initial sharp increase and becomes nearly linear thereafter and remains linear for loads approaching 90 per cent [46]. The important consequence of this result is that the penalty to be paid in going from a low to a high load factor operating point is not as great as the mean delay function implies. In other words, cost incurred in terms of worst case delays increases only linearly with system load.

The fourth set of results concerns the frequency distribution of the job starting delay. For low to moderately high load factors this important function is found to be a monotonically decreasing delay function not unlike a hyperbolic curve. The initial decline is sharp followed by an almost horizontal long tail. This type of probability distribution function implies that a large percentage of jobs start without an appreciable delay. Also a monotonically decreasing function implies that the probability of a job starting without a delay is higher than with any other value of delay. For loads in excess of ninety per cent the distribution function increases slightly before gradually decreasing with an asymptotic tail.

The fifth set of results concerns the comparison of two job dispatch strategies viz., First-Come-First-Serve (FCFS) and Shortest-Job-First (SJF). The probability distribution function of the job start delay for SJF starts at a higher value but extends to larger delays. This indicates that more jobs start on time or with short delays with the SJF dispatch strategy. However this is done at the expense of longer jobs thereby increasing the probability of longer delays. The net result is that the range of delays is increased considerably while the mean delay is reduced only slightly. This gain is apt to be lost with a higher number of processors. The SJF strategy also increases the overheads by two to three per cent. Therefore, even though the SJF may work better for a single processor computer the simpler FCFS appears to be preferable for a multiprocessor.

The validation of the experimental results was carried out through Markov process methods. The frequent use of Markov chain techniques by other workers in the field of multiprocessor performance evaluation [43,44] attests to the conceptual simplicity of modeling, ease of numerical solution and several other advantages offered by this technique. On the other hand, the most obvious limitation is that most often not all the system states can be represented in the model without outgrowing the biggest digital computers at hand. A judicious choice of states is therefore an essential element of the success of Markov models. A truncation of states signifies

a truncation of accuracy. Therefore it is most important to use as many states as possible without jeopardizing the efficiency of the process of numerical solution of the model. In some simple cases it is even possible to obtain a closed form analytical solution of the model. This is precisely what was done to model the limiting case of $R = \infty$ (zero executive length). An expression for the normalized mean delay for the three processor system was obtained. This was close to the experimentally found mean delay function for R=50. This model was expanded to include the system states corresponding to running of the Executive. The run times of the Executive and jobs were assumed to be exponentially distributed random variables. Hence this model was called the 'Exponential Distribution Markov Model'. The exponential model with 120 states was numerically solved by the Gauss-Jordan reduction method. The results regarding throughput efficiency, average delay, maximum delay and frequency functions of delay showed striking similarity to those obtained from the experiments. However there were sufficient numerical discrepancies to warrant a further refinement of the model. An investigation led to the difference in assumed and actual frequency functions of job and executive lengths as the possible source of error. An Erlang model was developed to rectify this defect. The solution of the Erlang model, which had six times more states than the exponential model, showed an excellent agreement with the experimental results.

8.3 Perspective Summary

In perspective, the results discovered during the course of this thesis are seen to fill an important gap in the knowledge regarding the behavior of computer systems.

The ratio of the average job length to the executive length should be at least of the order of ten to make the throughput penalty acceptably small. For lower values of the ratio the executive program and the executive lockout

together account for a large percentage of the processor time. In such cases a possible solution to the throughput problem is to split the Executive into two or more segments that may be executed concurrently and hence reduce the lockout time. This ratio also has a strong impact on the mean job starting delay. The higher the ratio the lower is the average delay. However, irrespective of the value of the ratio the percentile maximum delays are found to have an important characteristic viz., the near linearity with respect to the system load in the range of low to moderately high values of load factor. As a result it is possible to operate a real time system at a high load factor without penalizing the system performance with inordinately long worst case delays. Even at high load factors a large percentage of jobs start on time or with short delays as indicated by the hyperbolic nature of the probability distribution function of the delay. Regarding the job dispatch strategies, the First-Come-First-Serve schedule appears to be more suitable for multiprocessors than the Shortest-Job-First schedule which reduces the average job start delay slightly but significantly widens the delay spread. Finally, the Markov process method is found to be an excellent mathematical tool for analyzing the multiprocessor system performance with high fidelity.

8.4 Areas for Further Research

There are many areas where further investigation and research hold promise. First of all, microprocessor based multiprocessors are proliferating [47]. These tend to have a large number of processors interacting with each other. It would be useful to extend the results of this thesis to such a case. This can be most easily done by expanding the Markov models of Chapter seven to account for a greater number of processors. Second, even though the SJF strategy proved to be of marginal utility with respect to multiprocessor system performance, it might be possible to work on a different job dispatch scheme that makes the probability distribution function of delay more favorable for a certain class of jobs,

even for a large number of processors. Third, changes in the Executive such as concurrent execution of different segments of the executive program could have significant impact on system throughput and job start delays. Finally, there are limitless numbers of memory access patterns and job step length distributions for which it would be important to evaluate the system performance.

In conclusion, important results concerning multiprocessor performance with respect to real time environment were obtained experimentally and verified theoretically that should have a significant bearing on multiprocessor design.

BIBLIOGRAPHY

1. Alarcon Cuello, F.A., "A Monitor/Loader for an Experimental Multiprocessor", S.B. Thesis, M.I.T. Department of Electrical Engg., Sept. 1971.

2. Amiot, L., et al, "Evaluating a Remote Batch Processing System", Computer, Vol. 5, No. 5, Sept./Oct. 1972, pp. 12-47.

3. Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, Vol. 5, No. 1, March 1973.

4. Bhandarkar, D.P., "Analytical Models for Memory Interference in Multiprocessor Computer Systems", Ph.D. Thesis, Carnegie-Mellon University, 1973.

5. Calingaert, P., "System Performance Evaluation: Survey and Appraisal", Communications of the ACM, Vol. 10, No. 1, Jan. 1967.

6. Chang, W., et al, "Throughput Analysis of Computer Systems - Multiprogramming versus Multiprocessing", Proceedings of ACM Workshop on System Performance Evaluation, Harvard University, April 1971.

7. Chow, W., "A Memory Interference Model for Multiprocessing Systems", IBM RC - Report No. 4867, June 1974.

8. Colson, C.F., "Input-Output Data Handling in a Hierarchical Digital Control System", S.M. Thesis, Dept. Of Aero. and Astro., M.I.T., Also available as C.S. Draper Lab. Report T-611, Feb. 1975.

9. Computer, Special Issue on Performance Measurement, Vol. 5, No. 4, July/August 1972, pp. 30-53.

10. Computer, Special Issue on Performance Evaluation, Vol. 5, No. 5, Sept./Oct. 1972, pp. 12-47.

11. Covo, A.A., "Analysis of Multiprocessor Control Organization with Partial Program Memory Replication", IEEE Transactions on Computers, Vol. C-23, No. 2, Feb. 1974.

12. Daly, W.M., A.L. Hopkins, and J.F. McKenna, "A Fault-Tolerant Digital Clocking System", 1973 International Symposium on Fault-Tolerant Computing, Palo Alto, Cal., June 1973.

13. Daly, W.M., "CERBERUS Information Memorandum", Digital Development Memo. No. 624, C.S. Draper Lab., Cambridge, Mass., 23 July 1971.

14. Diruf, Gunther, " A Model Structure for the Optimization of Multiprocessing Systems" in "Operational Research in the Design of Electronic Data Processing Systems", Proc. of a NATO Conference in Munich, 1971, Crane, Russak and Co., Inc., New York, 1973.

15. Green, A.I., et al, "STS Data Management System Design", Report E-2529, C.S. Draper Lab., Cambridge, Mass., June 1970.

16. Hansmann, F., Editor, "Operational Research in the Design of Electronic Data Processing Systems", Proceedings of a NATO Conference in Munich, Germany, 1971, Crane, Russak and Co., Inc., New York, 1973, pp. 251-252

17. Hatfield, D.J., "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance", IBM Journal of Research and Development, Vol. 16, No. 1, Jan. 1972.

18. Holland, F.C., et al, "Simulation of a Multiprocessing System using GPSS", IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-4, No. 4, Nov. 1968.

19. Hopkins, A.L., Jr., "A Fault-Tolerant Information Processing Concept for Space Vehicles", IEEE Transactions on Computers, Vol. C-20, No. 11, Nov. 1971.

20. Hopkins, A.L., Jr., and T.B. Smith, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor", Digest of the Fourth International Symposium on Fault-Tolerant Computing", University of Illinois, June 1974.

21. King, W.F. III, et al, "Effects of Serial Programs in Multiprocessing Systems", IBM Journal of Research and Development, July 1974.

22. Kurtzberg, J.M., "On the Memory Conflict Problem in Multiprocessor Systems", IEEE Transactions on Computers, Vol. C-23, No. 3, March 1974.

23. Lala, J.H., "CERBERUS Assembly Language", Digital Development Memo. No. 795 Rev. A, C.S. Draper Lab., Cambridge, Mass., 16 Oct. 1974.

24. Lampson, B.W., "A Scheduling Philosophy for Multiprocessing Systems", Communications of the ACM, Vol. 11, No. 5, May 1968.

25. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", Proc. of IEEE, Vol. 54, No. 12, Dec. 1966.

26. Lucas, H.C., Jr., "Performance Evaluation and Monitoring", Computing Surveys, Vol. 3, No. 3, Sept. 1971.

27. Madnick, S.E., "Multiprocessor Software Lockout", Proc. of the 1968 ACM National Conference.

28. Mallach, E.G., "Analysis of a Multiprocessor Guidance Computer", Ph.D. Thesis, Dept. of Aero. and Astro., M.I.T., June 1969, Also available as C.S. Draper Lab. Report T-515.

29. Mallach, E.G., "Job-Mix Modeling and System Analysis of an Aerospace Multiprocessor", IEEE Transactions on Computers, Vol. C-21, No. 5, May 1972.

30. McKenna, J.F., "CERBERUS Program Plan and Description", CERBERUS Memo. No. 1, Digital Development Group, C.S. Draper Lab., Cambridge, Mass., 3 Feb. 1971.

31. McKenna, J.F., "CERBERUS Instruction Set", Digital Development Internal Report, C.S. Draper Lab., Cambridge, Mass., 4 Nov. 1974.

32. Mitchell, J., et al, "Multiprocessor Performance Analysis", National Computer Conference, 1974, pp. 399-403.

33. Pessel, D., "Primary Memory Contention in a Multiprocessor Multi-memory Computer System Executing Related Tasks", Ph.D. Thesis, Dept. of Electrical Engg. and Computer Sciences, University of California, Berkeley, 1974.

34. Ramamoorthy, C.V., et al, "Optimal Scheduling Strategies in a Multiprocessor System", IEEE Transactions on Computers, Vol. C-21, No. 2, Feb. 1972.

35. Rosenberg, S.C., "An Executive Program for an Aerospace Multiprocessor", S.M. Thesis, Dept. of Electrical Engg., M.I.T., Sept. 1971, Also available as C.S. Draper Lab. Report T-552.

36. Smith, T.B., "A Damage-And-Fault-Tolerant Input/Output Network", Digest of the Fourth International Symposium on Fault-Tolerant Computing, University of Illinois, June 1974.

37. Smith, T.B., "A Highly Modular Fault-Tolerant Computer System", Ph.D. Thesis, Dept. of Aero. and Astro., M.I.T., Also available as Report T-595, C.S. Draper Lab., Cambridge, Mass., Nov. 1973.

38. Stanley, W.I., "Measurement of System Operational Statistics", IBM Systems Journal 8, No. 4, 1969.

39. Swanson, A.K., "Objectives of Simulation", IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-4, No. 4, Nov. 1968.

40. Tetzlaff, W.H., et al, "SGP: An Effective Use of Performance and Usage Data", Computer, Vol. 5, No. 5, Sept./Oct. 1972.

41. Cox, D.R., and W.L. Smith, "Queues", Methuen and Co., London, 1961.

42. Drake, A.W., "Fundamentals of Apllied Probability Theory", McGraw-Hill Book Co., New York, 1967, pp. 188-191.

43. Bhandarkar,D.P.,"Analysis of Memory Interference in Multiprocessors", IEEE Transactions on Computers, Vol. C-24, No. 9, Sept.1975.

44. Sastry, K.V., et al,"On the performance of Certain Multiprocessor Computer Organizations", IEEE Ttansactions on Computers, Vol. C-24, No. 11, November 1975.

45. Fernarndez, E.B., and T. Lang,"Computation of Lower Bounds for Multiprocessor Schedules", IBM Journal of Research and Development, Vol. 19, No. 5, September 1975.

46. Colson, C.F., A.L. Hopkins, and J.H. Lala,"Program and Input Output Management for a Distributed Fault-Tolerant Digital System", 5th International Symposium on Fault-Tolerant Computing, Paris, France, June 1975.

47. Agajanian, A.H.,"A Bibliography on System Performance Evaluation", IEEE Computer, November 1975, pp.63-74.

48. IEEE Spectrum,"Large Capacity Computer Development", December 1975, p. 75.

BIOGRAPHICAL NOTE

Jaynarayan H. Lala was born in Hydrabad, Pakistan on January 12, 1951. He attended the Mission Higher Secondary School in Seoni, India graduating in 1966. Based on an All India Competitive Examination he was selected a National Science Talent Scholar in 1966.

He entered the Indian Institute of Technology, Bombay in 1966 and graduated with honors in 1971 obtaining a B. Tech. degree in Aeronautical Engineering. While at I.I.T., he held a Government of India Merit Scholarship for the years 1966-70. Also, he was an Institute Merit Scholar for the Academic year 1970-71.

Continuing his education at the Massachusetts Institute of Technology he obtained an S.M. degree in Aeronautics and Astronautics in February, 1973. At M.I.T., he was elected to Sigma Gamma Tau and Sigma Xi.

CONFERENCE PAPERS

1. Lala J.H., "A Cost and Reliability Model of a Partitioned Digital System", 1973 International Symposium on Fault-Tolerant Computing, Palo Alto, Ca., June 1973.

2. Colson C.F., A.L. Hopkins and J.H. Lala, "Input Output and Program Management for a Distributed Fault-Tolerant Digital System", 1975 International Symposium on Fault-Tolerant Computing, Paris, France, June 1975.