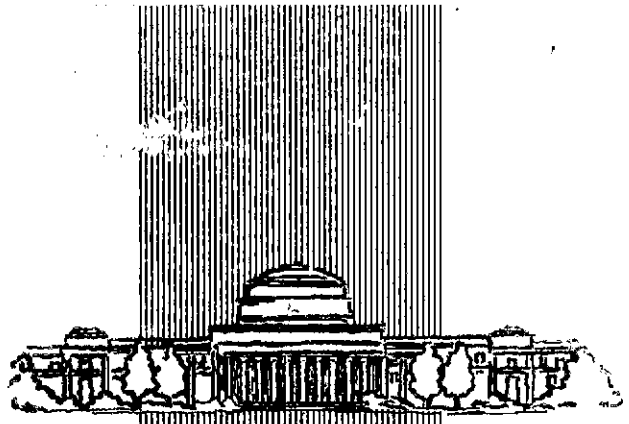


MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ANALYSIS OF A MULTIPROCESSOR GUIDANCE COMPUTER

by

Efrem G. Mallach

June 1969

Degree of Doctor of Philosophy

T-515

PREPARED AT

INSTRUMENTATION LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE 39, MASSACHUSETTS

T-515

ANALYSIS OF A MULTIPROCESSOR GUIDANCE COMPUTER

by

Efrem G. Mallach

B. S. E., Princeton University (1964)

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Doctor of Philosophy

at the

Massachusetts Institute of Technology
June, 1969

Signature of Author: Efrem G. Mallach
Department of Aeronautics and Astronautics,
May 5, 1969

Certified by: Wallace E. Vander Velde
Thesis Supervisor

James E. Potter
Thesis Supervisor

Daniel L. Slotnick
Thesis Supervisor

Accepted by: Andrew F. Bennett
Chairman, Departmental Committee on Graduate Students

ANALYSIS OF A MULTIPROCESSOR COMPUTER

by

Efrem G. Mallach

Submitted to the Department of Aeronautics and Astronautics on May 5, 1969 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This thesis is concerned with the design of the next generation of spaceborne digital computers. It analyzes a possible multiprocessor computer configuration.

Such a computer would be composed of a number of processors and memory units connected by a central data bus. A "job stack" would store information describing tasks to be performed, and would issue job requests when they come due. Any processor may accept any task. It would obtain needed data from memory, carry out the calculations, and return results when done. All data and messages would be transmitted via the bus. Such a system has advantages of expandability, suitability for sampled-data calculations, and "graceful degradation" characteristics.

For the analysis, a set of representative space computing tasks was abstracted from the Lunar Module Guidance Computer programs as executed during the lunar landing, from the Apollo program. This computer performs at this time about 24 concurrent functions, with iteration rates from 10 times per second to once every two seconds. These jobs were tabulated in a machine-independent form, and statistics of the overall job set were obtained.

A simulation of the multiprocessor was then developed. This simula-

tion permits the running of the set of jobs mentioned on page iii (or any other set) on the multiprocessor, and provides statistical and other information as to multiprocessor performance during a run. The multiprocessor configuration, speed, etc., can be varied at will.

The multiprocessor was then analyzed using queuing theory. A simple model was solved analytically. More complex models, modeling the system with greater accuracy, were also developed and were solved numerically with a Markov process approach. These models permit specification of a multiprocessor and its job mix in terms of very few parameters, which makes for simple evaluation of the multiprocessor under widely varying conditions. The Markov model that was decided upon as providing minimum acceptable accuracy has 191 states.

It was concluded, based on a comparison of simulation and Markov results, that the Markov process analysis is accurate in predicting overall trends and in configuration comparisons, but does not provide useful detailed information in specific situations. Using both types of analysis, it was determined that the job scheduling function is a critical one for efficiency of the multiprocessor. It is recommended that research into the area of automatic job scheduling be performed.

Furthermore it was found that a multiprocessor with many slow processors is more efficient than one with a few fast processors. To utilize such a system, long jobs must be broken down into groups of short jobs. It is recommended that research into means of performing such a breakdown automatically be conducted.

It was finally concluded that the algorithm by which system components are given access to the bus in case of simultaneous demands is of little importance to system performance.

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

ACKNOWLEDGMENT

The author would like to acknowledge his appreciation to his thesis committee: to Prof. Wallace E. Vander Velde, whose perceptive and constructive criticism was invaluable throughout the course of the work; to Dr. Ramon L. Alonso, who first suggested the research topic and provided steady guidance throughout its development; and to Prof. James E. Potter, who provided considerable assistance, especially in the area of Chapter 4. Sincere thanks are also due to many individuals at the M.I.T. Instrumentation Laboratory, particularly to Dr. Albert L. Hopkins. The Technical Publications Group of the Instrumentation Laboratory was extremely helpful at every step from rough draft to final copy. Finally, the author would like to express publicly what he has on many occasions expressed privately: that without the patient support and encouragement of his wife Linda, this thesis could not have been completed.

This report was prepared under DSR Project 55-38640 sponsored by the Manned Spacecraft Center of the National Aeronautics and Space Administration through Contract NAS 9-4065 with the Instrumentation Laboratory of Massachusetts Institute of Technology in Cambridge, Massachusetts.

The publication of this report does not constitute approval by the Instrumentation Laboratory or the National Aeronautics and Space Administration of the findings or conclusions contained therein. It is published only for the exchange and stimulation of ideas.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 The Aerospace Computing Task

The use of digital computers in manned spacecraft has gained general acceptance in recent years. The multitude of control, guidance and navigation tasks that must be performed accurately and rapidly is beyond human capability, and reliance on ground-based computers is often impractical.

For the first generations of manned spacecraft, computing requirements have been moderate and missions have been short. The requirements could be met by a computer of "traditional" design. Reliability, though a problem, could be handled by ensuring adequate reliability of the components of which the computer was built. No internal error-correcting or "graceful degradation" capabilities were included in the computers used for Gemini or Apollo, though program tests to detect errors and to attempt to restart the computation were used.

With the planning of post-Apollo missions, it becomes apparent that computing requirements and mission durations will both increase. Missions being proposed, such as a mission to Mars or a long-term earth orbital space station, have durations of over a year. New uses are being suggested for computers on board, such as resource management, experimental data reduction, and others. These changes mean that the spacecraft computer of the future must have both increased computing capability and increased long-term reliability compared to those in existence today.

The nature of the requirements for an advanced space guidance computer have been analyzed by Vacca, Phipps and Burke [1], by the staff of the MIT Instrumentation Laboratory [2, 3] and by Alonso and Randa [4]. The unanimous conclusion of these studies is that the spaceborne computer of the future will be of the "multiprocessor" variety. That is, it will consist of a number of "processors" connected to one or more "memories"; computing tasks will be assigned to the processors according to convenience at the time of execution. Advantages cited for this concept are ease of expansion by the addition of components, high computing capacity compared to a single computer for a given state of the art, and,

especially, graceful degradation in the event of failure of one or more components.

1.2 Description of the Multiprocessor

One design for a multiprocessor has been studied at length by the MIT Instrumentation Laboratory [2]. A multiprocessor similar to the one proposed there is the subject of this study.

The multiprocessor consists of four basic parts: the processors, the memory units, the input-output controller and the data bus connecting them.

The processors perform the actual computations. When starting a computing task, they obtain the required data from the memory units. Using a small "scratchpad" memory associated with each processor, they perform the required calculations. Having finished, they return the results to memory.

The memory units contain whatever data may be required by the programs being executed. For reliability, such methods as triplication and error-correcting codes would be used here. This study is not connected with this aspect of the memory, however; for the remainder of the discussion, it will be assumed that there is one completely reliable memory device in the system.

The input-output controller handles input-output in much the same way (from the point of view of a processor) as the memory handles storage. Again, for present purposes, it will simply be noted that this device exists and it will be assumed that it is perfectly reliable.

The data bus connects the other units of the system. In any collaborative system such as this, the ability of the various units to communicate is vital, and the data bus provides this ability. Each unit in the system can have at some time "control" of the bus. It can then send messages out over the bus, and if the nature of the messages is such that they require a reply (such as a request for data from memory) they will receive this reply.

The rationale for electing to use a time-multiplexed bus, rather than one of the other possibilities, for this purpose is discussed in detail in Ref 2. Here, this will be taken as one aspect of the definition of the subject computer. Also, as in the case of the memory units, we will assume an infinitely reliable data bus without going into the question of how it is made reliable.

Computing tasks —"jobs"— are initiated in the system by a bus message. A free processor, sensing this "job request" message, would prepare an acceptance message for the job. It would transmit this message when given control of the bus (unless some other free processor had already accepted the job). It would then obtain such data as the job requires from memory ("phase 1 of bus use") and would execute the job.

When it has finished the calculations, the processor would wait its turn for the bus. It would then return data to memory ("phase 2 of bus use"). In addition, it might specify that a certain job or jobs are to be executed at some future time: for example, in a sampled-data control calculation executed twice per second, each execution of the job would specify that the job is to be executed again one-half second in the future. This is done in the multipressor by sending a message to a "job stack" or "waitlist", where these requests are stored and sent out on the bus as job requests when they come due. The job stack is implemented in the multiprocessor as hardware.

In addition, the system would contain program memory — the sequences of instructions to be executed. This would be separate from data memory and would be transmitted over different channels. There are many possible organizations for this program memory: each processor might have its own copy of all programs, a crossbar switch arrangement might be used, or a second bus. In any case, requests for instructions are more predictable and more uniform over time than requests for data, and thus the instruction-transmission device (if present) does not present the problems that are presented by the data bus. It will be assumed in this study that each processor has its own copy of all programs, or, equivalently, the problem of instruction access will be ignored.

This has been by necessity a very brief description of the multiprocessor. For a further discussion of the points touched on above, and for discussion of many points not mentioned, see Ref 2.

Much of the performance of the multiprocessor depends on the algorithm by which devices are granted access to the data bus. Presumably, the processors are arranged in sequence along it, each one in turn having access to the bus if desired and then enabling its neighbor. Beyond this, however, the question of bus access method is open, particularly as regards the priority of the job stack. Four possibilities are:

1. Job stack has priority between each pair of processors. If a processor accepts a new job, it obtains use of the bus immediately, without waiting for its turn.

2. Job stack has priority between each pair of processors, as in scheme #1. If a processor accepts a new job, it waits its turn on the bus to send its acceptance message and to begin obtaining data from memory.

3. Job stack has priority at one point in the ring of processors. It is enabled by the processor proceeding it, and in turn enables the processor following it. The processor accepting the new job obtains use of the bus immediately.

4. Job stack is connected at one point in the ring, as in #3. The processor accepting the new job waits its turn to use the bus, as in #2.

Intermediate systems are also possible: the job stack might be connected at two points in the chain, for example, if it were desired to give it faster access than one connected at only one point, but with perhaps less hardware than would be necessary to connect it between every processor pair.

For this study, a multiprocessor will be considered as completely defined if one can state the number of processors in it, the speed of these processors, the speed of the bus, and which of the above four priority schemes it uses.

1.3 The Problem

It is clear that a multiprocessor computer such as described above can be built and can be made to operate. It is not clear just how efficient it will be when faced with a program of the general nature of spacecraft control programs, though one feels intuitively that its structure is well suited to their repetitive, sampled-data nature. It is not clear to what extent processors will interfere with each other in demanding the use of the bus, degrading performance to an unacceptable degree. Finally, it is not clear just what sorts of jobs characterize "spacecraft control programs", which makes it difficult to answer the other questions.

This study attempts to answer these questions. Proceeding from the last one, the question of characterizing "spacecraft control programs", a large example of such a program has been analyzed and broken down into smaller units. This provides the first definitive statement as to the composition of a typical "job mix" for space computers, and is used as a set of programs against which to test the multiprocessor performance. Chapter 2 is concerned with this analysis.

In a simple computer, in which various jobs are not competing for use of limited resources at the same time, the availability of such a statement as to the tasks to be performed by the computer would permit determining whether a certain system meets the requirements, and would permit sizing the various components — in terms of speed, capacity, etc. — for best performance. Where there is competition between jobs, the simple procedure of adding up the individual job requirements will not work, because there is an additional factor of delays introduced by waiting for a busy part of the system. More sophisticated methods of analysis must be used.

Two tools for the analysis of the multiprocessor are developed in this study. The first of these is a simulation, which accepts a definition of a multiprocessor computer (in the sense of the previous section) and simulates the execution of a set of jobs on it. This simulation is described in Chapter 3.

The second tool is the modeling of the multiprocessor as a Markov process, together with the development of computer programs for the analysis of such a model. These models are the subject of Chapter 4.

Chapter 5 discusses the results of the simulation and of the Markov process analysis of the multiprocessor. Conclusions are summarized in Chapter 6, which also gives recommendations for further research.

CHAPTER 2

THE JOB ANALYSIS

2.1 The Reasons for the Job Analysis

In order to evaluate any computer design, for any purpose, it is necessary to have an estimate of the type of tasks that the computer will perform. Thus, in selecting a commercial computer, a prospective user will ordinarily define certain "benchmark tasks" to be executed on all candidates for his selection, and base his decision at least in part on the results.

The need for an accurate characterization of the tasks to be performed is no less valid in the evaluation of a genuine computer design. This is particularly true when the computer being evaluated represents a departure from standard practice in a number of important respects, so that a user's "feel" for the importance of various performance parameters can be quite misleading. Unfortunately, no specification of the nature of the tasks performed by a space guidance computer — in terms of running times, iteration rates, data requirements, etc. — is available.

It was therefore decided to create such a specification by analyzing a suitable ensemble of spacecraft computer tasks. This analysis would have a certain importance in its own right, since it could be used in the future whenever such a specification is required for any purpose. More importantly for this study, though, it would provide a set of jobs and job statistics, not dependent on any one person's preconceptions and prejudices, for use by the simulation and Markov process analyses in the following chapters.

The jobs analyzed were taken from the most ambitious aerospace computer programing task available (and, to the author's knowledge, the most ambitious yet undertaken): the Apollo lunar mission. The highest computational loads maintained for more than a brief instant of time during this mission are imposed during the lunar landing itself on the Lunar Module (LM) Guidance Computer (LGC). The programs executed by this computer during this mission phase form the basis for the "job models" which were assumed to be executed on the multiprocessor during the simulation and theoretical analysis.

2.2 Lunar Landing: General Situation

The Apollo lunar mission has been described extensively elsewhere (for example, Ref 5). While a detailed description of it would be voluminous and not relevant to this study, a brief outline of it provides useful perspective for the descriptions of the computing tasks analyzed. It consists of the following stages:

1. A Saturn V rocket carries the spacecraft and a crew of three into earth orbit. The vehicles are the Command Module (CM), the Service Module (SM) (the linked CM and SM are referred to jointly as the CSM), and the Lunar Module (LM).

2. The three spacecraft leave earth orbit on a translunar trajectory.

3. They are inserted into a lunar orbit.

4. Two of the crew members transfer to the LM and separate from the CSM. Using the LM's descent engine, they brake from orbit and land on the moon.

5. Leaving the descent stage (engine, landing gear, etc.) on the moon, the two astronauts take off from the moon and enter a lunar orbit.

6. The LM and CSM rendezvous in lunar orbit, and the LM crew return to the CSM.

7. The CSM enters a trajectory toward earth, leaving the LM behind.

8. The CM separates from the SM, reenters the atmosphere, and parachutes to the surface.

The current task analysis is concerned with the activities of the LM guidance computer (LGC) during phase 4 of the above capsule description. This phase starts with the LM and CSM orbiting the moon a short distance apart. The descent engine of the LM injects it into an elliptical "descent orbit" and shuts down. As this orbit reaches its perilune, at an altitude of about 50,000 feet, the engine is re-ignited for the "braking phase" and remains on until touchdown. The braking phase lasts approximately eight minutes, at the end of which the LM is at an altitude of 8,600 feet and at a distance of about eight miles to the landing site. The LM then pitches forward, so that the crew can see the landing site through windows in the spacecraft. LGC displays inform the crew where the LM is to land, and the crew can modify this predicted landing site by moving a control device. At the landing site, the spacecraft rotates to a vertical position and lands more or less automatically (according to a mode selected by the crew at that time).

The present job analysis and simulation began just before the start of the

braking phase. At that time, the computer is performing a group of periodic monitoring jobs, such as checking for gimbal lock in the inertial guidance system. At a sufficiently early time to permit required actions to be completed before ignition, the crew keys in the braking phase program, referred to in the flight plans[6] and below as "program 63" or "P63". All jobs executed during the remainder of the landing follow in a spreading tree-like fashion from P63, except for the monitoring jobs that continue running as before the braking phase and crew-initiated landing site redesignations.

In the actual lunar landing, the visibility phase terminates when the LM reaches its landing site, about 75 to 135 seconds after it began. Determination of this event requires simulation of the environment or a predetermined cutoff time. In order to accumulate as many statistics as desired during the mission phase when the computer is most heavily loaded, it was assumed for purposes of this study that the landing site is never reached. The programed tests to determine this event are never successful, and the computer reaches a "steady state" of job execution in the visibility phase.

2.3 Development of the Job Model

The development of any "job model" must start from the definition of "job" in the given context. Intuitively, a "job" can be described as the execution of a series of computer instructions, initiated by a stimulus external to the job (such as reaching a given time, an external interrupt, etc.), and terminated at the end of the series of instructions. A "job" will, in general, require certain inputs and produce certain outputs; some of these outputs might be requests for the computer to execute some other job at some future time.

Since the basic definition of a "job" is taken as the execution of a sequence of instructions, a computer program such as that of the LGC cannot be divided into jobs in the literal sense. It can, however, be divided into sequences of instructions which, when executed, are jobs. These sequences can be identified, numbered, modeled as desired and used as input to the simulation, and this is what is done in the present study.

It would have been possible to start with a definition of a "job" as the sequence of instructions itself; one would then refer to the "execution of a job" when referring to what is called a "job" above. Either definition sacrifices convenience in some contexts for convenience in others. The definition used here is perhaps more meaningful from the viewpoint of the computer, which executes the jobs and is not concerned with their representation.

In the remainder of this study, the word "job" will on occasion be used loosely to refer to either the execution of a job or to the sequence of instructions being exe-

cuted. It should always be clear from context which is meant.

In the Apollo guidance computer systems, including the LGC, the identification of job sequences is simplified by the presence of the "executive" and "waitlist" programs [7]. These make it possible for the Apollo programer to code individual procedures without concerning himself with the mechanisms by which the computer will begin them or with others that might be running at the same time. These programs provide for two types of procedures:

"Tasks": short, high-priority functions that the programer specifies are to be executed at a certain time in the future. These are entered into a "waitlist", somewhat analogous to the multiprocessor job stack, and a timer is set to interrupt the computer when a waitlist entry is to be executed. Tasks are limited to four milliseconds duration, since they pre-empt the computer entirely until their termination. Only one task, therefore, may be active at a time.

"Jobs": longer, lower-priority procedures. Many jobs may be "active" at one time, and the executive program selects one of them for execution between tasks on the basis of assigned priorities. Jobs may be given to the executive only at the time that they are to be executed (they cannot be specified for a future time, as a task is.) Thus, when it is necessary to specify a long procedure for execution at a future time, the programer uses the artifice of entering a task into the waitlist for the time in question, and provides a task which, when executed, initiates the job and terminates itself.

It is, therefore, possible to define rules by which the LGC programs may be broken down into independent program sequences. Such a sequence is:

1. An LGC task, when that task has functions other than the initiation of an LGC job.

2. An LGC task and an LGC job together, when the task terminates itself after initiating the job and performs no appreciable additional computation.

3. An LGC job not covered by item 2 above.

4. When an LGC job pauses to wait for a crew response, the pause is treated as a separator between two job sequences. The response interrupts the computer to start the second one.

5. Exception to item 1 above: In one case, where the LGC executes a group of periodic monitoring functions, they are artificially combined into one "task" in order to keep the number of separate functions within the limits imposed by the LGC hardware and software. These functions are coded separately, and the first order of business when the overall task starts is to

select the individual functions to be performed on that specific iteration. In the breakdown of the program into job sequences, each of these functions was treated as a separate sequence, with its iteration rate determined by analysis of the LGC program. (These functions comprise job models 1 through 7 in the tables.)

The question of a general defintion of a "job sequence" is not nearly as clear-cut. A few comments of general relevance may, however, be made.

It can be stated with good generality that aerospace computing tasks will be characterized by a number of tasks that must be repeated at certain intervals. The computer on which these tasks are performed will have one or more clocks used for timing these intervals. Tasks performed at different intervals, such that the performance of one does not necessarily imply the performance of the other, are clearly separate jobs. Tasks that are related, so that the performance of one does imply the performance of the other, would probably be considered different jobs if there were an appreciable time lag between the two, but not necessarily if they are performed immediately after each other.

In the case where the above definitions—and particularly the last—leaves a number of jobs that are quite long and must be broken down further, it appears that a productive line of attack would be via the results produced by each job. It should be possible to take the calculations leading up to each item of data returned to storage and call each a separate "job". Often, there would be considerable duplication between "jobs" thus defined; in that case, they could be combined into one.

This is far from the last word in defining methods of job breakdown. The question is important, since the LGC program breakdown indicates a tendency on the part of programers to lump related functions—navigation, for example—into one job which becomes excessively long as a result. Efficient use of a multiprocessor requires that no job predominate in the total set of jobs, or else individual processors will have to be fast enough to handle that job with the required iteration rate and there will be considerable unused capacity in the form of the other processors. It appears that automatic job breakdown in a general situation would be a fruitful area for further investigation.

The job models abstracted from the LGC programs do not describe the sequences down to the functional level of addition, multiplication, etc. Rather, they are concerned with the ways in which the jobs interact with other jobs and with what would be, in a multiprocessor, such components as the bus, the job stack, etc. The job model is, then, concerned with the following:

*The number of words of data that the job in question must obtain from central memory, via the bus, before execution.

*The execution time of the job.

*The amount of data it returns to central memory when done.

*The system input-output requirements it might have.

*Its interactions with other jobs, such as setting flags that affect their execution.

*Its insertion of other jobs into the job stack.

*Its causing events that will cause future system interrupts (such as activating a radar unit that will cause an interrupt when it has completed its reading).

*Its periodic execution rate.

*Et cetera.

These job models and their parameters as listed just above were extracted directly from the listing of the LGC computer programs [6] [8]. Since this listing is voluminous— some 1400 pages of single-spaced computer printout—an attempt was made to devise an automatic analysis program that would perform the required work. It soon became apparent, however, that the construction of a program that would ferret out the ingenious programing tricks used by Apollo programers (under considerable pressure to conserve memory space at the expense of clarity) would be a more difficult task than performing the analysis by hand. Consequently, the analysis was done manually, using references 7 and 9.

The procedure used was conceptually quite simple. It was known that, before the braking phase, a group of monitoring jobs would be running. These were located in the listing and analyzed; instructions were counted, data requirements tabulated, and cyclic execution rates determined. The next job is P63, from which all further jobs flow. P63 was thus analyzed, noting the same parameters plus the occasions on which it submits other procedures to the waitlist or executive programs. The same process is carried out for these, and in turn for their descendants until the ends of the branches are reached or until a branch loops back on itself (as often happens, after an appropriate time delay, with periodic jobs.)

Where a program branched, an average execution time was taken if the branches differed only in execution time and if it was impossible to determine valid statistics for the branch without resorting to a simulation of the environment or of the detailed internal calculations in the LGC. If they differed qualitatively—e.g., one branch inserts a job into the job stack and the other does not—the program was analyzed in depth to determine the logic controlling the branch point. Where possible, the job models followed the actual program logic. Where this would require simulation of

the environment (for example, in calculating the number of pulses to be sent out to a gyro-torquing device, where this number depends on spacecraft rotation since the last group of pulses were sent out), simplifying assumptions were made; they were selected to correspond to reasonable conditions that would impose a heavy steady-state load on the computer. These assumptions are included in the job model descriptions of Appendix A.

The execution time of each job sequence was obtained by counting the instructions executed in one pass through the sequence. Apollo computer instructions can be written either in "basic" mode, in which the computer executes the instructions directly, or in "interpretive" mode, in which case another program called the "interpreter" interprets the instructions and carries them out. The advantage of basic mode is speed; the advantage of interpretive mode is the availability of a large repertoire of double-precision and vector arithmetic operations not built into the hardware of the computer. Since improvements in a computer order code, such as hardware floating-point capability, would speed up interpretive-type programs more than basic-type programs, separate counts were kept for the two types of instructions.

The "basic" execution time of a sequence was taken as the number of instructions executed in it. All AGC machine instructions (except "divide") take from 1 to 3 memory cycle times, with an average figure of 2 MCT being quite close to the true average. ("Divide", which takes 6 MCT, was treated as three instructions in the count.) Since one MCT is 12 microseconds in the AGC, one AGC basic instruction in the job sequence corresponds to approximately 25 microseconds of real time.

The "interpretive" execution time of a sequence was obtained by addition of the times for each instruction. Instruction counting was not acceptable here, since the times of different interpretive instructions vary quite widely (from 0.18 to 8.90 milliseconds). The count here is the number of milliseconds total time, as obtained from instruction timing charts[10] that the sequence takes for its execution. In the AGC, this time unit corresponds, of course, to one millisecond, or 1000 microseconds.

The total execution time, therefore, was specified as a pair of numbers: the number of basic instructions in the sequence and the interpretive execution time of the sequence in milliseconds. To obtain the total time, each of these numbers is multiplied by the appropriate timing factor and the products are added.

Counts of data "used" (i.e., obtained from central memory before the job can start) were obtained during the same analysis. Such a data item is easily defined: it is any word, except for constants, the contents of which are examined before being altered by the procedure in question. (The addressing structure of the LGC, in which references to a read-only rope memory containing instructions and constants are

distinguishable from references to the alterable core memory, simplifies the identification of constants.)

Counts of data words returned to memory at completion were obtained analogously. Any word, the contents of which are altered and not subsequently used by the procedure, must fall into this category. In addition, words that are altered and subsequently examined within a job may fall into it, if they are also used by other jobs. This inspection was done on the basis of the "data used" lists described in the previous paragraph.

Other factors in the job descriptions, not susceptible to numerical summarization (such as a significant change in the execution time of a procedure based on whether another procedure has or has not been executed, or the fact that a procedure may initiate another one every fifth execution, or similar) were noted as appropriate and were incorporated into the job models used in the simulation.

The job models themselves were presented to the simulation in the form of Fortran subroutines, which were either hand-coded or generated automatically by another program which accepted a simplified description of the job. The subroutines incorporated, by necessity, a considerable amount of procedural bookkeeping coding not related to the functioning of the mulitprocessor or to the nature of the job, so their detailed description is of minor relevance here. A typical subroutine is reproduced in Appendix B, together with descriptive comments. Their format is also discussed more fully in the description of the simulation, in Chapter 3.

2.4 Descriptions of the Individual Jobs

Within the programs executed during the braking phase, 43 independent sequences of instructions were identified and called "jobs". The present section describes these sequences, numbered 1 through 43 for ease of reference.

The identification of a section of LGC coding with a specific job model is not always unambiguous. For example, consider a job that is called for by another, is executed two times with an interval between them, and then terminated without calling for another execution. (Such a function might be a radar reading, where a number of readings would be made and averaged into the final result.) In the actual program, such a procedure would maintain an internal counter, which would be incremented on each execution and used to determine the correct terminal action.

In modeling such a program, there are two alternative approaches. One is to copy the logic of the program. In this case, the simulation subroutine representing the job would maintain an internal counter analogous to that in the actual program. The second approach would consider the LGC program to be two jobs, and hence two job models and two subroutines. The first of these, when executed, would call for execution of the second; the second would terminate without initiating any other jobs. Both would be identical in other characteristics such as execution time, data requirements, etc.

The difference between these two approaches is purely semantic. The performance of the LGC, the multiprocessor and the simulation will not be affected by the choice, since both would result in jobs of similar characteristics being executed at similar times. The computer does not care if humans are consistent in referring to its programs by one name at all times, or if they find it more convenient to use different names for the same coding according to the circumstances under which it is executed. It is the execution of a job that affects the computer, not the name of the job.

For this reason, too much importance should not be attached to the existence of precisely 43 jobs. The type of choice described above was made a number of times, always arbitrarily according to what seemed convenient in a particular situation, and thus not consistently one way or the other. The number of jobs could have been easily reduced below 40 or increased to over 60. What is of greater importance is the number and nature of the job executions during a period of time of interest. In the analysis of these executions, the job identification numbers play no part.

The jobs executed by the LGC during the braking phase of the landing can be divided into a number of categories. These are:

A. "Timeline" jobs (no. 10-13, 15-18, 29 and 30 in the tabulations). The crew-initiated "P63" is the first in this sequence, and the others follow in order, either after a wait for crew response or at a pre-determined time relative to engine ignition. Most of the other jobs are started either directly or indirectly, by one of these.

B. Monitoring jobs (no. 1-9). These are executed at varying, but pre-determined intervals, ranging from 0.02 to 2 seconds depending on the particular job. They monitor and control a variety of spacecraft functions. Jobs 1-7 would already be running when the crew keys in P63; jobs 8 and 9 are started later on in the sequence.

C. Navigation loop jobs (no. 22-27, 31, 42 and 43). This set of jobs, executed every two seconds, updates the vehicle state vector, processes radar readings, and computes landing trajectories.

D. Autopilot jobs (no. 36 and 38-40). Executed every 0.1 second, these jobs operate the control jets that control the attitude of the spacecraft, and the gimbals that point the descent engine.

E. Initialization jobs (no. 19-21, 28, 32-35, and 41). These jobs are each performed once. Jobs 19-21 are performed just before ignition, and job 28 just before the visibility phase; they set certain variables used

by other programs. Jobs 32-35 and 41 reposition the landing radar so as to point to the lunar surface. (The landing radar is used to update the LM state vector during landing by making measurements of the spacecraft position and velocity.)

F. Miscellaneous jobs (no. 14 and 37). Job 14 controls computer displays during a short portion of the landing; it is executed every second until the phase in question is over. Job 37 is executed whenever the crew operates a controller to redesignate the landing site. It sets flags that are tested by the navigation programs to control calculation of new trajectories.

When the "steady state" of the system is reached in the visibility phase of the landing, these jobs are not all active. Many have been executed once and will not be called out again. The active jobs are those of categories B, C and D above, plus the landing site retargeting job of category F. Some of these jobs are self-perpetuating: each time they are executed, their final action before termination is to insert into the job stack the request for their next execution. The others are initiated, for each execution, by jobs that are self-perperuating, so they too cycle at fixed intervals. (The exception is the landing site retargeting job, which is executed in response to a crew action.)

The jobs are described individually, and their numerical characteristics are tabulated, in Appendix A.

During the steady state of the visibility phase, jobs are executed at a mean rate of 67.35 per second. Twenty-three jobs are "active" at this time, with varying iteration rates. The jobs executed during a typical minute are listed in Table 2-1.

Table 2-1

#	JOB	TIMES/MIN	COMMENTS
1	1	619	477 without changing displays, 142 with
2	2	500	
3	3	125	
4	4	125	
5	5	250	
6	6	125	
7	7	125	
8	8	120	
9	9	30	
10	22	30	
11	23	30	

#	JOB	TIMES/MIN	COMMENTS
12	24	30	18 short, 12 long (landing site retargeted)
13	25	30	
14	26	30	
15	27	120	
16	31	30	
17	36	600	
18	37	12	using assumed crew activity statistics
19	38	600	
20	39	30	
21	40	300	
22	42	150	
23	43	30	

Using these figures, weighted averages of job execution times, bus use requirements, etc., can be obtained from the tabulations of the numerical job parameters given in Appendix A. The figures thus obtained are presented in Table 2-2.

Table 2-2

mean job execution rate: 67.35 per second
mean instruction execution rate: 25400 per second
mean job duration—instructions: 377.13
mean bus demand—cycles—obtaining data: 31.48
mean bus demand—cycles—returning data: 23.02
mean external interrupt rate: 2.7 per second

The "job mix" of the computer during this phase is characterized by the execution of a large number of very short jobs and a small number of very much longer ones. This result is obtained whether jobs are ordered by execution times (as seems natural) or by bus use requirements (as is perhaps more relevant to the multiprocessor).

A plot of bus use by jobs, showing the fraction of bus usages that exceed a given number of cycles in duration, is given as Fig. 2-1. This plot was obtained as follows: A "bus use" was defined as that period after a processor obtains access to the bus, having requested access, and before it enables another unit, having finished with the bus for the moment. During this period of time, the processor can send messages, send data and receive data. It was assumed that each message sent (job acceptance, job insertion in the stack, and job termination) takes one bus cycle, and each word of data sent or received takes two bus cycles—one to specify the word

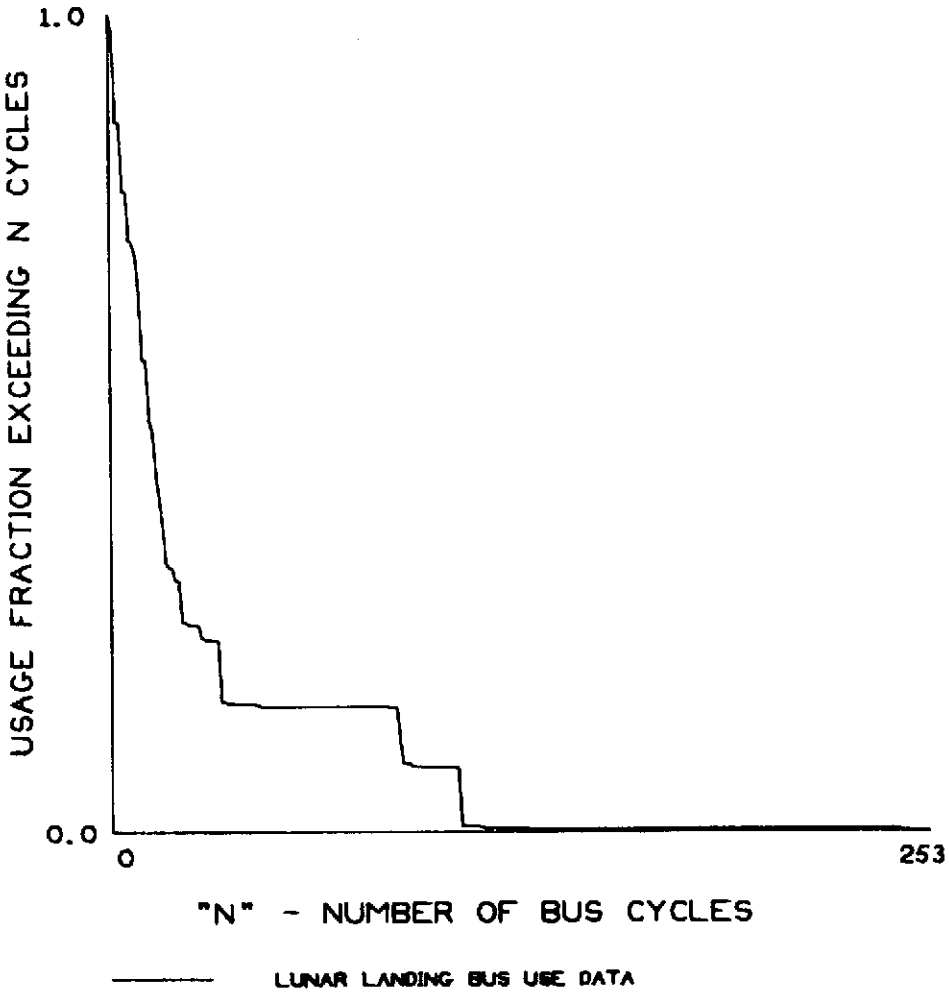


Fig. 2-1 Bus use distribution.

and one to transmit it. The number of cycles occupied in time by each bus use can thus be determined. The frequency with which each of these numbers occurs is also easily determined: it is the execution frequency of the job in question. The frequency of bus usages of varying durations, expressed as varying numbers of bus cycles, is therefore available. It is this frequency, plotted cumulatively, that is shown in Fig. 2-1. The largest number of cycles in one bus usage was 253. The tendency to a large number of short usages and a small number of longer ones is clearly seen.

This job breakdown reflects, presumably, the way in which people "like" to write guidance computer programs in the absence of any constraints. It is possible that constraints, particularly in the area of bus use, will be necessary to ensure efficient use of the multiprocessor. In such a case, it may well be advisable to place the burden of fulfilling such constraints on a compiler or other such tool used in the preparation of multiprocessor programs.

2.5 Conclusions from the Analysis

The figures of Table 2-1 in conjunction with the execution times for each job as given in Appendix A can be used to find the fraction of the total computing load accounted for by each job. Such calculations provide the information that job no. 24, the main navigation loop job, accounts for 59.14% of all the computations, and job no. 36, the autopilot job, accounts for 22.05%.

These figures are significant for two reasons. The first is that they reflect (presumably) some sort of "natural" breakdown of the total computing task, in the sense that it is "natural" for programers to break the task down in this manner. We thus have a number of very short jobs, and a few very long ones.

This information also has relevance to the multiprocessor. A five-processor configuration, each of the processors of which is one-fifth as fast as the AGC, could presumably (with an infinitely fast bus) execute the AGC computing load. With the existing job breakdown, though, this is not the case by any means. The navigation loop must be cycled every two seconds, and takes about 1.18 seconds of AGC computing time. A multiprocessor in which the processors were one-half the speed of the AGC, then, could not physically complete one cycle of the loop before it was time for the next one—even though, in the overall system, there would be unused capacity in the other processors. A system in which the individual processors were one-fifth the speed of the AGC could not complete the autopilot job within one cycling period for that job, either.

We may therefore conclude that, for efficient use of the multiprocessor, it is necessary to structure the total computation so that no individual job accounts for an overly large portion of it. If it is inconvenient or otherwise impractical for this structuring to be done by the programer—and, in a "convenient" system, the pro-

gramer should not have to do it—then it must be done automatically, by compiler or other such programing aids.

A side effect of this distribution is that the simulation (described more fully in the following chapter) cannot simulate a multiprocessor having processors less than half as fast as an AGC as if it were executing this job set, for the simple reason that such a multiprocessor could not execute this job set. Since it is desirable to simulate a multiprocessor in a heavily loaded condition, and since the only way of loading the system for a given job set is to slow it down, this posed a problem.

The expedient that was chosen to circumvent this difficulty was to create a "modified" job set, identical to the one discussed above except for jobs no. 24 and 36. The autopilot job was reduced to one-half its former computational size, and the navigation job was divided by a factor of 4.5. (Data requirements, bus use, etc., were not touched.) The resulting job mix represents the lunar landing fairly well, though not with complete accuracy, and could be thought of as representing it in a computer with an instruction set especially suited to the execution of navigation-type programs. In any case, this provided a means of obtaining a job set for simulation that had a strong connection to reality and could also be used to simulate a heavily loaded system. In discussion of simulation results, the modified job set was generally used.

CHAPTER 3

THE MULTIPROCESSOR SIMULATION

3.1 The Reasons for the Simulation

When a system of interdependent processes reaches a certain degree of complexity, it becomes impossible for humans to appreciate fully the interactions among parts of the system and to evaluate the effects of these interactions. It becomes necessary to use a systematic method of studying the system, which can provide some facts on which an evaluation of the system can be based. Two different approaches to this analysis are in use: simulation[11] and queuing theory[12].

The differences between these approaches have been discussed fully in the literature. Wallace and Rosenberg[12] describe them in justifying a queuing theory analysis; Merikallio and Holland[13] do so in justifying a simulation study; and Scherr[14] contrasts them as part of a study using both approaches. It is fair to state that each has its advantages.

Simulation has the advantage of describing exactly what is happening in the system being analyzed, not an average of what might happen. A system being simulated can be examined during the simulation at any time. There are no restrictions on the analysis of simulation data, while the types of statistics available from queuing theory studies are often limited. Finally, the validity of the simulation as representing the system is usually obvious (though the statistical validity of the results might not be—see, for example, Fishman and Kiviat[15]).

Disadvantages of simulation, as contrasted with queuing theory, include lack of generality: the output describes the system behavior only for the given initial conditions, which might not be typical. The effects of this can often be overcome only at the expense of large amounts of computer time—another disadvantage. Programing requirements are also large, since each part of a complex system needs normally to be modeled separately (or at least each unique part; advantage can often be taken of modularity). Thus, the two approaches are complementary. Queuing theory can be used for comparisons between systems and for overall evaluations, while simulation is invaluable for examination of "worst case" conditions and for close examination of the system

3.2 The Job Models in the Simulation

The usual simulation of a computer system does not use specific tasks to be executed by the computer being simulated[14, 16]. Instead, tasks are created as required by the simulation from a stochastically defined "job mix". Parameters such as job length, input-output requirements, arrival intervals between jobs, etc., are specified to some degree of statistical precision for the set of all jobs executed by the computer, and these specifications are used in conjunction with a random-number generator to create the jobs that are processed by the simulated system.

If the statistics used in the specification of this job mix are sufficiently close to the true statistics of the system being simulated, the results of such a simulation can be quite accurate, as shown by Scherr[14]. For a system as yet nonexistent, however, the estimation of accurate statistics is subject to serious errors. A starting point for these statistics, having demonstrable validity beyond the user's "best guess", is an absolute necessity if the simulation is to produce useful results. No such statistical data describing space guidance computer programs were available.

It would have been possible to take the weighted means of the running times, iteration rates, etc., for the jobs analyzed in the previous chapter, and to use them as defining the job mix. In doing so, however, some of the advantages of simulation would be lost. The knowledge of how the system behaves for a realistic program would be replaced by the knowledge of how the system would behave for programs that, on the average, resemble realistic programs. Also, the usefulness of the simulation as a basis for evaluating the results of the theoretical analysis would be sharply reduced, and in this case there is no other way to evaluate those results for accuracy.

To circumvent these problems, the simulation used in this study did not create its jobs from statistically-defined parameters. Instead, it used the actual jobs of the previous chapter, each specified individually and each distinguishable from all the others. Each of these jobs carries in the simulation a specification of its execution time, input-output and data requirements, iteration rate, interactions with other programs, etc. Thus, the simulation provides a look at a multiprocessor as it would execute the lunar-landing Apollo computer programs.

The difference between this simulation and the typical statistical simulation of a digital computer accentuates the differences between the usefulness of simulation and the usefulness of queuing theory as methods of analysis. This simulation is far more deterministic, less stochastic, than most others. It therefore acquires statistical validity in less time than most others, but this validity is less general. This, on the one hand, makes the choice between simulation and theoretical analysis more clear-cut on the basis of the results desired, and on the

other hand reduces simulation running time to the point where it is roughly similar to the running time of the analysis discussed in the following chapter.

3.3 Structure of the Simulation

The simulation programs were prepared in the Fortran IV language for the IBM 360 computer. Examination of languages written expressly for the purpose of simulation[17, 18] led to confirmation of the conclusions expressed by Scherr[14]: generally-available simulation languages are inefficient in their use of computer time, not well suited to the modeling of computer systems, inflexible in input-output, or difficult to learn, generally having more than one of these drawbacks. Fortran, with its flexible subroutine calling capability, permits one to write a set of subroutines that simulate the performance of the various multiprocessor functions; once these subroutines are written, coding the simulation is at least as simple as it would be in a special-purpose language, and the capabilities of the Fortran language are available for use as needed.

Internally, the simulation represents the state of the system by a group of tables. There is a table to represent the contents of the job stack, a table giving the state of activity of the processors, etc. A simulated clock keeps track of the "time" in the simulation; at the completion of one event it is advanced to the start of the next event, so that time is not wasted simulating periods of time in which the state of the system does not change. A final set of tables stores accumulated statistics of the run.

Control of the simulation is exercised by a "central simulation loop". It examines the job stack and the processors (in an order determined by the priority scheme of the multiprocessor being simulated) to determine their requirements for the bus. If a demand for bus use is "past due" when the unit demanding the bus is scanned, it is given the bus. If no requests for the bus are pending, the earliest request is satisfied.

An event can be of three types: an external interrupt, a job due to start, or a running job needing the bus.

An external interrupt, when it occurs, enters into the job stack a request for the execution of a job. The interrupt then becomes equivalent to any other job waiting to start.

If a job is due to start, and a processor is available, the processor is assigned to the job, and control of the simulation is passed to the subroutine representing the job that is to be executed. A flag internal to the simulation is set by the central control loop, to inform the subroutine that it is being called for the purpose of initiating job execution.

If a job that had been running needs the bus, control is again passed to the subroutine representing the job. In this case, the control flag takes on a different value, so that the subroutine can tell that it is being called to terminate the execution of the job.

After completion of the activity in question, which includes advancing the system clock by the duration of the activity, the central simulation loop is again given control. This cycle continues indefinitely, modified by control cards supplied by the user.

General control over the simulation is exercised by a packet of control cards. The configuration of the multiprocessor can be specified completely for each run, as can the speeds of the individual processors (see Fig. 3-1). In addition, various output options can be selected, and the user may specify if the simulation run is to start from scratch (in which case the initial contents of the job stack must be given) or to continue where a previous run terminated (in which case a "restart deck", punched by that previous run, must be supplied). The user may also obtain "snapshots" of system status at any desired time during a run (see Fig. 3-2), change the output options during a run, cause external interrupts to occur, and cause failure of a selected processor or of the processor executing a selected job.

At the end of the run, a statistical summary of the run is printed. This summary is useful both in studying the multiprocessor and in evaluating the accuracy of the Markov models discussed in the following chapter. The numerical portion of such a summary is reproduced as Fig. 3-3 and 3-4.

3.4 Description of Simulation Output

Four items of simulation output are reproduced as Fig. 3-1 through 3-4, and were referred to briefly above. This section is designed to make them more comprehensible to the reader.

In addition to the items reproduced, additional simulation output is available when selected by the user. This additional output consists largely of lists of individual events and their descriptions. It is of interest only to a person following the progress of a specific job or processor, or for debugging purposes.

Figure 3-1 is output from the beginning of a run, and is taken entirely from user input cards. The configuration of the multiprocessor is specified by the number of processors, the number of job models supplied to the simulation, the number of positions available in the job stack and the priority scheme of the multiprocessor. The speed of the system is given by three parameters: the time it takes a simulated processor to execute one Apollo Guidance Computer instruction, the time it takes to carry out the computation implied by one millisecond of AGC

```
START OF SIMULATION


INITIAL JOB ASSIGNMENTS:
 1: JOB NO.  1 FOR T=    0.010000
 2: JOB NO.  2 FOR T=    0.011000
 3: JOB NO.  3 FOR T=    0.012000
 4: JOB NO.  4 FOR T=    0.014000
 5: JOB NO.  5 FOR T=    0.015000
 6: JOB NO.  6 FOR T=    0.017000
 7: JOB NO.  7 FOR T=    0.019000
 8: JOB NO. 10 FOR T=    0.900000
 9: JOB NO. 36 FOR T=   20.000000

SIMULATION SET TO TERMINATE AT T= 300.000000 SECONDS.

RUN PARAMETERS:
 5 PROCESSORS
43 JOBS
10 JOB STACK POSITIONS
    25.0 MICROSEC PER BASIC AGC INSTRUCTION
  1000.0 MICROSEC PER AGC INTERPRETIVE MSEC
    15.0 MICROSEC PER BUS MESSAGE
PRIORITY SCHEME 1
```

Fig. 3-1 Simulation output: system description and initial conditions.

```
TERMINAL SNAP


BUS MOST RECENTLY USED BY PROCESSOR NO.  2

BUS WAS RELEASED AT T= 300.001620

CURRENTLY ACTIVE PROCESSORS:
PROCESSOR   JOB   NEXT BUS REQUEST
    1        24      300.479151
    2        36      300.015295

CURRENT CONTENTS OF JOB STACK:
STACK POS   JOB   TIME TO BE CALLED
    1         1      300.010000
    2         2      300.011000
    3         3      300.012000
    4         4      300.014000
    5        22      301.830861
    6         5      301.996791
    7         *      300.015000
    8         8      300.356774
   11         6      300.017000
   12         7      300.019000
   13        23      301.780861

NEXT RUPT SCHEDULED FOR T= 302.702733


END OF SNAP
```

Fig. 3-2 Simulation output: "snapshot" of system status.

```
BUS USED  5.17 PERCENT OF RUN TIME
TOTAL JOB ACCEPTANCE DELAYS:  0.239443 SECONDS, MEAN OF  0.000012 SECONDS PER JOB; MAX DELAY WAS  0.003539 SECONDS
 19295 JOBS INITIATED THIS RUN, MEAN OF    64.347 PER SECOND
 19295 JOB REQUESTS ISSUED


PROCS IN USE      PCT OF TIME (BASED ON 300.000015 SECONDS)

      0              47.6598
      1              41.9743
      2               9.5719
      3               0.7504
      4               0.0440
      5               0.0004

TOTAL PROCESSOR BUSY TIME:190.643624 SECONDS
EQUIVALENT SINGLE-PROCESSOR LOAD:  63.55
ACTUAL COMPUTATION TIME:175.171171 SECONDS
COMPUTATIONAL EFFICIENCY:  91.89

TOTAL PRODUCTIVE TIME (COMPUTATION PLUS BUS USE):190.401736 SECONDS
TRUE SINGLE-PROCESSOR LOAD:  63.47%
OVERALL EFFICIENCY:  99.87%
SYSTEM LOAD IS   12.71%
TRUE LOAD AT 100% EFFICIENCY WOULD BE   12.69%
INFORMATION TRANSFER EFFICIENCY  98.44%
WASTED TIME IS    1.59% OF BUS USE TIME


MAXIMUM OF 17 SIMULTANEOUS JOB STACK ENTRIES


DELAYS IN JOB ACCEPTANCE - MILLISECONDS:
   0     0.5      1        2        4        8       15       30       60      125      250      500    1 SEC    2 SEC
  TO      TO      TO       TO       TO       TO       TO       TO       TO       TO       TO       TO       TO      AND
0.499   0.999   1.999    3.999    7.999   14.99    29.99    59.99   124.9    249.9    499.9    999.9   1.999     OVER

19246      17       33        3        0        0        0        0        0        0        0        0        0        0
```

Fig. 3-3 Simulation output: statistical summary, sheet 1.

JOB EXECUTION FREQUENCIES:

JOB	TIMES
1	3075
2	2500
3	625
4	625
5	1250
6	625
7	625
8	449
9	113
10	1
11	1
12	1
13	1
14	45
15	1
16	1
17	1
18	1
19	1
20	1
21	1
22	140
23	139
24	140
25	125
26	139
27	560
28	1
29	1
30	1
31	125
32	1
33	1
34	1
35	1
36	2901
37	68
38	2800
39	140
40	1400
41	0
42	625
43	139

Fig. 3-4 Simulation output: statistical summary, sheet 2.

interpretive time, and the time it takes the system to send one message out via the bus. The initial contents of the job stack are also specified, implying that the run reproduced was a "fresh start" rather than a "restart"; the particular numbers used have no significance.

Figure 3-2 is a typical "snapshot" of the system, which can be requested by the user for any simulated time and which is also provided automatically by the system on termination (as was the one shown) or in case of errors. It is self-explanatory.

Figure 3-3 contains the bulk of the statistical summary of the run. The performance figures used in the analyses of Chapter V were obtained from summaries such as this. The data on this page are obtained as follows:

The bus use time is obtained by summing all requests for bus use over the period of the simulation. Each message sent takes a time of one bus cycle, and messages are sent for job requests, job acceptances, insertions into the job stack, and job terminations. Each request to transmit a word of data, in either direction, takes two bus cycle times: one to identify the word and one to transmit it.

Job acceptance delays are defined as the interval between the time at which the job should have been executed—based on job stack information—and the time at which the job acceptance message was sent out by a processor, minus the time of the one bus cycle taken up by the job request message itself. They are tabulated over the run to obtain the data in the second line of the summary and in the table at the bottom of the page.

The number of jobs "initiated" is specified in the summary rather than the number of jobs "executed", for the sake of precision. It is felt that a certain ambiguity arises in the definition of a job being executed if it is in progress when the simulated run starts or ends.

The number of job requests is not necessarily the same as the number of jobs initiated, since a job can be unacceptable when a request for it is issued. Should this be the case, the request will be issued again, as soon as a presently-running job terminates.

The amount of time during which a processor is busy is taken as starting at the beginning of its transmission of the job acceptance message, and ending at the end of transmitting its termination message. (The percentages in this table are not necessarily based on the precise termination time specified in the run request, since the simulation will run past this time until the next bus release.)

The total processor-busy time is computed from the entries in the "processors in use" table directly above it. The "equivalent single-processor load" is the processor-busy time divided by the duration of the run. The "actual computation time" is accumulated during the run from the instruction execution specifications of the job models, and the "computational efficiency" is defined as the actual computation time divided by the total processor-busy time. This efficiency is not a particularly good measure of system performance, since some bus use is implied in every job; therefore, not even a system with no delays could achieve 100% computational efficiency.

The "total productive time" is obtained by adding the bus use time (bus use percentage times run time in the summary, but actually carried separately internally) to the computation time mentioned above. From this sum, the time during which the bus was used for job-request messages must be subtracted, since this time is not assignable to processors and misleading results would be obtained if this correction were not made (such as efficiencies of 102%). This correction is obtained by multiplying the number of job requests issued by the bus message time.

True single-processor load is the time thus obtained divided by the run time. This load is "truer" than the one above because a single processor presumably would not interfere with itself in using the bus.

Overall efficiency is defined as total productive time divided by total processor-busy time. This is a better measure of system utilization than computational efficiency, but is biased upwards as system computational speeds go down because computation is by definition 100% efficient.

System load is the total processor-busy time divided by the total available processor time: run time multiplied by the number of processors. Alternatively, it is the "equivalent single-processor load" divided by the number of processors in the system. The "true load at 100% efficiency" is the system load that would prevail in the absence of interference in bus use; it is the system load multiplied by the overall efficiency.

Information transfer efficiency is the best measure of the delays caused by interference in use of the bus. It is defined as the time spent by processors using the bus, divided by the time during which processors were busy but not computing. Equivalently, it is the time processors were using the bus, divided by the time in which they were either using or waiting for it:

$$ITE = \frac{T_b}{T_b + T_w} \qquad (3.1)$$

Wasted time is information transfer efficiency expressed in a different way; they are related by a simple expression obtained from Eq. (3.1)

$$T_w/T_b = 1 - \frac{1}{(ITE)}$$

The maximum number of simultaneous job stack entries is of interest largely to a person preparing a set of job models, and who might be interested in the overall system requirements of this set.

Figure 3-4 shows the number of times that each job model in the simulation was executed. Some of the execution times in this example are zero; these correspond to jobs that would be executed during mission phases other than the one simulated by this particular run.

3.5 Presentation of Job Models to the Simulation

Job models, corresponding to the 43 jobs discussed in the previous chapter (or to any other job that the multiprocessor might execute), were presented to the simulation in the form of short Fortran subroutines. These subroutines, in turn, consist largely of calls to the multiprocessor-simulating subroutines—such as GET, to obtain data from memory; INSERT, to insert a job into the job stack; TERMIN, to terminate execution of a job; and others.

A still easier method of preparing job models for simulation is provided by a special-purpose program generator written for use in conjunction with the simulation. This program generator takes a bare-bones description of a job, in terms of its numerical parameters and their stochastic variations, and creates the required Fortran subroutine from this description. It is possible with this program to create a complete set of job models without any knowledge of Fortran or of the simulator. It is felt that this "program generator" approach combines all the advantages of a special-purpose simulation language with the efficiency and flexibility of Fortran—which remains available for the more sophisticated user or for the user for whom the capabilities of the special-purpose language are not adequate.

Of the 43 jobs in the lunar landing, about 35 could be written with the "program generator" above. The other jobs generally involve inter-job communication (such as a navigation job that performs additional functions whenever the crew landing-site redesignation job is performed, and must be able to determine

whether or not to perform them). For these jobs, Fortran was used directly.

An annotated example of a job model, as expressed in the program-generator language and in Fortran, is given in Appendix B.

3.6 Internal Functioning of the Simulation

This section goes into the internal operation of the simulation in a somewhat greater degree of detail than did Section 3.3. It may be bypassed by those with no interest in this topic with no loss of continuity.

The first requirement in any simulation is the definition, in precise terms, of the system being simulated. This definition can be broken down into three parts: specification of the limits of the system, specification of the parameters of the system, and specification of the state of the system.

The limits of the system being simulated are, in this case, provided by the ground rules stated in Chapter 1. A multiprocessor computer is being simulated in the overall sense, with no attention being given to either the environment of the computer or to the internal functioning of the components (e.g., processors) of the computer. Only those components that must be simulated because of their effect on system behavior—such as the job stack—will be.

The parameters of the system were also given in broad terms in Chapter 1. They are:

* the number of processors in the system
* the number of positions in the job stack
* the speed of the processors, expressed as a pair of numbers: the time in which one processor could execute the equivalent of one AGC instruction, and the time in which it could carry out the computations which the AGC carries out in one millisecond of interpretive calculation
* the speed of the bus, expressed as time in microseconds to send one message
* the priority scheme of the multiprocessor
* the number of job models in the simulation (not strictly a parameter of the multiprocessor, but required by the simulation programs).

Given these parameters, and given the specification of each job, a person could conceptually simulate the multiprocessor with pencil and paper. He would write down the activity of each processor, would assign jobs to processors when they come due, and perform all functions that would be performed in the multiprocessor itself. In doing so, he would find himself altering again and again a group of quantities that describe what the system is doing at any moment—the "state" of the system. In the present simulation, the state of the system is represented by the following list of items:

* the present time
* the time at which the bus will be released, if busy
* for each job stack entry: the name of the job and the time at which it is to be executed
* for each processor: the job using it (if any) and the time at which the job will next require use of the bus
* the number of the processor now using the bus

This set suffices to describe the state of activity of the multiprocessor with an adequate degree of completeness for purposes of the current simulation. To it, we may add two additional sets of information, which are convenient to have in a computer simulation:

* bookkeeping information required for simulation "administration": the time at which the run is to stop, the time of the next interrrupt, the output option selected by the user, etc.
* statistical information accumulated during the run: bus use times, job-starting delays, processor-busy times, job requests, and so on—limited only by the time and imagination of the person creating the simulation.

When these sets are written down, attention may be transferred to another area. It is possible to write down a list of the various functions that are performed in the multiprocessor and which will have to be simulated. These functions are:

* determination of the next event
* sending of a job request by the job stack
* acceptance of a job by a processor
* obtaining data from central memory
* reading data from system input devices
* releasing the bus
* job execution
* returning data to central memory
* writing data on system output devices
* inserting jobs into the job stack
* job termination and processor freeing
* occurrence of system interrupts

When this list also has been written down—and it should be recognized that such lists are rarely final, but change as understanding of the multiprocessor and of the simulation grows—it is possible to determine the effect of each such event on the state of the system. For each event in the system (all the functions in the above list are events, except the first) a "subroutine" (Fortran terminology for a semi-independent section of a program, which is executed only when called for by some other program section) was written. The first function, determination of the next event, is performed by the main program, which through this function

exercises control over the other parts of the simulation.

In operation, the main program scans the state of the system and determines the next event. In this scan, it is concerned with three factors: the times of the entries in the job stack, the times at which the busy processors will next want the bus, and the time of the next interrupt. The scan is performed in a sequence determined by the priority scheme of the multiprocessor being simulated.

If the next event is an interrupt, the job to be executed is inserted into the job stack by the appropriate subroutine. It is then treated exactly as any other job stack entry. Eventually the system must therefore reach a state in which the next event is not an interrupt.

The system is then scanned, each processor and the job stack in its turn, for events that should have already started but were not able to because other events were in progress. The first such event found is executed. It will be either a job trying to start or a job trying to get the bus so as to terminate.

If no such past due event is present in the system, the event that is next due is executed. It must also be one of the above two types of events.

If the event thus selected is a new job to start, the processors are again scanned, in an attempt to find a free one for the new job. If no such processor is found, the job is flagged as unacceptable, and the scan is repeated. (This procedure may appear strange at first because, if all processors are busy, it requires each job to be flagged as unacceptable individually. It was selected because, first, jobs can be flagged as unacceptable for reasons other than all processors being busy;* and, second, because it is likely to resemble the procedure that would be adopted in a real multiprocessor because of its conceptual simplicity). Eventually, the next event will become a running processor wanting the bus; this event can be executed.

The next executable event will thus be defined as to type and as to the processor on which it will take place. It is also defined as to the job model that describes it more fully. Further activity is under control of the job model subroutine (These are described more fully in Appendix B).

The job model subroutine controls the execution of most of the multiprocessor functions. It defines to the appropriate lower-level subroutine the amount of data it takes and returns, the time it requires for calculations, the jobs it inserts into the job stack, and any other relevant items of its behavior. The lower-level subroutine in question takes these specifications

*Such as memory conflict. This occurs when two or (Continued on next page)

and translates them into changes in the appropriate state and statistical tables. Control is then returned to the central simulation program, which searches again for the next event; this procedure continues until the end of the run.

more jobs alter the same item in memory. When one of these jobs is running, the other must not be allowed to; the final result in memory if this rule is not followed will be the result of only one of these jobs, the other one being ignored. When this can occur, it is therefore necessary that a job, before starting, check for the possibility that portions of memory that it requires might have been "locked out" by another job.

The simulation provides facilities for specification of memory lockout by job models and for testing for conflicts at the time of job acceptance. If a job cannot be accepted due to memory conflict, it is flagged as unacceptable and the simulation looks for the next event.

In the analysis of the lunar landing jobs, memory conflict was not a factor because the activities that could alter each part of memory were collected into the same job in every case. Memory conflict will not, therefore, be further considered in this study. In another situation, however—in particular, in a situation where long jobs were broken down into a number of shorter ones—this problem might have to be dealt with.

CHAPTER 4

THEORETICAL ANALYSIS

4.1 Concepts of Queuing Theory

This section is not intended as a text on queuing theory. It is, rather, designed to provide the reader with enough background to understand the sections that follow. For further reading on the subject, Morse[19] provides a good introduction; Markov processes are treated by Howard[20]; and the computational methods used are based in large part on Wallace and Rosenberg[12], whose paper also includes a description of Markov processes.

The general name of "queuing theory" refers to an approach to the analysis of a collection of interdependent processes, totally unlike simulation. The approach consists of defining a number of "states" of the system in question, and of determining the probable behavior of the system from analysis of the transitions among the states.

The first step in the analysis is to define the variables that describe the system; these are called "state variables". Consider, as an example, the Acme Widget Company. Upon receipt of an order for widgets, its manufacturing department makes them, one at a time. They are stacked for inspection and checked individually before shipment (Acme guards its reputation for quality jealously). We would have the following state variables to describe the state of the firm:

VARIABLE	POSSIBLE VALUES
number of unfilled orders	0 to infinity
state of manufacturing department	0 or 1 (idle or busy)
number of widgets waiting for inspection	0 to capacity of bin
state of inspection department	0 or 1 (idle or busy)

Each combination of state variables is called a "state" of the system. In the above example, the state (4, 1, 0, 0) might mean "four unfilled orders, manufacturing department working, none waiting for inspection, inspection department idle."

For convenience, numbers are usually assigned to the states; the above state might be "state 180".

It should be noted that the "state" of a system defines only that part of a system described by the state variables. There are often many alternatives in the choice of these, and both the complexity and the usefulness of the analysis will depend on the choice made. (In times of stress, Acme could perhaps have the vice-president make widgets. This would increase the speed of manufacture, and would introduce another state variable or another possible value for an existing one. Furthermore, the above set of variables gives no information about the adequacy of Acme's accounting methods, the quality of its customer relations, etc.) Part of the job of the analyst is to select the minimum set of states consistent with useful results.

A potentially bothersome characteristic of the above example is that it has an infinite number of states. This is not a difficulty in many types of analysis, but when the system becomes sufficiently complex to require a computer it is necessary to limit the number of states. It is often possible to reason that very large values of a potentially infinite variable are extremely unlikely and to impose a limit on the value of this variable, thus turning the problem into a finite-state one. This simplification can be justified in either of two ways: by noting that, in the analysis of the finite-state problem, the states corresponding to the highest values retained have sufficiently small probabilities; or by determining, through theoretical analysis of a simpler but related problem, that the states discarded have sufficiently small probabilities. This type of situation arises in the analysis of the multiprocessor, where there is no fixed limit to the number of jobs that can be past due for starting (or, equivalently, the limit is so large as to make the total number of possible states intractable).

The second stage of the analysis is to determine the events that can cause transitions from one state to another. Even in a system with a large number of states, there are usually few such events. In the case of Acme Widgets, transitions would be caused by the arrival of an order, the completion of the manufacture of a widget, and the completion of the inspection of one.

The rates at which these events take place must then be determined. One way of expressing this rate is by a function $S_o(t)$, the probability that the interval between two successive occurrences of the event is greater than t. This interval can be the time between order arrivals, etc.

It is now possible to compute the steady-state probability of the system

being in any particular state. Assuming that within a sufficiently short period only one such transition-causing event will occur, it is possible to write an equation balancing transition rates into and out of each state. The equations expressing this balance involve the probabilities of being in the various states (since the rate of transition from state A to state B is related to the probability of being in state A to begin with) and can be solved for these probabilities.

The form of these equations is particularly simple when the distribution of intervals between events is exponential in time (that is, the number of events within a given interval has a Poisson distribution). The probability of having an event in a given interval is in this case independent of the time since the last such event. The transition rate from one state to another is then simply proportional to the probability of the initial state. The constant of proportionality is the reciprocal of the mean arrival time, service time, etc., of the event in question. For a process with mean time $1/\mu$, we have

$$S_o(t) = e^{-\mu t} \quad . \tag{4.1}$$

and the probability of a transition by this mechanism out of a state A within the interval dt (where P_A is the probability of being in state A) is given by

$$\mu P_A dt \quad . \tag{4.2}$$

The equations of balance for the exponential case are sums of terms of this sort, and are therefore linear in the P_i. For a system with n states, there are n such equations in the n unknown probabilities. They are dependent, and are equivalent to n-1 independent equations. A final equation is added by the condition that the state probabilities sum to 1; there are thus n equations in the n unknowns and the system can, in principle, be solved.

It is often possible to relax the restriction to exponential processes by the introduction of additional states, modeling a non-exponential process by a collection of exponential ones. Morse[19] gives a number of examples of this; this procedure will be used in subsequent sections of this chapter.

The traditional approach of queuing theory, as described by Morse[19], is to solve the equations of balance analytically. One obtains expressions for the state probabilities P_i, and numerical values for the transition rates can be entered into these expressions as desired. This type of solution is limited to the case where there are very few states or very few state variables. Many complex problems are quite intractable by this approach, attractive as the thought of an easily evaluated expression for the state probabilities might be.

A second approach to the solution, which permits numerical solutions of very large systems, is to treat the system as a "Markov process". A Markov process can be defined mathematically as a system in which the state transition rates are independent of system past history; thus, the next system state in such a process is a stochastic function of the present state. A queuing-theory problem with all exponential processes satisfies this definition. Markov processes introduce a vector-matrix notation that simplifies the handling of large systems and is suitable for implementation on a digital computer.

If the transition rates are independent of time, each state has a steady-state probability. Rather than balancing the transitions into and out of each state to find these probabilities, the analyst describing a Markov process writes down, for each state, the transitions into it only. (Equivalently, he might choose to concentrate on the transitions out.) Each transition is thus noted one time, rather than twice as in the equations of balance above. He then writes down a "transition intensity matrix" Q, the elements of which are the rates of these transitions. The diagonal elements of the matrix are chosen to make each column sum to zero. The state probability vector $\bar{x}$ can then be shown to satisfy the differential equation

$$\frac{d\bar{x}}{dt} = Q\bar{x} \tag{4.3}$$

and, in the steady state,

$$Q\bar{x} = 0 \tag{4.4}$$

where the ith element of $\bar{x}$ is the probability that the system is in state i.

Various methods of finding the steady-state value of $\bar{x}$ exist. Two, described by Howard[20], require the calculation of functions (the inverse or the exponential) of a matrix the size of Q. This is impractical, even on a large-scale digital computer, for moderately large problems: the transition intensity matrix for a 500-state problem has 250,000 entries!

An alternative method described in Wallace and Rosenberg[12] eliminates much of the storage requirement and makes possible the numerical solution of Markov-process problems with large numbers of states. It employs an iterative approach to the solution, using

$$\bar{x}_{k+1} = (I+\Delta Q)\bar{x}_k \tag{4.5}$$

where Δ is a scalar and I is the identity matrix. If Δ is chosen so that all the diagonal elements in ΔQ are less than unity in magnitude (they are all negative, because of the method by which they are calculated), then all the elements of $I+\Delta Q$

will be positive and less than 1, and each column will sum to 1. The new matrix I+ΔQ can be interpreted as taking the system state probabilities through a short time step, the size of which varies with the size of Δ.

This iterative procedure will always converge. The speed of convergence depends on the convergence criterion, on the size of Δ, and on the accuracy of the initial estimate of the state probabilities.

With this method, it is not necessary to store the entire Q matrix (or any other matrix). Only the non-zero elements of Q need be stored, and Q is generally quite sparse (since any given state can generally go to very few other states.) For a 500-state system with six independent transition mechanisms, there will be at most 3500 non-zero elements in the 250,000-element Q matrix. This is a very reasonable number of elements to store in a digital computer and permits a relatively "fast" program. (The high degree of repetition of equal values in systematically related locations in Q, which is also characteristic of these matrices, permits further storage savings at the expense of some running time, if desired. Wallace and Rosenberg[12] discuss this approach more fully.)

A program implementing this iterative scheme was written in Fortran IV for the IBM 360 computer system. The non-zero Q-matrix elements and their coordinates are stored individually, since further compression was not required. On a System/360 model 75 computer, this program performs approximately 4,000 iterations of a 200-state model per minute; producing a complete solution (to a difference of 0.00001 between successive iterations, more accuracy than really needed) in less than five seconds for most problems.

4.2 An Elementary Multiprocessor Model

By assuming that the data bus load is sufficiently light for the bus to be available virtually immediately when needed, it is possible to develop a queuing-theory model for the multiprocessor that can be solved directly without recourse to numerical techniques. Although this model is not a very accurate representation of the multiprocessor, it is useful because

a. it is an indication of the trends to be expected;

b. its comparison with the more complex models is instructive;

c. with modifications to account for delays due to bus use, its predictions can be surprisingly close to those of the more complex models;

d. it shows the effects of varying the number of processors in the system more easily than do the more complex models.

This model assumes a multiprocessor with Q processors. The priority scheme is not important here, for since the bus is always available the priority scheme does not matter. (A major drawback of this model is, of course, that it cannot indicate the relative performance of multiprocessors with different priority schemes.) Jobs arrive exponentially, with rate λ, and terminate exponentially, with rate μ. The number of jobs that can be in the system at one time is limited to N, which may be infinite. At any time, there will be $n \leq N$ jobs in the system; if $n \leq Q$, they will all be running, and if $n > Q$, $n-Q$ of them will be stacked up waiting to start. The total number of states in the system, for finite N, is N+1: there can be from 0 through N jobs in the system. The number of jobs in the system is the only state variable. The states are numbered 0 through N.

The equations of balance have three or four forms. There is one equation describing transitions into and out of state 0 (idle system), another form for $n \leq Q$, a form for $n > Q$ and, if N is finite, a special form for n=N.

Writing $\rho = \lambda/\mu$ for convenience, the equations are

$$(n=0) \qquad P_1 = \rho P_0 \tag{4.6}$$

$$(n \leq Q) \qquad \rho P_{n-1} + (n+1)P_{n+1} = (n+\rho)P_n \tag{4.7}$$

$$(n > Q) \qquad \rho P_{n-1} + QP_{n+1} = (Q+\rho)P_n \tag{4.8}$$

$$\text{For } n=N \text{ if N finite:} \quad \rho P_{n-1} = QP_N \tag{4.9}$$

The first of these equations gives P_1 in terms of P_0. The equation for transitions into and out of state 1 has the form of Eq (4.7), and involves P_0, P_1 and P_2; either P_1 or P_0 can be eliminated with Eq (4.6), giving P_2 in terms of the other. This procedure can be continued, giving an expression for each P_n in terms of P_{n-1} or P_0. Writing the probabilities in terms of P_0, these expressions are

$$P_n = \frac{\rho^n}{n!} P_0 \quad (n<Q) \tag{4.10}$$

$$P_n = \frac{\rho^n}{Q!Q^{(n-Q)}} P_0 = \frac{Q^Q}{Q!} \left(\frac{\rho}{Q}\right)^n P_0 \qquad (n \geq Q) \tag{4.11}$$

In terms of physical meaning, ρ is the load on one processor, and should be less than Q for the system to be able to handle its load. The total system load is

given by ρ/Q as a fraction of system capacity.

The above equations indicate the following:

a. For $n<\rho$, the fraction of time that n processors are busy increases with n. Thus, in a heavily loaded system, two processors are in use more often than just one is.

b. For $Q \geq n > \rho$, the fraction of time that n processors are busy decreases with n. Thus, in a lightly loaded system with five processors, all five will be busy less often than four will be busy.

c. For $n>Q$, the state probability decreases as (ρ/Q); there are n jobs waiting to start less often than $n-1$ jobs, and so on.

An expression for the actual values of the state probabilities, rather than their relative values, can be obtained by imposing the condition that these probabilities must sum to one. This condition gives the following expression for P_0:

$$P_0 = \frac{1}{\sum_{n=0}^{Q} \frac{\rho^n}{n!} + \sum_{n=Q+1}^{N} \left(\frac{\rho^n}{Q!\, Q^{(n-Q)}} \right)} \tag{4.12}$$

The last term in the denominator can be expressed in closed form; one then obtains one of the following two expressions:

For infinite N:
$$P_0 = \frac{1}{\sum_{n=0}^{Q} \frac{\rho^n}{n!} + \frac{\rho^{(Q+1)}}{Q!\,(Q-\rho)}} \tag{4.13}$$

For finite N:
$$P_0 = \frac{1}{\sum_{n=0}^{Q} \frac{\rho^n}{n!} + \frac{\rho^{Q+1}}{Q!\,(Q-\rho)} \left[1 - \left(\frac{\rho}{Q} \right)^{N-Q} \right]} \tag{4.14}$$

Many informative items can be obtained from these probabilities. For example, the fraction of time during which there is a job unable to start because there are no free processors is given (for infinite N) by

$$P_{waiting} = \frac{1}{1 + \frac{Q!\,(Q-\rho)}{\rho^{(Q+1)}} \sum_{n=0}^{Q} \frac{\rho^n}{n!}} \tag{4.15}$$

The mean time that jobs have to wait is also easily obtained. The mean waiting time must, for a steady state, be given by the mean number of jobs unable

to start multiplied by the mean interval between job arrivals, λ. If time is non-dimensionalized dividing the mean delay by λ, the non-dimensional mean delay is given by

$$D = \sum_{n=Q+1}^{N} (n-Q) P_n = \frac{P_0 \rho^Q}{Q!} \sum_{i=1}^{N-Q} i \left(\frac{\rho}{Q}\right)^i \qquad (4.16)$$

which holds as written for finite N; and, with the obvious change in the upper limit of the summation, for infinite N.

Similar expressions can be obtained for the variance in the delay time, etc.

4.3 The Multiprocessor as a Markov Process

For a more accurate model of the multiprocessor than the one developed in the preceding section, the Markov process approach and formalism were used. Four state variables were defined for the multiprocessor. They are

x_1: the number of jobs being executed,

x_2: the number of running jobs waiting to use the bus,

x_3: the state of the bus,

x_4: the number of jobs waiting to start.

The number of jobs being executed can take on any value from zero to the number of processors in the system. A job is considered as "being executed" in this sense if it occupies a processor in any way; it can be using the bus, performing calculations, or waiting idly to use the bus to return data.

The number of currently-running jobs that are waiting to use the bus can vary from zero up to one less than the number of jobs that are running. (The running jobs cannot all be waiting to use the bus, since if they all wanted to use it one of them would in fact be using it.)

The state of the bus is, for a simple model of the system, 0 or 1, representing the bus being free or in use. More complex models, incorporating non-exponential bus use distributions, can have additional possible values for this variable. These distributions and models are discussed more fully in Section 4.5

The number of jobs waiting to start can vary from zero up to an arbitrary maximum related to the number of states with which the analyst and his computer are prepared to cope. Equation (4.11) provides a guide to the maximum value

required as a function of system load, and to the improvements in accuracy to be gained by going to higher maximum values. Generally, an upper limit of 2 or 3 proves adequate for moderately loaded multiprocessors; beyond a value of 5, the gains in accuracy do not justify (it is felt) the added model complexity.

4.4 A Simple Multiprocessor Markov Model

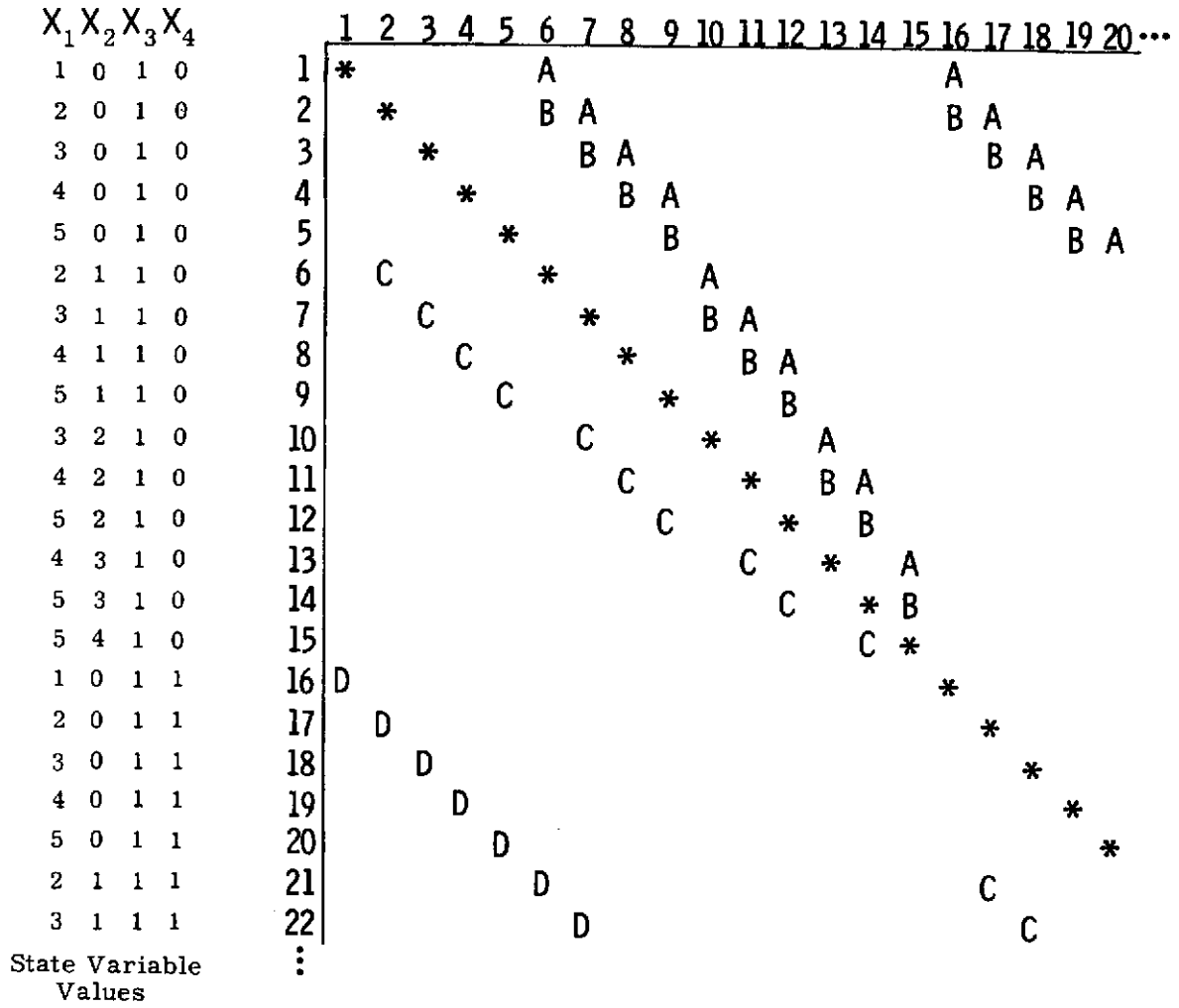
The simplest Markov model of the multiprocessor assumes exponential job arrivals, exponential job durations, and an exponential distribution of bus use requirements. For a system with five processors, and a maximum of two jobs waiting to start at one time, such a model has 53 possible states; for each additional waiting job allowed, 16 states are added to this number. A representative part of the transition matrix for this model is shown in Fig. 4-1, and the output from a typical computer run is reproduced as Fig. 4-2.

The types of events that can cause transitions in this model are similar to the events that can cause transitions in the more complex models. They are the following:

1. a job comes due for execution,
2. a running job requires the bus,
3. a job using the bus releases it.

The nature of the transition that takes place on the occurrence of one of these events depends both on the type of event and on the state of the system before the event. For the first type of event (a job wants to start execution) the type of transition depends on whether the bus is busy or not, and on whether there is a free processor to accept the job or not. If both these conditions are met, the job can start: x_1 is incremented by one, to indicate that one more job is running, and x_3 goes from 0 (bus free) to 1 (bus busy). If one of these conditions is not met, x_4 is incremented by one to indicate that an additional job is waiting to start as soon as possible, and the other state variables are unchanged. If, in addition, x_4 has already reached its maximum allowable value (2 in the simple model), the job is ignored. If the model is to represent the system accurately, the frequency with which this happens must be held to a small value.

An example of a transition in which a job comes due for execution and starts immediately is the transition from state (2, 0, 0, 0) to state (3, 0, 1, 0). An example of a transition in which the new job must wait because the bus is being used by another job is from (3, 0, 1, 0) to (3, 0, 1, 1). A transition in which the job must wait because there are no free processors is from (5, 0, 0, 1) to (5, 0, 0, 2).



Transition Matrix
(transitions are from state on left to state on top)
(complete matrix is 53×53)

Transition Types:

- **A:** job terminates
- **B:** job releases bus and starts calculations
- **C:** job finishes calculations and needs bus
- **D:** job comes due for execution
- ***:** diagonal entry chosen to make column sum to one

Fig. 4-1 State transition matrix (incomplete) for 53-state model.

```
BASIC MODEL ANALYSIS

EXPONENTIAL JOB ARRIVALS, RATE 135.000000 PER SECOND
EXPONENTIAL JOB DURATIONS, TERMINATION RATE 100.000000 PER SECOND
EXPONENTIAL BUS USE, RELEASE RATE1800.000000 PER SECOND

PROCS IN USE     PCT OF TIME

      0            22.0109
      1            33.2683
      2            25.1398
      3            12.6776
      4             4.7835
      5             2.1199


JOBS PAST DUE    PCT OF TIME

      0            98.0464
      1             1.4916
      2             0.3006
      3             0.1068
      4             0.0546


JOBS WANT BUS    PCT OF TIME

      0            98.7057
      1             1.1793
      2             0.1059
      3             0.0087
      4             0.0004

BUS USED  14.99% OF THE TIME
TOTAL PROCESSOR-BUSY TIME: 151.31%
LOAD FRACTION:  30.26%
ACTUAL COMPUTATION TIME: 134.91%
COMPUTATIONAL EFFICIENCY:  89.16%
TOTAL PRODUCTIVE TIME: 149.91%
OVERALL EFFICIENCY:  99.07%
MEAN DELAY BEFORE ACCEPTANCE:    0.000195 SECONDS
MEAN WAIT FOR BUS WHEN RUNNING:    0.000105 SECONDS
MEAN NO. OF JOBS WAITING TO START:    0.026317
MEAN NO. OF JOBS WAITING TO USE BUS:    0.014185
INFORMATION TRANSFER EFFICIENCY:  91.41%
WASTED TIME:    9.40% OF BUS USE TIME

VARIANCE:    0.044597
  VARIANCE:    0.016679

STD DEVIATION:    0.211180
  STD DEVIATION:    0.129140

END OF STATISTICAL SUMMARY
```

Fig. 4-2 Computer output from the Markov model run.

The actual entry in the transition matrix is the rate at which this event takes place while the system is in a state such that it can take place. (The event "running job needs the bus" cannot take place if no jobs are running, or if the only running job is already using the bus.) If an average of 100 jobs arrive per second, the mean arrival rate is 100 sec^{-1}, and so on. This value of 100 is the entry in the transition intensity matrix Q, and would be further scaled as described in Section 4.1. Similarly, if jobs use the bus with a mean use duration of 2 msec, the transition intensity matrix entry for those transitions caused by bus release would be 500 sec^{-1}.

The second type of transition is caused by a running job requiring the use of the bus, to return data to central memory (phase 2 of bus use). If the bus is free, the job obtains it immediately, such as in the transition from (3, 0, 0, 0) to (3, 0, 1, 0). If it is not free, the job requesting the bus must wait; this is indicated as a state change by incrementing x_2 by one. Such a transition is the one from (4, 0, 1, 0) to (4, 1, 1, 0).

In the case of either of the above types of events, the state after the transition is defined completely by the state before the transition and the type of event causing the transition in question. This is not true for the third type of transition, in which the job using the bus releases it. The description of the state of the system does not state whether the job using the bus is obtaining data from memory (phase 1 of bus use) and will therefore continue to occupy a processor after releasing the bus, or is returning data to memory (phase 2 of bus use) and will free its processor after releasing the bus. It is necessary to make an assumption about the relative probabilities of each event occurring. Since every job must go through each phase exactly once, it can be assumed that each event is of equal probability.

If there are no jobs waiting to start and no jobs waiting to use the bus, the state of the bus (x_3) will go to 0 when the bus is released. The number of busy processors will either remain unchanged or be reduced by one, with equal probability of either as discussed in the preceding paragraph.

If there are jobs waiting, however, the state of the bus will not go to zero—since the bus will be "grabbed" immediately by one of the waiting jobs. In addition, a job will be removed from one of the waiting lists. For example, if there is a job waiting to start, x_4 will be reduced by one. If the state of the system is (3, 0, 1, 1) and the bus is released, the system will go with equal likelihood (as discussed above) to state (4, 0, 1, 0) (if the job releasing the bus does not terminate) or to state (3, 0, 1, 0) (if it does terminate). In either case, the size of the waiting list

is reduced, and the bus remains busy. The procedure if there are running jobs waiting to use the bus (x_2, rather than x_4, non-zero) is analogous.

If there are both jobs waiting to start and running jobs waiting to use the bus, as in state (3, 1, 1, 1), the choice of the next state depends — as it does in a real multiprocessor — on the priority scheme of the system being analyzed. If the job stack has priority on the bus, the job waiting to start will always be chosen as the next bus user, if there is a free processor; running jobs will wait. If the job stack does not always have priority, as in priority scheme 4, it is impossible to determine unambiguously from the state of the system which user will obtain the use of the bus, and probabilities must be assigned. The probabilities can be assigned accurately from knowledge of the multiprocessor structure and of the number of jobs running. For example: if there is a job waiting to start, and one job waiting to use the bus, and the job stack is connected at one point in the processor ring, it is equally likely that the job stack will be polled before the one waiting processor or that the processor will be polled before the job stack. The two transition probabilities are, in this case therefore, equal to each other. Were there two running jobs waiting to use the bus, there would be a two-thirds probability that a processor would be polled before the job stack.

The simple model requires that the user specify, in order that the transition rates may be calculated, three parameters: the mean rates of the three exponential processes involved. These are expressed in units of inverse time (so many job arrivals per second, etc).

It is possible to specify the system more compactly by using non-dimensional parameters. The ratios between any two pairs of mean times are particularly simple to use, and with these ratios only two numbers need be given. (The same result can be obtained by selecting the unit of time such that one of the original parameters has a value of unity, and then expressing the others in the same unit system. Only the other two parameters need then be supplied.) This was not done in order to permit expressing Markov input parameters in the same dimensional terms, having intuitive meaningfulness, that characterize the input to and the output from the simulation. This type of non-dimensionalization is, however, of great usefulness in the analysis of the results.

4.5 Additional Multiprocessor Markov Models

Six major assumptions, or limitations, are inherent in the simple 53-state model described in the previous section. They are

1. exponential job durations,

2. exponential job arrivals,
3. exponential bus use demand distribution,
4. equal bus use distributions, phases 1 and 2,
5. limit of two past-due jobs waiting to start,
6. state ambiguity following bus release.

Each of these can be removed or relaxed, at the cost of complicating the Markov model by the addition of more states.

More complex models were developed, and are the topic of this section.

The second model of the multiprocessor (the simple one discussed in the previous section is the first) relaxes the third assumption above: that the bus use distribution is exponential. The bus use distribution used is based on the data from the lunar landing job analysis. This analysis shows that very long and very short bus use demands occur more often than would be predicted by an exponential model. To fit the data more closely, a "two-term hyper-exponential" distribution, as described by Morse[19], was used. Such a model consists, conceptually, of two parallel exponential processes, one of which is selected each time the bus is to be used. One of these has a mean time larger than the mean time of the original process, while one has a smaller mean time. (There is, of course, no implication that there are physically two data busses in the multiprocessor.) The "non-exponentialness" of the process is described by a parameter σ, which is the probability with which the long-duration branch of the process is chosen. It takes on values from 0.5 (exponential process) to 0 (highly non-exponential process, one branch has infinite mean time). The probability that the interval between two successive events, in a process so distributed, is greater than t, is given by

$$S_o(t) = \sigma e^{-2\sigma\mu t} + (1 - \sigma) e^{-2(1-\sigma)\mu t} \qquad (4.17)$$

Figure 4-3 gives an example of such a distribution, with the "non-exponentialness" parameter σ having a value of 0.1.

In such a model, the "bus use state variable" x_3 can take on three values: 0, to indicate that the bus is free; 1, to indicate that it is in use and that the conceptual "short-duration branch" is busy; and 2, to indicate that it is in use and that the "long-duration branch" is busy. For a five-processor system with a limit of two jobs waiting to start, such a model has 98 states. It is described by the parameters that described the simple model, plus the new parameter σ giving the "non-exponentialness" of the bus use distribution.

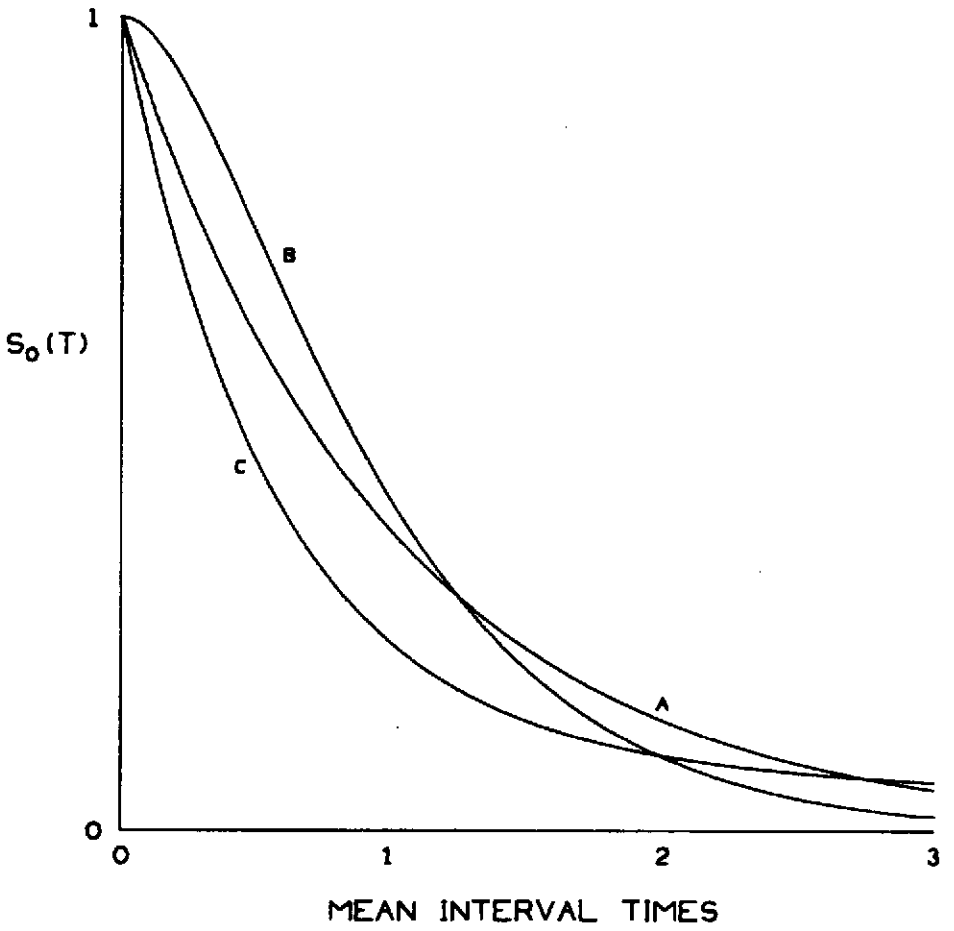


Fig. 4-3 Inter-event distribution.

This type of distribution was matched, using a least-mean-square fitting procedure, to the lunar landing bus use data. The best fit was obtained for $\sigma = 0.1694$. Figure 4-4 shows such a curve superimposed on the actual data (the data curve has been previously pictured by itself as Fig. 2-1).

The third Markov model of the multiprocessor eliminates the sixth limitation of the simple model: the state change ambuiguity that occurs when the bus is released. In this new model, the value of the bus use variable x_3 indicates explicitly whether the job using the bus is in phase 1 of bus use (in which case it will not release a processor when it releases the bus) or in phase 2 (in which case it will free a processor). The variable x_3 can thus take on values of 0, 1 or 2; 0 indicates that the bus is free, and 1 and 2 distinguish between phases of bus use in the system being modeled. Such a model, with a limit of two past-due jobs and five processors, has 98 states, as did the previous model. To distinguish it from the previous model, where confusion might arise, it will be referred to as the "98-state two-phase model".

The development of a model such as this one is tantamount to a statement that the assumption made with respect to system behavior on bus release in the simple model is inaccurate. It was assumed there that bus releases are equally as likely to result in processor release as not to. This is clearly globally true. It is not necessarily, though, locally true for each individual state. It is possible to reason that it should not be. The following argument will suffice to indicate that this is so:

Consider the state (1, 0, 1, 0) where the variables have the meanings associated with the simple model described in Section 4.4. This state represents a system executing one job, which is using the bus. The bus use may be either phase 1 or phase 2; the system state does not supply that information.

This state could have come about as the result of a transition from any of four previous states:

a. State (0, 0, 0, 0) — system idle. A job arrives for execution and uses the bus to obtain data. In this case, the current bus usage would be phase 1.

b. State (1, 0, 0, 0) — one job running. It terminates its calculations and uses the bus to return data: phase 2.

c. State (2, 1, 1, 0) — two jobs running, one using the bus and the other waiting to use it. The job using the bus releases it and terminates execution. The waiting job obtains the bus: phase 2.

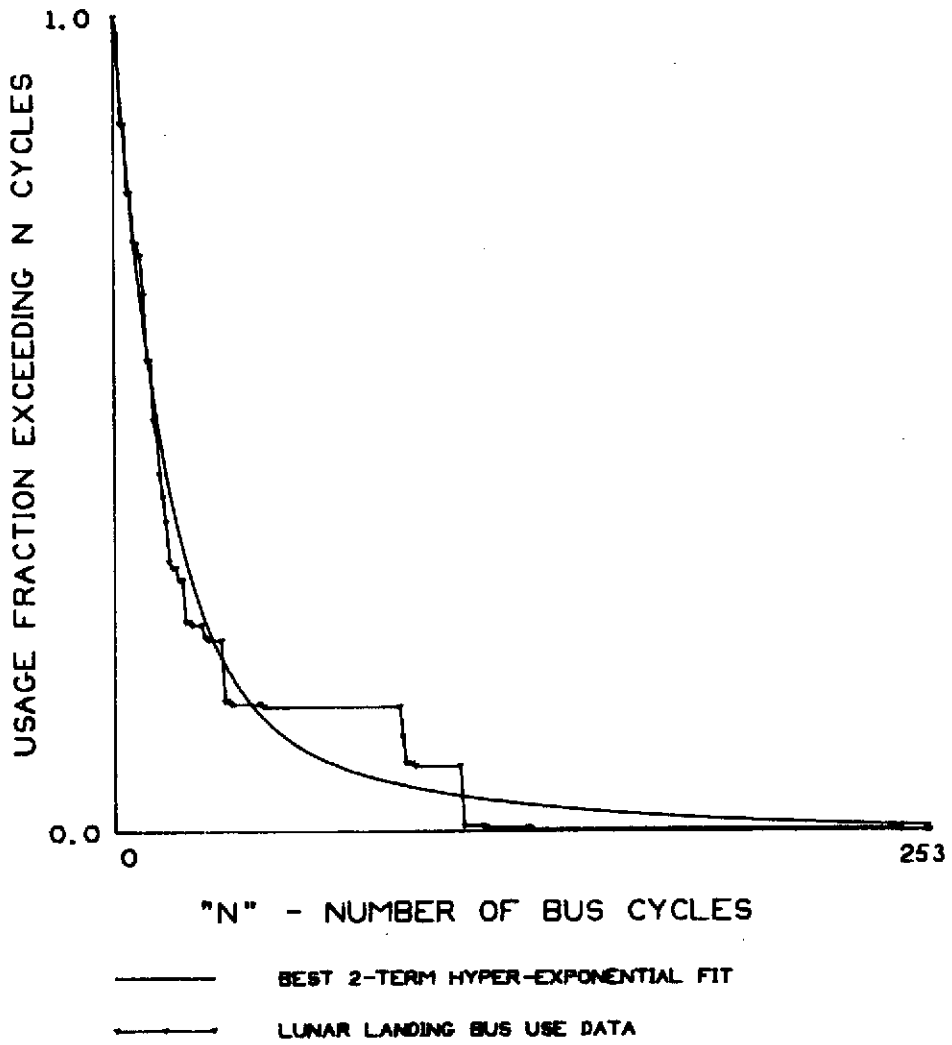


Fig. 4-4 Bus use distribution.

d. State (1, 0, 1, 1) — one job running and using the bus, another waiting to start. The job using the bus releases it and terminates execution. The new job obtains the bus: phase 1.

Let us now call the mean job duration 1/T, the mean rate of job arrival A, and the mean duration of a bus usage 1/R. (Reciprocals were used here for two of the parameters because the state transition equations involve parameters having units of inverse time.) (Using just three parameters assumes equal bus use durations for phases 1 and 2, but this does not affect the validity of the argument.) If the bus use mean duration is 1/R, then bus release occurs (in those states in which the bus is busy) at mean rate R, and the rate at which jobs release the bus and terminate must be $\frac{1}{2}$R according to the "equal phase probability" assumption of the simple model. Again, according to this assumption, if the two phases are to be equally likely in the state (1, 0, 1, 0) under consideration, we must have equal transition rates into this state from the state transitions that will bring about phase 1 bus use and those that will bring about phase 2 bus use. This condition can be expressed as follows:

$$\tfrac{1}{2}Rp\,(2,1,1,0) + Tp\,(1,0,0,0) = \tfrac{1}{2}Rp\,(1,0,1,1) + Ap\,(0,0,0,0) \qquad (4.18)$$

where p(w, x, y, z) represents the probability, in the steady state, of the system being in state (w, x, y, z). Rewriting for convenience,

$$\tfrac{1}{2}R\left[p(2,1,1,0) - p\,(1,0,1,1)\right] + Tp\,(1,0,0,0) - Ap\,(0,0,0,0) = 0 \qquad (4.19)$$

Now let us consider the balance equation for this state. Equating transition rates into and out of the state yields

$$\tfrac{1}{2}R\left[p\,(2,1,1,0) + p\,(1,0,1,1)\right] + Tp\,(1,0,0,0) + Ap\,(0,0,0,0) = (A+R)p\,(1,0,1,0) \qquad (4.20)$$

We may non-dimensionalize the coefficients in these equations by setting R' = R/A, T' = T/A; Eq (4.19) and (4.20) then become

$$\tfrac{1}{2}R'\left[p\,(2,1,1,0) - p\,(1,0,1,1)\right] + T'p\,(1,0,0,0) - p\,(0,0,0,0) = 0 \qquad (4.21)$$

$$\tfrac{1}{2}R'\left[p\,(2,1,1,0) + p\,(1,0,1,1)\right] + T'p\,(1,0,0,0) + p\,(0,0,0,0) = (1+R')p\,(1,0,1,0) \qquad (4.22)$$

It is now possible to eliminate T' from this pair of equations by subtracting Eq (4.21) from Eq (4.22). This yields

$$R'p\,(1,0,1,1) + 2p\,(0,0,0,0) = (1+R')\,p\,(1,0,1,0) \qquad (4.23)$$

which relates the probabilities of three states through an expression involving only one of the two independent variables that define the system. A similar, but some-

what more complex, equation could have been written after eliminating R' from Eq (4.21) and (4.22) rather than T'.

Such an equation in one parameter can be written for every state in the system that uses the bus. These equations will, as does Eq (4.23), also involve the probabilities of the states in which the bus is free. Thus, they provide a set of relationships among the system state probabilities not involving T', the mean job duration.

It is clearly absurd to suggest that the system state probabilities should not depend on the mean job duration. However, this conclusion is inescapable if one assumes that bus releases are divided equally, in each state, between jobs that do terminate execution and jobs that do not. This assumption must, therefore, be discarded. The desirability of having a multiprocessor model that does not include this assumption as part of its basic structure follows immediately.

This two-phase model is described by the same three parameters that described the simple model: job arrival rate, mean job duration (reciprocal), and mean bus use time (reciprocal). It would have been possible to use two mean bus use times, one for each phase of bus use, without complicating the model or adding additional states. This was not done because this model was developed, chronologically, after the next one discussed below. Results from that model had already indicated that no appreciable gain in accuracy was obtained by using two mean times, since for actual programs (lunar landing data) the mean times for phase 1 and phase 2 of bus use are quite close. It was therefore decided to retain the external simplicity of the simple model's input parameters.

The fourth model of the multiprocessor introduces a further refinement into the assumed distribution of bus use by jobs. It combines the improvements of the above two models: it incorporates two separate, and different, two-term hyper-exponential distributions, one for each phase of bus use. It thus removes the fourth restriction of the simple model, by permitting the user to specify different mean times and degrees of non-exponentialness for the distributions characteristic of phase 1 and phase 2 of bus use.

In this model, x_3 can take on five values: 0 indicates that the bus is free, 1 and 2 correspond to the two branches of the hyper-exponential process representing phase 1 of bus use, and 3 and 4 are analogous for phase 2. The model, with up to two jobs waiting to start and five processors in the system, has 188 states. It is described by six parameters: the job arrival rate, the job duration, and the two parameters descriptive of each hyper-exponential distribution.

These distributions were also matched with a least-mean-squares fitting procedure to the lunar landing data. The best fit was obtained when: the mean bus use duration for phase 1 was 1.155 times the mean duration of all bus uses; the phase 1 distribution had $\sigma = 0.1906$: the mean bus use duration for phase 2 was 0.845 times the overall mean duration; and the phase 2 distribution had $\sigma = 0.1472$.

The fifth Markov model introduces non-exponential job durations. A two-term hyper-exponential model similar to the ones used above for describing bus usages was used. This model was created as a modification of the basic 53-state model, independently of the other changes, both to isolate the effect of this change and to simplify the creation of the transition matrix (which can be a tedious clerical task, though capable of some automation). Such a model, with five processors and a maximum of two past-due jobs waiting to start, has 243 states. It is described by the basic parameters plus the additional one needed by the non-exponential job duration distribution. The job durations are, in the lunar landing data, very highly non-exponential; the best-fitting value of $\dot{\sigma}$ was found to be 0.007. It is felt that use of a higher-order non-exponential distribution (incorporating more states) would provide better modeling of the distribution of execution times in this particular job set.

A sixth model was developed to study the effect of non-exponential job arrivals on system behavior. It uses a job arrival distribution with reduced variance, to determine the extent to which this improves system efficiency. The distribution used, which was again taken from Morse [19], is known as "2-Erlang" (after an engineer with the Copenhagen telephone company, who did pioneering work in queuing theory shortly after the turn of the century). This distribution consists conceptually of two exponential channels in series. Each has a mean rate of twice the mean rate of the overall process, so that arrivals occur at the correct mean rate after passing through both stages. The effect of the double timing channel is to concentrate the inter-arrival intervals nearer the mean interval. (A uniform interval may be approached as closely as desired by using a sufficiently large number of exponential channels in series.) The mathematical form of this distribution is given by:

$$S_o(t) = (1+2\mu t)e^{-2\mu t} \qquad (4.24)$$

Such a curve is shown in Fig. 4-3.

This change also was made to the simple 53-state model; it was felt that this would provide results sufficiently indicative of the trends to be expected, without introducting the complication of large numbers of states. Such a model has 106 states, and is described by the same parameters as describe the basic 53-state model.

Finally, the restrictions on the number of jobs that can be waiting to start were relaxed by modifying some of the above models so as to increase the allowable queue length. The basic 53-state model, which permits two waiting jobs, was enlarged into a 69-state model with a maximum queue length of 3, and into an 85-state model with a maximum queue length of 4. The 98-state two-phase model was enlarged into otherwise identical models having 129, 160 and 191 states. This provides, respectively, 3, 4 and 5 past-due jobs waiting to start.

CHAPTER 5

RESULTS

5.1 Introduction

This chapter discusses the results obtained from the simulation and the queuing-theory analyses of the multiprocessor. It is divided into four sections (beyond the present one), each dealing with a different phase of the results.

The first section deals with the selection of a "minimum adequate" Markov model of the multiprocessor. The models of Section 4.5, which remove the limitations of the basic model of Section 4.4, are compared with each other (and with simulation results, where these aid in the selection process). One Markov model is selected, and is used in subsequent sections where appropriate.

The second section discusses the effect of arrival scheduling on system efficiency. Markov models incorporating exponential and 2-Erlang job arrivals are compared with each other and with simulation results.

The next section uses the elementary multiprocessor model of Section 4.2 and the simulation to evaluate the effect of varying the number of processors in the system.

The final section uses the Markov model selected in Section 5.2 and simulation results to evaluate the effect of a change in system architecture. Priority schemes 1 and 4, as defined in Section 1.2, are compared.

Conclusions from these results are presented with the results from which they are derived, and are summarized in Chapter 6.

5.2 Selection of a Markov Model

5.2.1 Selection of a bus use model.

Three statistical distributions of bus use demands are incorporated in the Markov models discussed in Sections 4.4 and 4.5. They were the exponential distribution, the two-term hyper-exponential distribution, and the two separate two-term hyper-exponential distributions (one for each phase of bus use).

To select one of these, a series of computer runs was made with each model, and the results compared. Two system performance parameters were used in the comparison: "information transfer efficiency" and "job acceptance delays".

Information transfer efficiency was discussed in Section 3.4 and defined in Eq (3.1). The definition was as follows:

$$ITE = \frac{T_b}{T_b + T_w} \tag{5.1}$$

where T_b is the sum, over a run, of the time during which the data bus was in use, and T_w is the sum, over a run, of the time during which processors were waiting for the bus. (Since T_w is summed over all processors, it can exceed the total time of the run.)

Job acceptance delay is the interval between the time at which a job should start — based on its job stack entry — and the time at which a processor sends a job acceptance message for it, corrected for the one bus cycle during which the bus request message is being transmitted. (With this correction, a job accepted on time will have zero delay; without it, such a job would have a delay equal to one bus cycle time under the original definition of delay.) This delay is a measure of the amount of time a job might have to wait before starting, and is important because the sampled-data nature of many aerospace calculations makes time a critical factor. The delay is non-dimensionalized by dividing it by the mean interval between job arrivals: if the mean delay is 1 msec, and jobs arrive at a rate of 100 per second so that the mean inter-arrival interval is 10 msec, the non-dimensionalized delay is 0.1.

In the series of computer runs used for the comparison, job duration and bus use duration parameters were selected to maintain an average ratio of 9 units of computing time to 1 unit of bus use time per job. The Markov model can then be defined completely by one load-related parameter, which was used as the independent variable. The percent of time that the bus is in use was chosen since it has a clear physical meaning; any other could have been.

Figure 5-2 shows no significant differences among the mean job acceptance delays as predicted by the three models. From the standpoint of calculating delays, therefore, they may be regarded as equivalent.

Figure 5-1 does show a difference between, on the one hand, the results obtained from the simple 53-state model, and the results obtained from the two hyper-exponential models on the other hand. The two hyper-exponential models

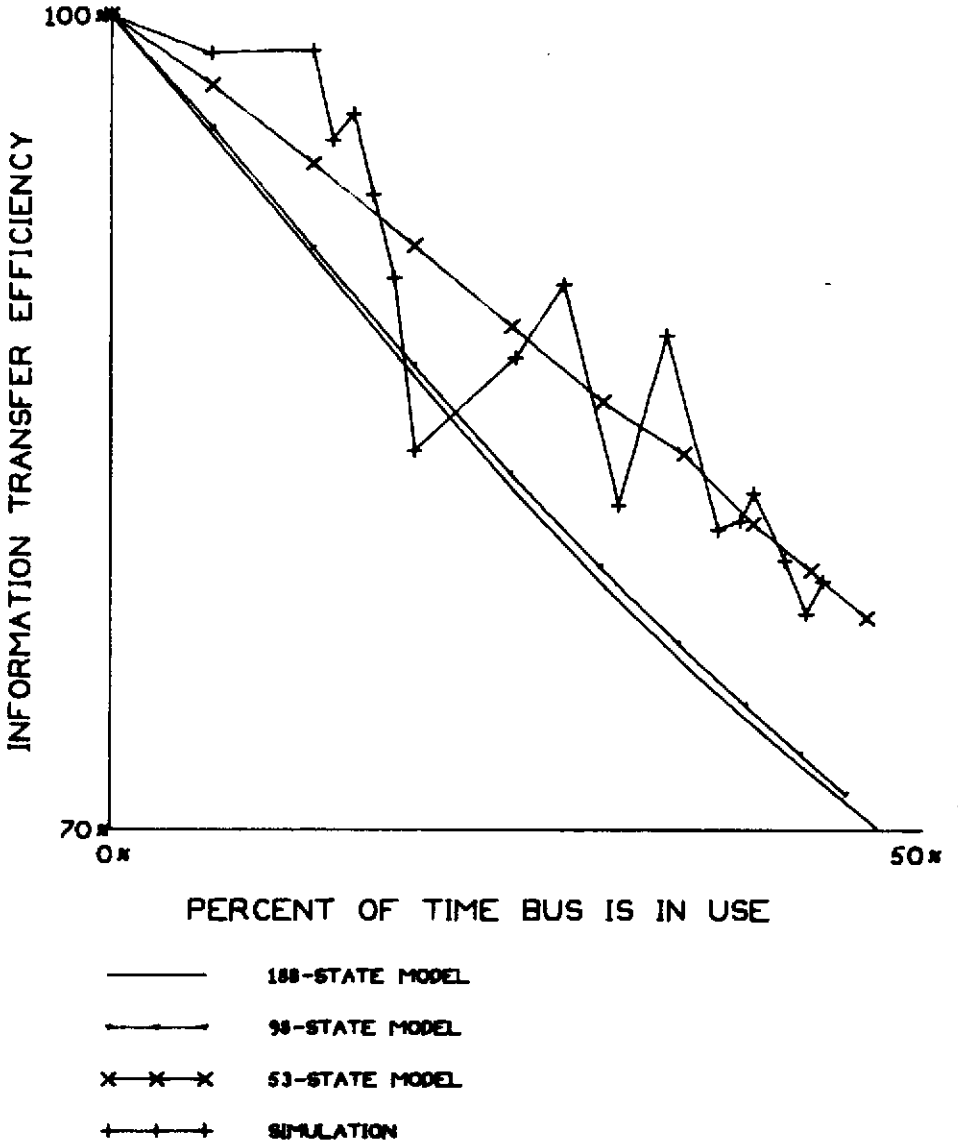


Fig. 5-1 Markov model comparison.

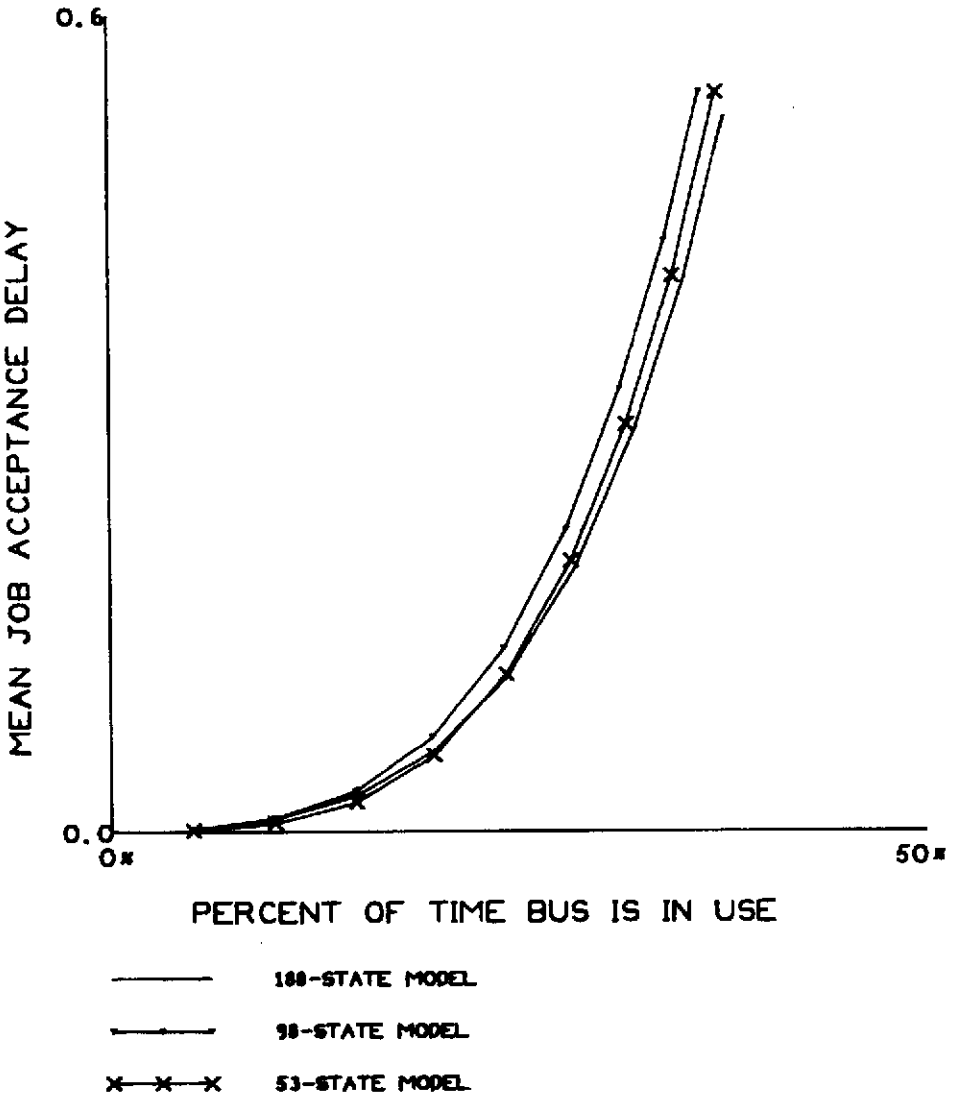


Fig. 5-2 Markov model comparison.

differ but little between themselves, as might be expected from the closeness of the mean bus use times characteristic of each phase of bus use in the lunar landing program data. The simple 53-state model, however, predicts appreciably higher efficiencies than do the others.

Interestingly, the 53-state model predicts efficiencies that are quite close to those actually measured in the simulation. The predictions of the more complex, and presumably more accurate models, are not nearly so close to simulation results.

This phenomenon can be explained as follows: A hyper-exponential distribution predicts more very long bus usages than does an exponential distribution. This is, in fact, characteristic of the lunar landing program data. In a Markov analysis, these long usages will interfere with each other, lowering efficiency below what would be predicted by an exponential model. This, in fact, did happen. In the simulation, though — and in the actual situation — these long usages are not found at random, but are found on successive executions of the same job, or at the two bus-use phases of the same long job. Thus, they cannot interfere with each other. Here is a case where increased accuracy in the mathematical model actually reduces the accuracy of the results, because of a counterbalancing effect in the real situation that cannot be introduced into the Markov model. (It could be, but only at the expense of a great many additional states.)

The basic, exponential model of bus use will therefore be used in modeling the multiprocessor. It is felt that it predicts delay times as well as do the other models, and for the reason discussed just above, predicts information transfer efficiency better than do the other two models.

It is interesting to note the relationship between the information transfer efficiencies predicted by the Markov models and those actually recorded in simulated multiprocessor activity. In contrast to the predicted smooth curve, the data from the simulation give a very jagged graph.

This effect can be attributed to the interactions among the jobs being executed. When a specific set of jobs is performed at specified intervals, it is inevitable that there should be interactions that are statistically quite unlikely but that, in fact, occur with appreciable frequency. This can result in the odd phenomenon of a slow system executing a given set of jobs with higher efficiency than a faster one: one job might be slowed down just enough so that its request for bus use, which in a faster system would have come when another job was using the bus, now comes when the bus is free.

The validity of this explanation is substantiated in Fig. 5-3 and 5-4. Figure 5-3 shows the result of applying a random variation to the time at which jobs are

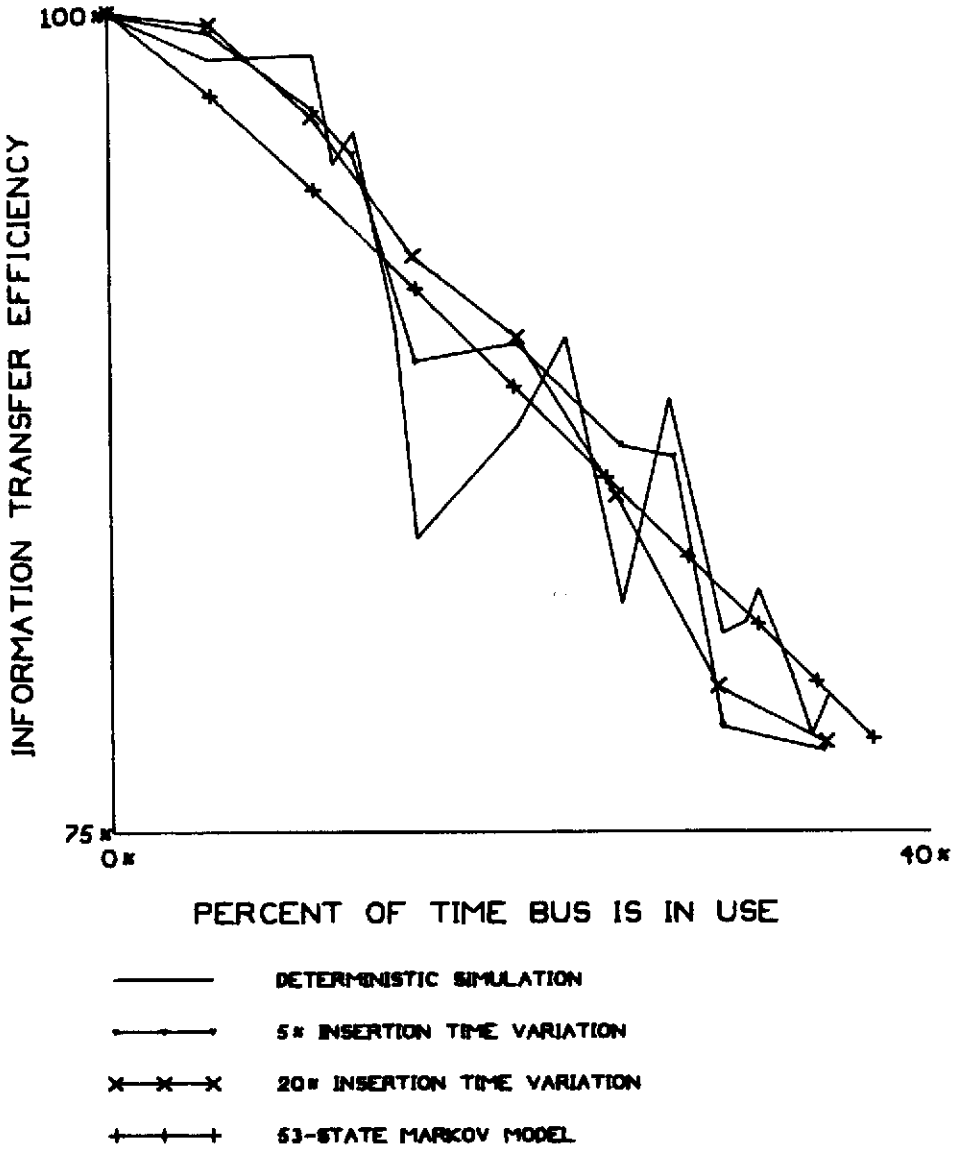


Fig. 5-3 Randomized job requests.

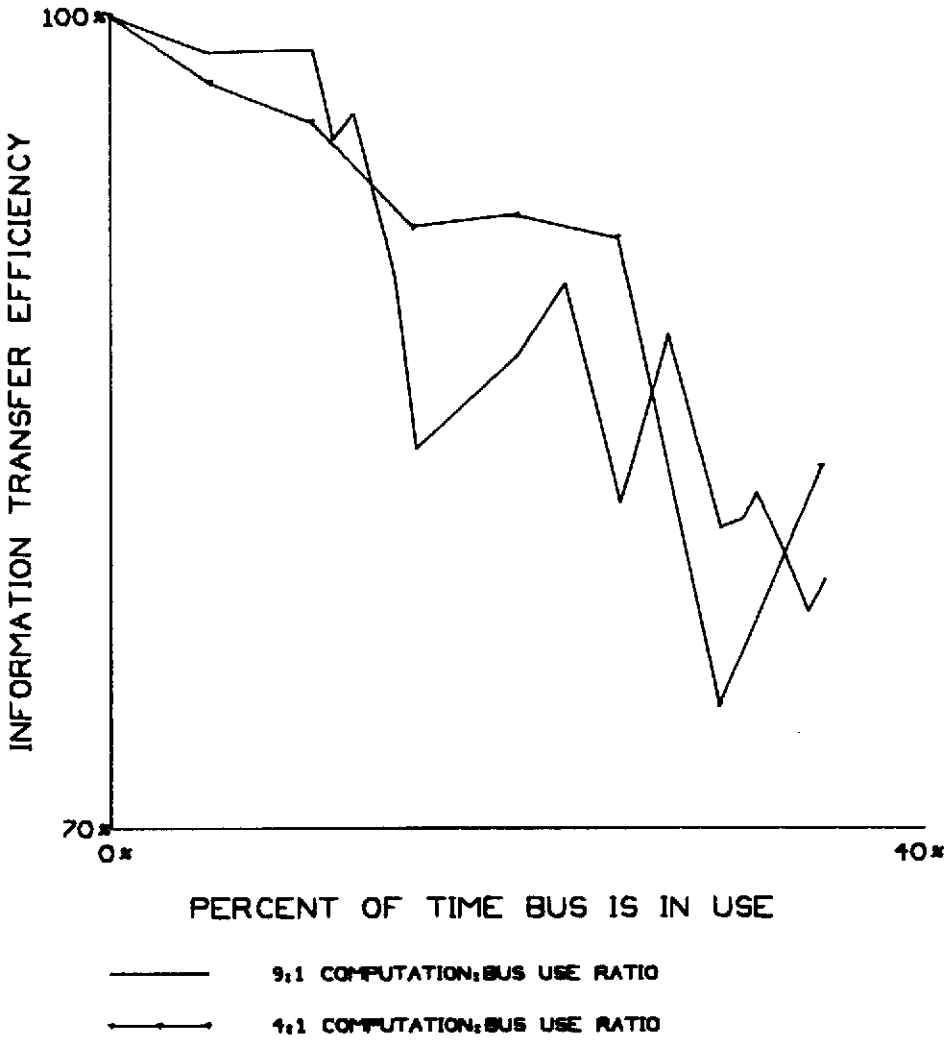


Fig. 5-4 Effect of computation speed.

executed. This variation was applied in the following way: whenever a job inserted another job into the job stack, the time specified for execution of the second job was perturbed by a small amount. This perturbation was applied to the interval between the time at which the insertion took place and the time at which the inserted job was to be executed. It was taken from a random distribution uniform over a range expressed as a percentage of this interval. Two sets of runs with this modification were made: one with 5% perturbations (the new job would be executed after an interval ranging from 95% to 105% of the originally specified interval) and one with 20% perturbations (the interval until execution was from 80% to 120% of that requested). The results of these modified simulation runs are shown in Fig. 5-3, which plots information transfer efficiency against system load (expressed, again, as the fraction of time that the bus is in use).

With these random perturbations applied to the times at which jobs are executed, the chance of having statistically improbable interactions repeated a large number of times goes down drastically. It would be expected that the peaks and valleys of the original curve would be smoothed out, and this, in fact, does occur.

Figure 5-4 shows the effect of a different change to the simulation: the processors were speeded up, while the speed of the bus was left unchanged. It would be expected that the improbable interactions would still occur, but would be different; the curve should still be jagged, but with different peaks and valleys. The simulation runs were made with processors able to perform their calculations in four times the data transfer time, at a 4:1 ratio of computing time to bus use time, rather than the 9:1 ratio used in the other simulation runs. The expected effect does take place.

In a real situation, random perturbations could not be applied, and the speed of the processors would be fixed. Statistically unlikely interactions would be the rule, not the exception. It is reasonable to assume that, in this case, operating efficiency would fall somewhere in a band on each side of the mathematically predicted curve. For efficient use of a multiprocessor, it would be desirable to ensure that the actual system performance lay in the upper half of this band; that is, jobs should be scheduled so as to interfere with each other as little as possible. It is unreasonable to place the burden of such scheduling on the user, who should be free to concentrate on the content of the jobs. This scheduling must therefore be performed by a scheduling program at the time jobs are prepared for input into the computer. (The loss of time involved with trying to do this dynamically, as the programs are running, probably outweighs the loss of efficiency from not doing it at all.) This scheduling program would have to be given some type of specification as to the allowable range of iteration rates, insertion times, etc., relevant to the

jobs in question. It would then prepare a schedule satisfying the constraints and utilizing the system efficiently. Just how to do this remains an open question for study. It does appear that a small amount of effort expended in this area could result in appreciable gains in system efficiency.

5.2.2 Bus-use phase fidelity.

In the previous chapter it was mentioned that the basic 53-state model incorporates an ambiguity in the state change that takes place when the bus is released. It is impossible in this model to determine whether the release of the bus means that a job has terminated, thus freeing a processor, or is through obtaining data and ready to start calculations, thus not freeing a processor. This is because the state description in this model does not carry enough information to define the phase of the current bus use.

This information can, at the expense of including additional states in the model, be retained. If it were so retained, the transition occurring at bus release would no longer be ambiguous.

The two-phase bus use model discussed in Section 4.5 was developed to determine whether the inclusion of this additional information results in improved accuracy in the results. Performance predictions of this model are compared with those of the simple 53-state model in Fig. 5-5 through 5-7.

Figure 5-5 shows the effect of improved bus-use phase fidelity in the model on predictions of information transfer efficiency. This effect is clearly negligible.

Figure 5-6 is similar in intent to Fig. 5-2, and shows the effect of the change in the Markov model on predicted job starting delays. The difference between the predictions of the two models, while noticeable, is small.

Figure 5-7 shows the fraction of time that the 2-entry job queue, a feature of the simpler models, is full. The relevance of this parameter is as follows: if job arrivals are exponential (as they are assumed to be in these models), so that jobs are equally likely to arrive at any time, this represents the fraction of jobs that are skipped over in the Markov analysis. This skipping over jobs represents a major potential source of inaccuracy in the analysis, and should be kept as low as possible. The more accurate bus use model (the two-phase model) has, for a given system load, a full queue for an appreciably smaller fraction of the time.

Figure 5-8 is similar to Fig. 5-7. It shows the percent of time that a four-entry job queue is full, as obtained from computer runs using models able to accommodate more past-due jobs, but otherwise identical to those used to obtain

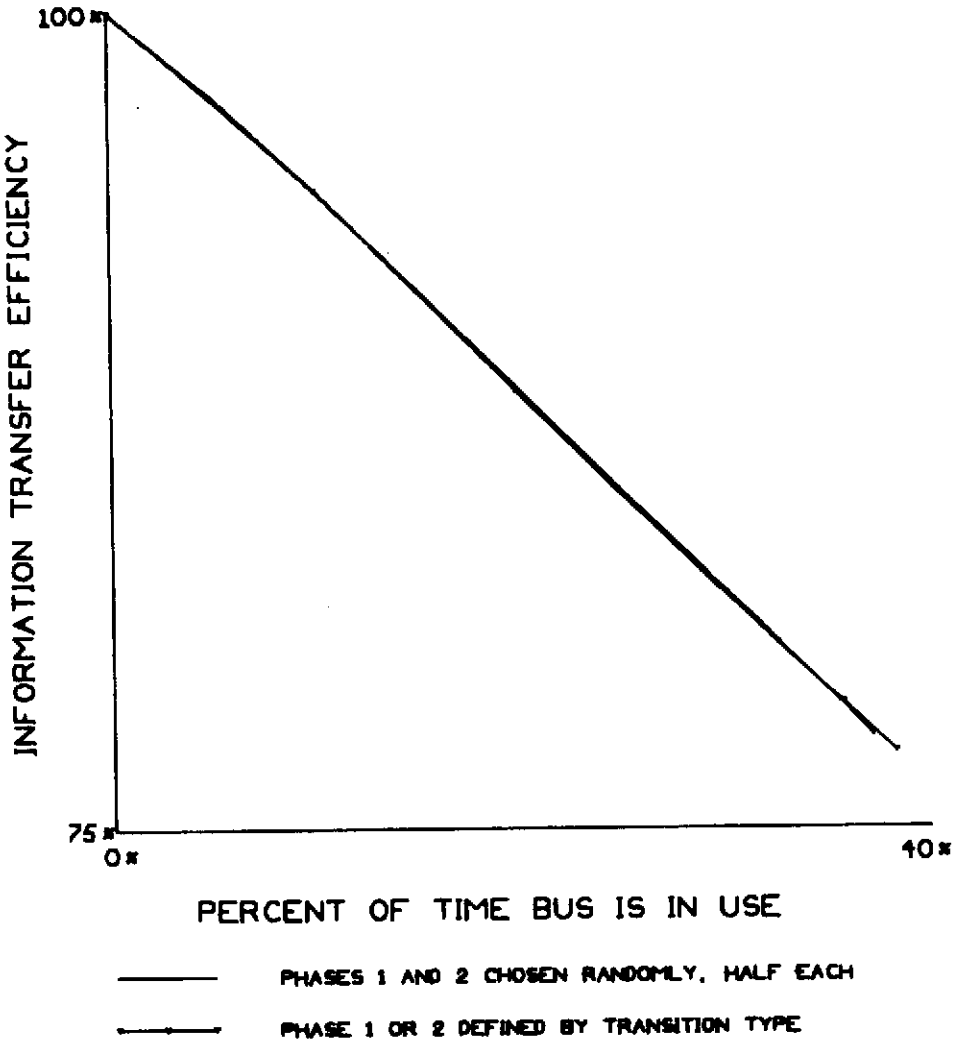


Fig. 5-5 Markov model comparison.

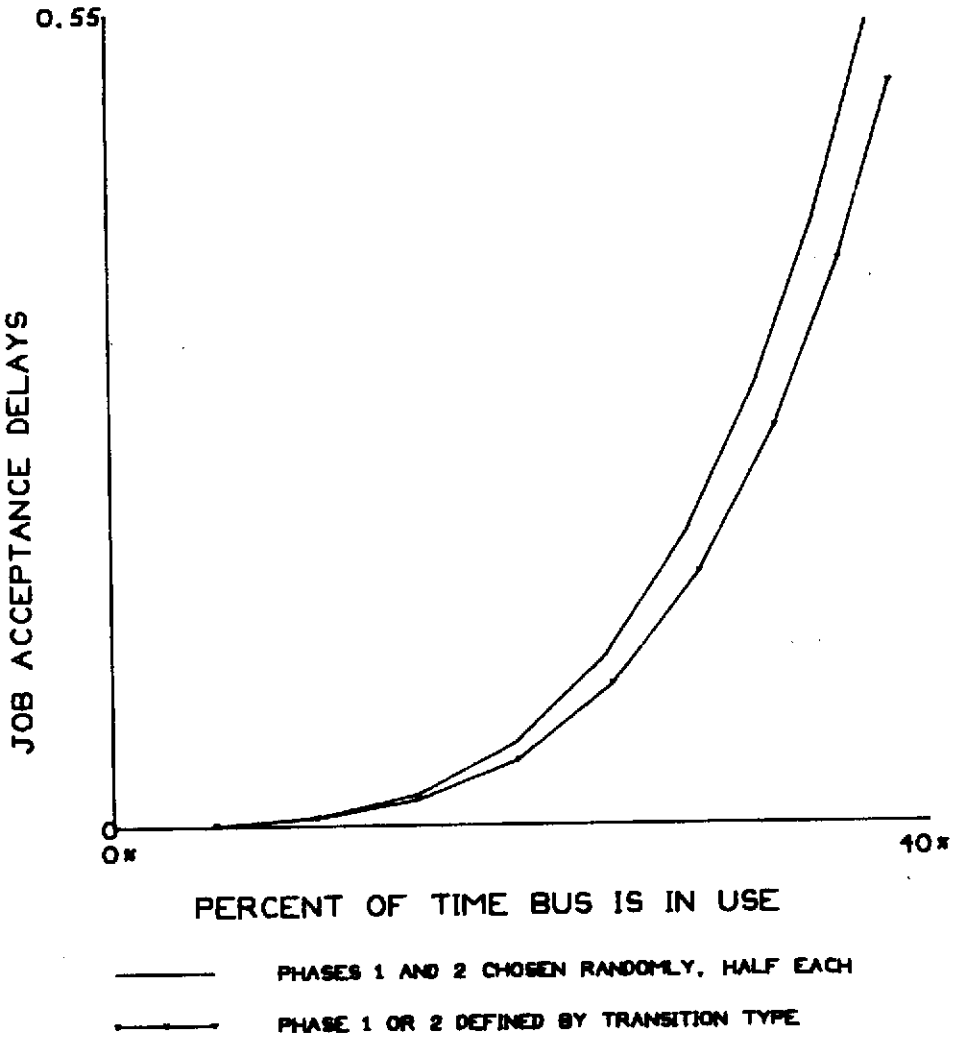


Fig. 5-6 Markov model comparison.

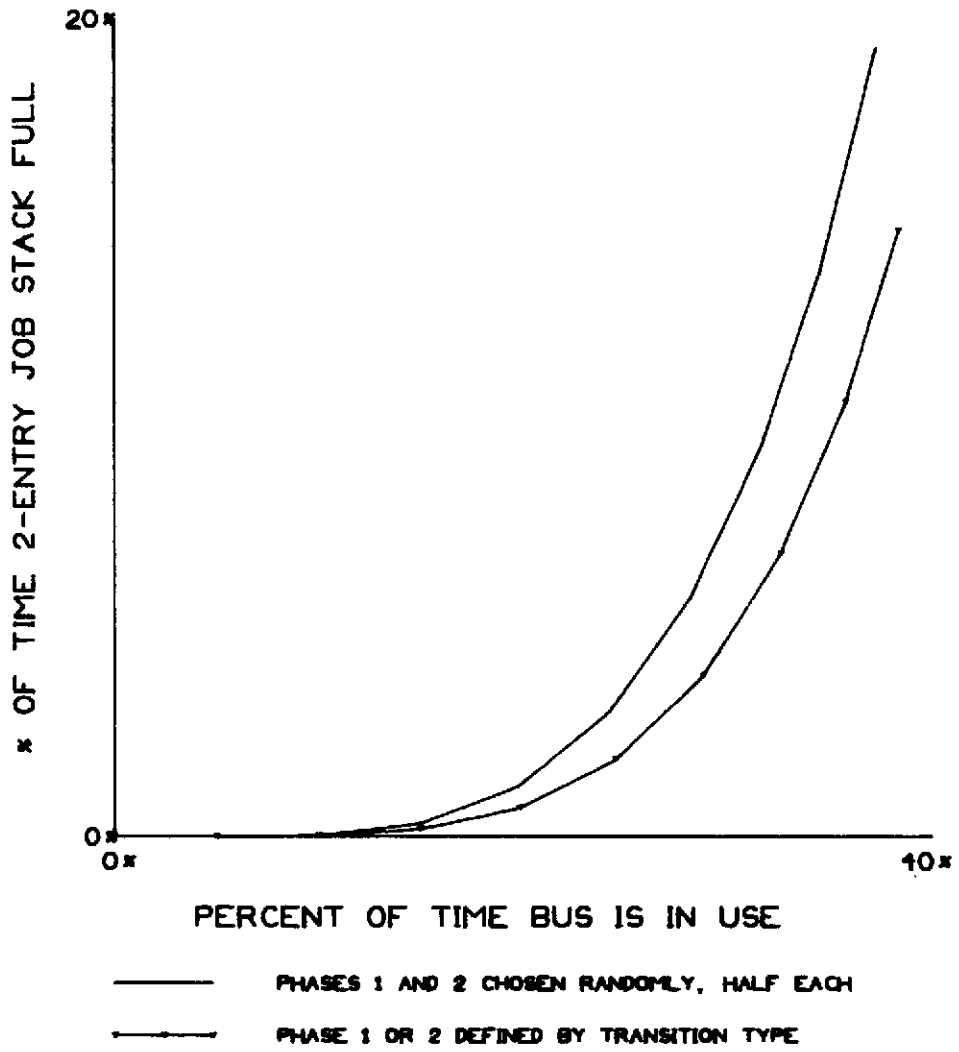


Fig. 5-7 Markov model comparison.

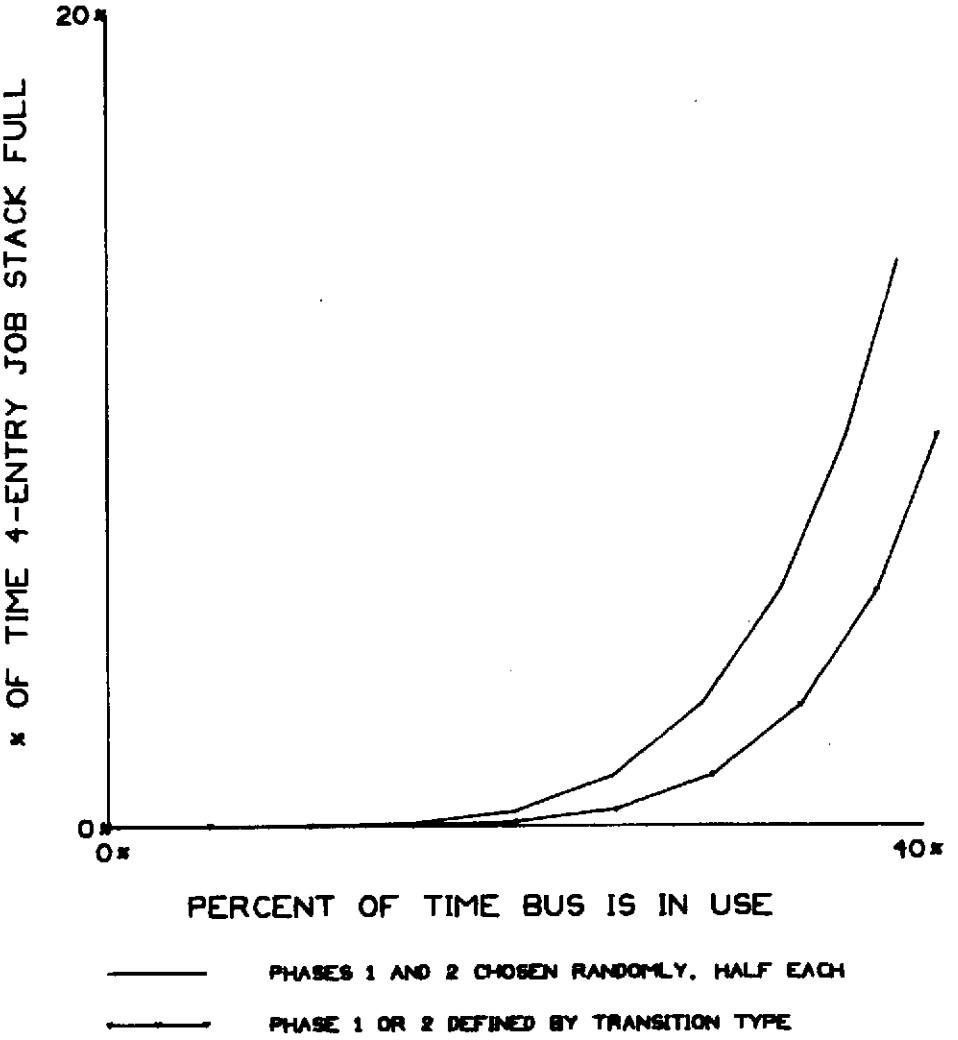


Fig. 5-8 Markov model comparison.

the preceding graphs. The difference is similar to that seen in Fig. 5-7, but is more marked.

Still another comparison between the two models is given by Table 5-1. It shows, for models with a four-entry job queue, the fraction of time that various numbers of jobs are past due. The phenomenon of having four past-due jobs a greater fraction of the time than one, two or three past-due jobs is extremely unrealistic. It conflicts both with one's intuitive notions about such problems and with the mathematical expressions for past-due job number developed in Section 4.2. It is a direct result of the inaccuracies in the assumption that bus releases are equally distributed between terminal and non-terminal releases in all states of the system.

Table 5-1

Distribution of numbers of jobs past due.

JOBS PAST DUE	PERCENT OF TIME	
	BASIC MODEL (Bus use = 38.7%)	TWO-PHASE MODEL (bus use = 37.7%)
0	54.43	61.19
1	12.23	14.38
2	9.99	10.45
3	9.46	8.22
4	13.89	5.76

This very unrealistic effect of the simplest model on the distribution of the number of jobs past due is perhaps the most compelling reason to discard it as an adequate representation of the system.

The two-phase model will be used in future analyses where possible. It is felt that the improvement in accurate modeling of the multiprocessor outweighs, for most purposes, the disadvantages associated with the increased number of system states.

It is, parenthetically, of some interest to note the relationship among the last four numbers in the right column of Table 5-1. The simple queuing-theory model of the multiprocessor predicts that the ratio between adjacent numbers should be given by the system load, in this case 75.4%. Although the bus is sufficiently busy in this example to invalidate the simple model and cause its predictions to be quite poor in the absolute sense, its prediction of this ratio is quite close to the ratio as predicted by the more complex models.

5.2.3 Job duration modeling.

The actual distribution of job times in the lunar landing job set is highly non-exponential, as mentioned in Chapter 4: very short jobs and very long jobs are more prevalent than the exponential distribution would predict, while jobs of intermediate duration are comparatively rare.

To test the importance of modeling this feature accurately, the sixth model of Section 4.5 was developed. This model uses a two-term hyper-exponential distribution for job durations, and is otherwise identical to the 53-state basic model. For simplicity, no other changes were made to the basic model concurrently; it was felt that this simplifies both the creation of the model and the examination of its results.

A series of computer runs under identical conditions, varying only the non-exponentialness of the job duration distribution, was made. These runs used a five-processor system, priority scheme 1, with a limit of two past-due jobs in the queue. Jobs were assumed to arrive at intervals averaging five milliseconds, use the bus for an average time of 1.406 milliseconds in each phase, and compute for an average time of 10.756 milliseconds. These figures result in an overall system load (with no interference) of 54.276% and a bus load of 56.26%. They were made for values of σ of 0.5 (exponential case), 0.1 and 0.01 (which corresponds closely to the best fit to the lunar landing data). Their results, together with simulation results for similar conditions, are shown below:

Table 5-2 Effect of job duration distribution.

	σ = 0.5	σ = 0.1	σ = 0.01	Simulation
Information transfer efficiency	65.78%	54.54%	50.34%	62.33%
Mean Starting Delay	.2765	.3003	.3138	.4244
Percent of time 5 processors busy	17.57%	20.01%	21.21%	15.27%

Three facts are worthy of note in this table. They are:

1. all performance parameters degrade as job durations become less exponentially distributied;

2. simulated job starting delay is considerably higher than any of the delays predicted by the mathematical models; and

3. other simulation results correspond most closely to the predictions of

the model using exponential job distributions, even though this distribution is the poorest match to the job set of the simulation.

Let us consider these observations in order. First, multiprocessor performance degrades as the job duration distribution becomes less exponential. This corresponds to results that can be obtained analytically for simpler problems (see Morse[19], Chapter 7) and appears reasonable. The mechanism is simple: when one of the very long jobs characteristic of the more highly non-exponential distributions is accepted by a processor, it retains control of that processor for a very long time. The system is in effect executing virtually the same number of jobs as before on a system with one less processor.

Second, the simulated mean job starting delay is considerably higher than that predicted by any of the models. The "simulation" curve of Fig.5-9 (the other curves are within the province of the next section) hints at the reason for this. When the system load reaches a certain level, the simulated job set "goes critical" and job acceptance delays become much worse over a very short range of loads. The mechanism causing this effect appears to be the set of closely spaced demands for execution tied to each cycle of the navigation loop. If the multiprocessor is slowed down beyond a certain point, the earlier jobs in this set cannot complete their execution before it is time to execute the later ones. Beyond this point, therefore, job starting delays rise much faster than any mathematical analysis, based on random job arrivals, could predict. The mean starting delay is therefore not, in this case, a valid criterion for use in selecting a model.

Finally, the other two system performance parameters (information transfer efficiency and the fraction of time that all five processors are busy) are best predicted by the model using exponential job duration distributions, even though it is the poorest match to the actual data. The reason for this is similar to the reason for the similar effect noted in consideration of bus-use distributions. The long jobs in the actual situation are largely the navigation loop executions, and since they are phased relative to each other they cannot interfere with each other. In the model, on the other hand, long jobs arrive at random and can interface with each other, be executed simultaneously, and in general degrade system performance to a greater degree.

There does not therefore seem to be a compelling reason to select the hyper-exponential job duration model over the simpler model. In view of the vastly greater complexity of the hyper-exponential model (243 states vs 53), the simpler one will be chosen.

5.2.4 Maximum size of the job queue.

The final parameter to be selected for the "minimum adequate" Markov model is the number of past-due jobs that the model will handle. It will be recalled from Chapter 4 that the number of states in the model increases rapidly with the maximum length of the job queue. For the type of model defined by the discussion thus far a model with a limit of two waiting jobs has 98 states, and each increment of one allowable waiting job adds 31 states.

The importance of the maximum queue length is simple. If a job arrives for execution, and cannot be accepted immediately by a processor, it is placed in this queue. If the queue is full, the job is skipped over completely; its existence cannot be recorded by the model. The fraction of time that the job queue is full is therefore a measure of the fraction of jobs that are skipped over, and thus a measure of one source of model inaccuracy. If arrivals are exponential, as they are in this model, the fraction of time that the job queue is full is exactly the fraction of jobs that are skipped over.

The elementary queuing-theory model of the multiprocessor that was the subject of Section 4.2 provides some assistance in selecting the maximum queue size to be allowed. Equations (4.10) and (4.11) of that section can be rewritten to yield:

$$P_w = \left(\frac{\rho}{Q}\right)^w P_Q \qquad (5.2)$$

where

P_w = probability of there being w past-due jobs waiting to start;

P_Q = probability of all Q processors being busy, but with no jobs waiting to start;

(ρ/Q) = total system load;

w = number of jobs waiting to start.

Although the existence of a bus having finite speed means that these predictions should not hold with absolute accuracy, they are a good indication of the trends to be expected. In particular, Eq (5.2) indicates that the fraction of time that the job queue is full will fall off approximately as a power of the system load.

For heavily loaded systems, therefore, the addition of one more job queue position will have only a small effect on the percentage of jobs skipped over. For lightly loaded systems, a fairly short queue should suffice to reduce the fraction

of skipped jobs to an acceptably small level. The problem is that of selecting the compromise that preserves acceptable accuracy in the middle range without adding an inordinate number of states to obtain marginal gains for heavily loaded systems.

A computer program able to prepare the transition matrices for this model was written, so as to lessen the clerical work that would otherwise be required. This program stepped through all states of the model, and for each state and each possible transition mechanism determined whether the transition mechanism were relevant to that state, and, if it were, to what state the transition would be. The program accepted as input a specification of the job queue length desired for the model, and produced the transition matrix in a form usable by the Markov process analysis program referred to in Chapter 4.

Computer runs using this model with varying job queue sizes were made under the conditions used for selecting bus use models: five processors, priority scheme I, and a 9:1 ratio of average computing time per job to average bus use time per job. Results of these runs are plotted in Fig. 5-9 through 5-11.

Figure 5-10 shows that the size of the allowable job queue has negligible effect on predictions of information transfer efficiency.

Figures 5-9 and 5-11 show the effect of allowable job queue size on predictions of job acceptance delays and on the number of jobs passed over because they would not fit in the queue. Delays increase as the queue size increases; this is expected, since the jobs occupying the higher queue positions, and hence being delayed the longest before execution, are passed over when the allowable queue is short. The number of jobs passed over decreases with increases in the allowable queue size; this too is an expected result, since the longer queue can accommodate more jobs.

It appears from Fig. 5-9 and 5-11 that there is an effect of "diminishing returns", whereby the addition of one more job queue position has less and less effect on the results. (Quite possibly, constant percentage increases in job queue length have constant effects on the results.) No such effect exists with respect to the complexity of the model, however; each additional job queue position adds 31 states to the model and adds an approximately constant increment of computer time to the analysis.

The compromise chosen was a maximum queue size of five jobs. This size provides accuracy of better than 5% to a system load of approximately 80% with reasonable computer time needs. Longer job queues, curves for which are not shown, have small effect on the accuracy; computer time was not so restricted as

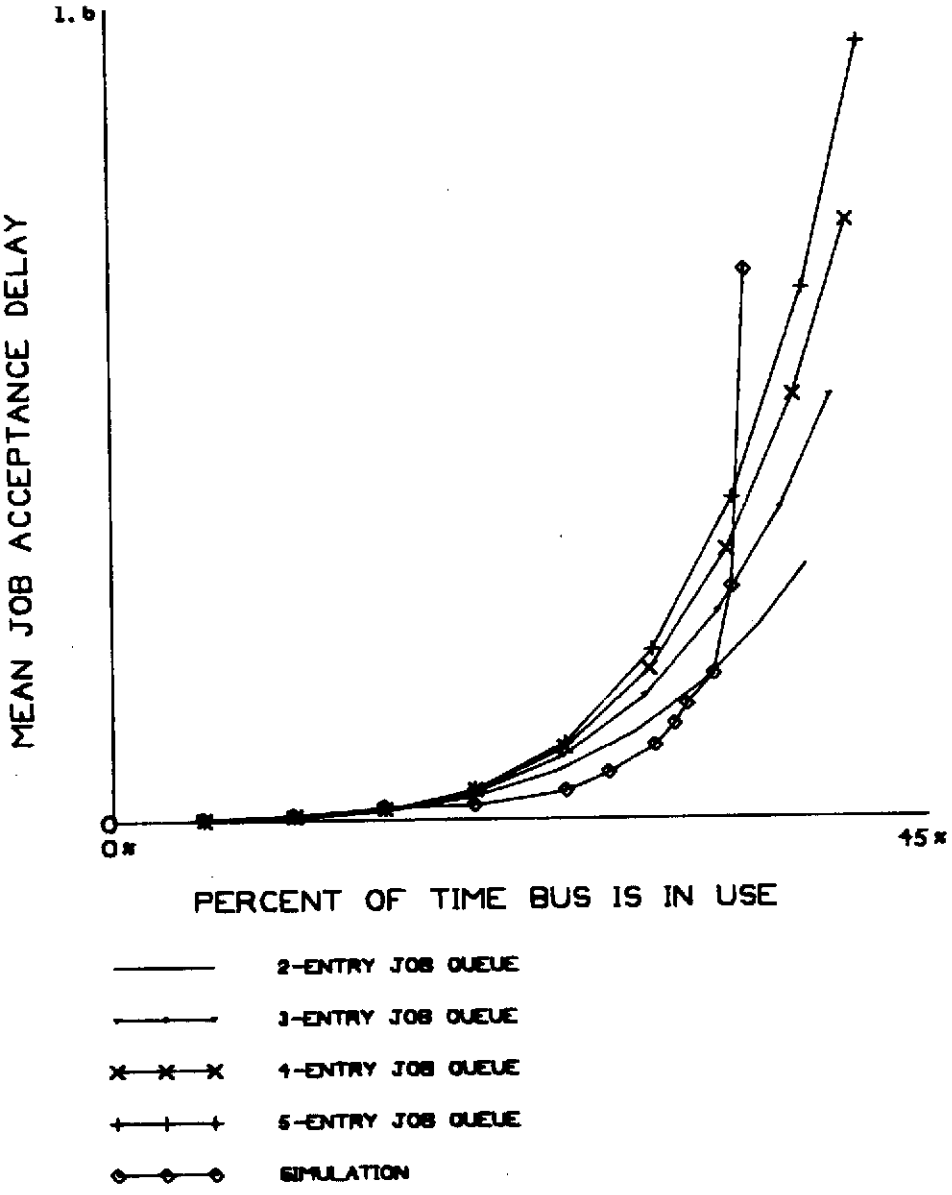


Fig. 5-9 Effect of maximum job queue size.

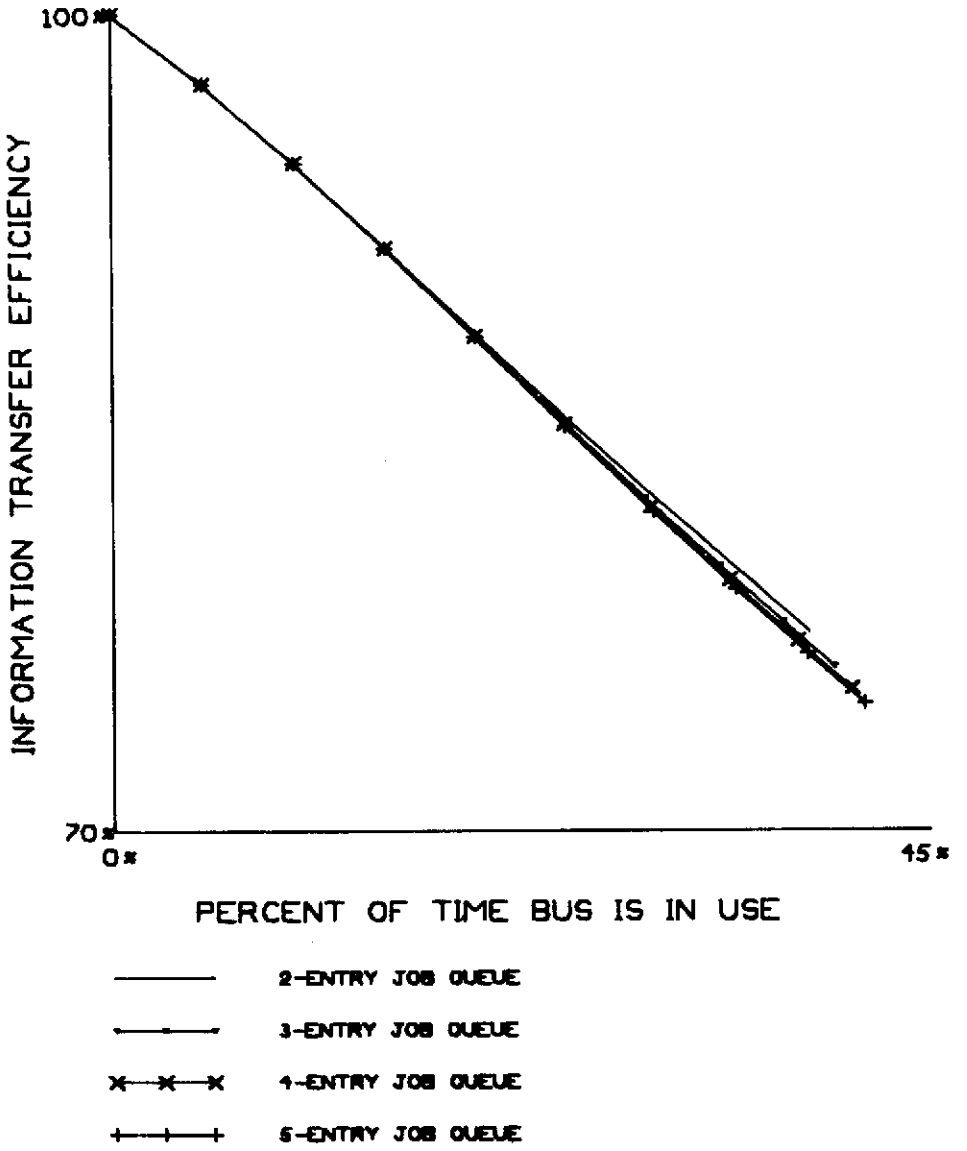


Fig. 5-10 Effect of maximum job queue size.

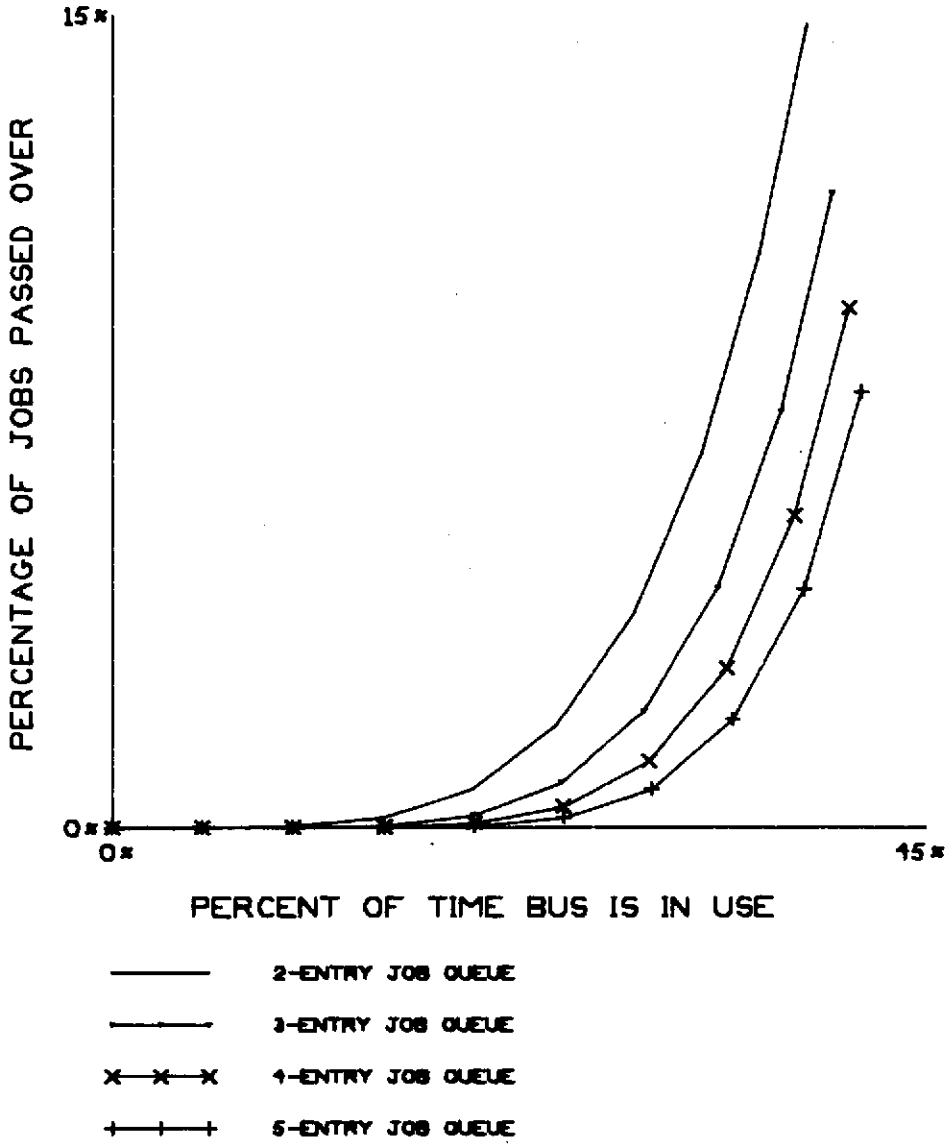


Fig. 5-11 Effect of maximum job queue size.

to require use of a shorter queue. It would be reasonable to presume that, under different circumstances, a different selection might be made.

The "minimum adequate" Markov model of the multi-processor thus has the following characteristics: it assumes exponential distributions for job arrivals, job durations and job bus usages; it retains information as to the phase of bus use; and it can accommodate a queue of up to five past-due jobs. For a five-processor system, this model has 191 states.

5.3 Effect of Job Arrival Distribution

It would appear reasonable that, as the distribution of job arrivals becomes more regular, the efficiency of the system would increase. There would no longer be, to the same extent, periods of frenzied activity followed by periods of almost no activity at all. The activity rate at any moment would be closer to the average activity rate, and the delays, etc., characteristic of peak activity periods would be reduced.

To test this assumption, a simple Markov model of the multiprocessor was developed which assumed a 2-Erlang job arrival distribution. This model was discussed in Section 4.5. Although it is a modification of the basic 53-state model, rather than of the more accurate 191-state two-phase model, comparisons between it and the corresponding model with exponential arrivals should still lead to valid conclusions.

Figure 5-12 shows the effect of job arrival distribution on information transfer efficiency. The difference is small but noticeable. Presumably, with more regular arrivals, the efficiency would be still higher. Another reason the difference is not larger is that the job durations are still exponential in the 2-Erlang arrival model, so that demands for bus use (which depend both on job arrivals and on job terminations) are intermediate in regularity.

Figure 5-13 shows the effect of job arrival distribution on job starting delays. Again, the distribution having more regular job arrivals shows slightly better performance than the other.

It appears reasonable to conclude that multiprocessor computers are more efficient when jobs are scheduled for execution at more regular intervals. The development of a means of performing such scheduling automatically, probably at program preparation time, would seem to be a fruitful area for further work.

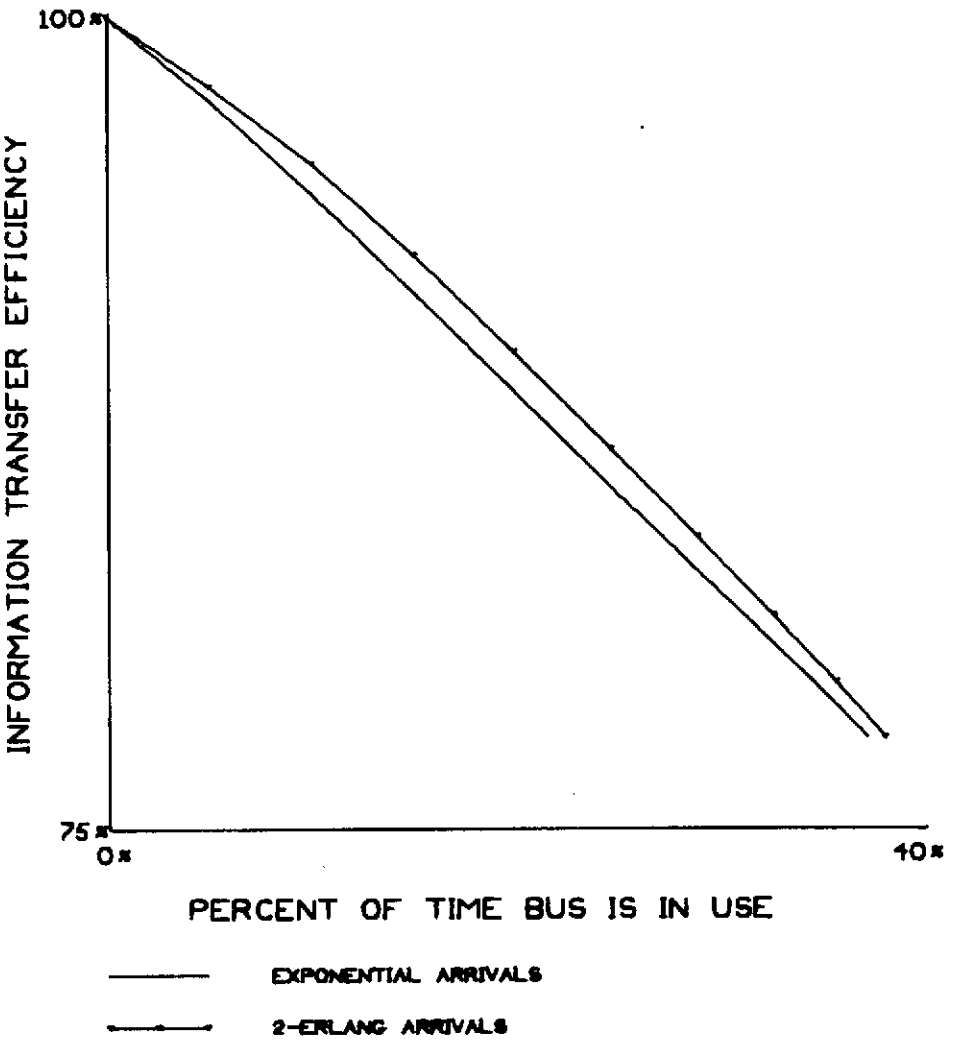


Fig. 5-12 Effect of job arrival distribution.

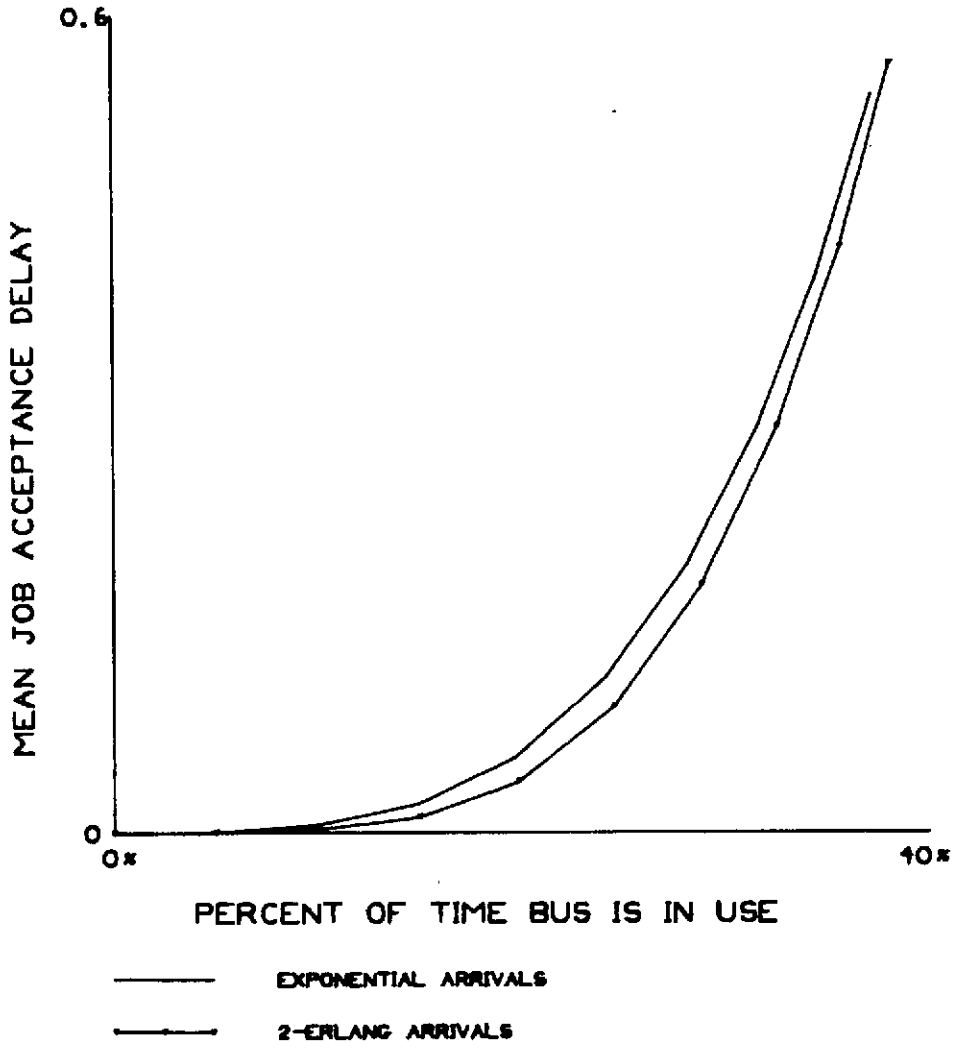


Fig. 5-13 Effect of job arrival distribution.

5.4 Effects of Varying the Number of Processors

A number of design tradeoffs exist in the design of a multiprocessor computer. One of these is the tradeoff between the number of processors in the system and the speed of the individual processors. For a given computational capacity, a designer can choose to use a small number of fast processors or a larger number of slower ones.

Reliability considerations might, under different sets of circumstances, argue in favor of either approach. For a given state of the art, a slower processor would use either fewer components or more conservatively operated components than a faster processor would. In either case, the probability of failure of the slower computer would be smaller. In addition, the failure of one out of many slow processors would reduce system capacity less than would the failure of one of a few fast ones. Offsetting these considerations is the fact that there would be more slow processors able to fail.

Consider, for example, a system with eight processors, each of which is 90% reliable over a given mission. Assume that each pair of processors can be replaced by one processor which is twice as fast as the original ones and exactly as reliable as the original pair — or 81% reliable for the same mission. If successful mission completion requires three-fourths of the system computing power, two of the slower processors are permitted to fail, but only one of the faster ones. Under these assumptions, the system having eight slow processors has a 96.2% probability of successful mission completion, while the system having four fast ones has a 94.8% probability of success. Naturally, the above reliability figures are quite arbitrary, and the particular design considerations that influence processor speed under a given set of circumstances would determine the reliability loss associated with a speed increase. Under some circumstances, a system having fewer, faster processors could be more reliable than a system having more, slower ones.

When the philosophy of "more, slower processors" is to be followed, there is a limit to the extent to which it can be applied in practice. This limit arises because of the nature of aerospace programming jobs: there are, in any situation, a number of tasks that must be executed periodically, at predetermined intervals. It is ordinarily necessary that one execution of such a task have completed its work and returned its results before the next can begin. Thus, the individual processors must be sufficiently fast that each such periodic task can be completed within its allotted time on one processor.

An additional consideration might be the possibility of memory conflict where various jobs modify the same portion of central memory. Such jobs must, as mentioned earlier, be prevented from running concurrently. The importance of this phenomenon, if applied to a particular set of jobs, is minimized when the number of processors in the system is minimized.

Finally, considerations of weight, number of external connections, power consumption, etc., might argue in favor of a small number of faster processors.

With these considerations, which will not be further discussed in this section, in mind, we may proceed to an examination of the effect of varying the number of processors in a system. Effects of this change were determined using the elementary queuing-theory model of the multiprocessor, as developed in Section 4.2, and the simulation. Markov models were not used because of the need to construct a separate transition matrix for each system.

The total system load was kept constant at approximately 60% of capacity. Computing speeds of the individual processors were varied to maintain this load as the number of processors changed. Because of the periodic nature of the tasks being performed, as discussed earlier in this section, the simulated processors could not be slowed down beyond a speed corresponding to a five-processor system. Systems having more processors and a load of 60% were not, therefore, simulated.

The results of these runs are shown in Fig. 5-14 through 5-17. There are large discrepancies between the results of the elementary queuing-theory model and those of the simulation, but the tendencies are in all instances in the same direction. It is quite clear from these graphs that a system with more, slower processors performs more efficiently than a system with fewer, faster ones.

Figures 5-14 and 5-15 show the fraction of time that no processors are busy (5-14), and that all the processors in the system are busy (5-15). The probability of each of these limiting cases is reduced as the number of processors increases and their speed decreases. Since the system is less often completely busy, one would expect the delays in job starting to be smaller with more processors; Fig. 5-16 bears out this assumption. Finally, Fig. 5-17 shows the fraction of time that the queue of jobs past due for starting is empty, and thus the fraction of time that the system is performing all its assigned tasks in a timely manner; this fraction goes up as the system goes from two fast processors to twelve slow ones. (There is no "simulation" curve on this graph since the information could not be obtained from available simulation output.)

It can be concluded that, for a job set not having memory lockout problems, and subject to the restrictions discussed earlier in this section, a multiprocessor

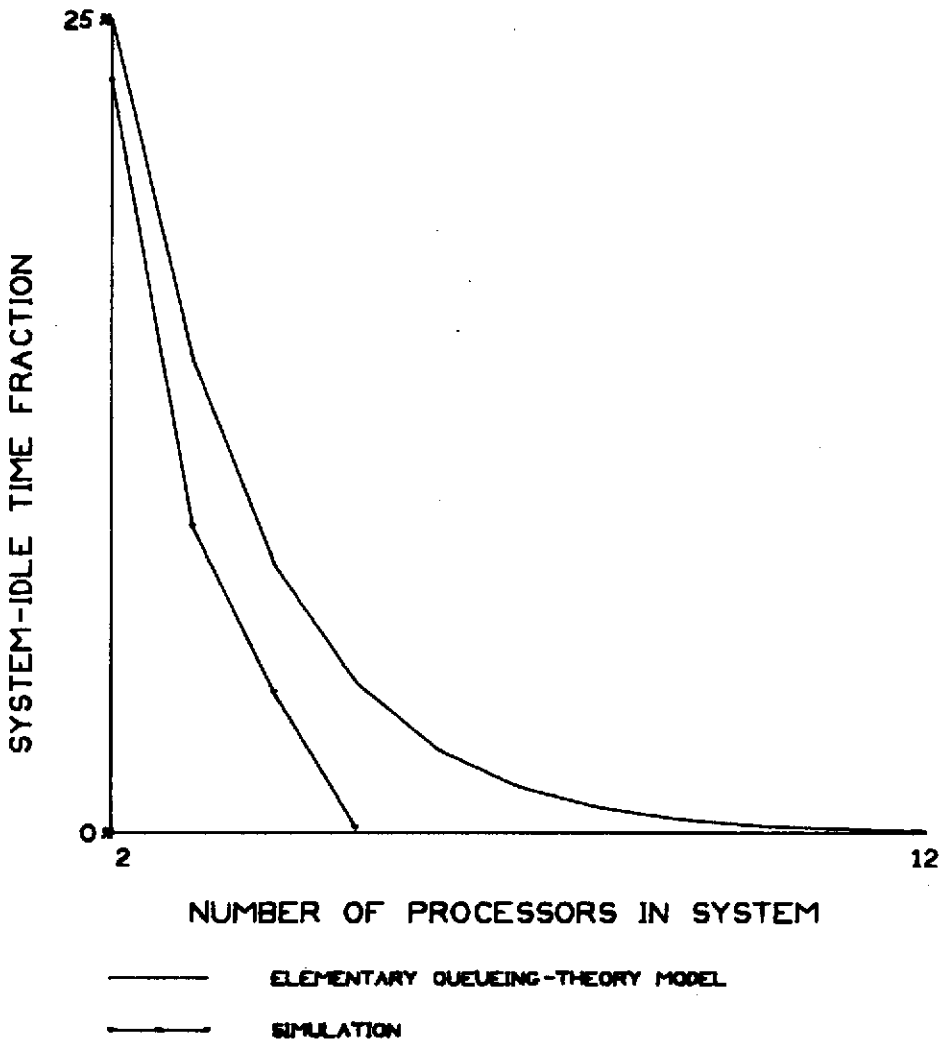


Fig. 5-14 Effect of additional processors.

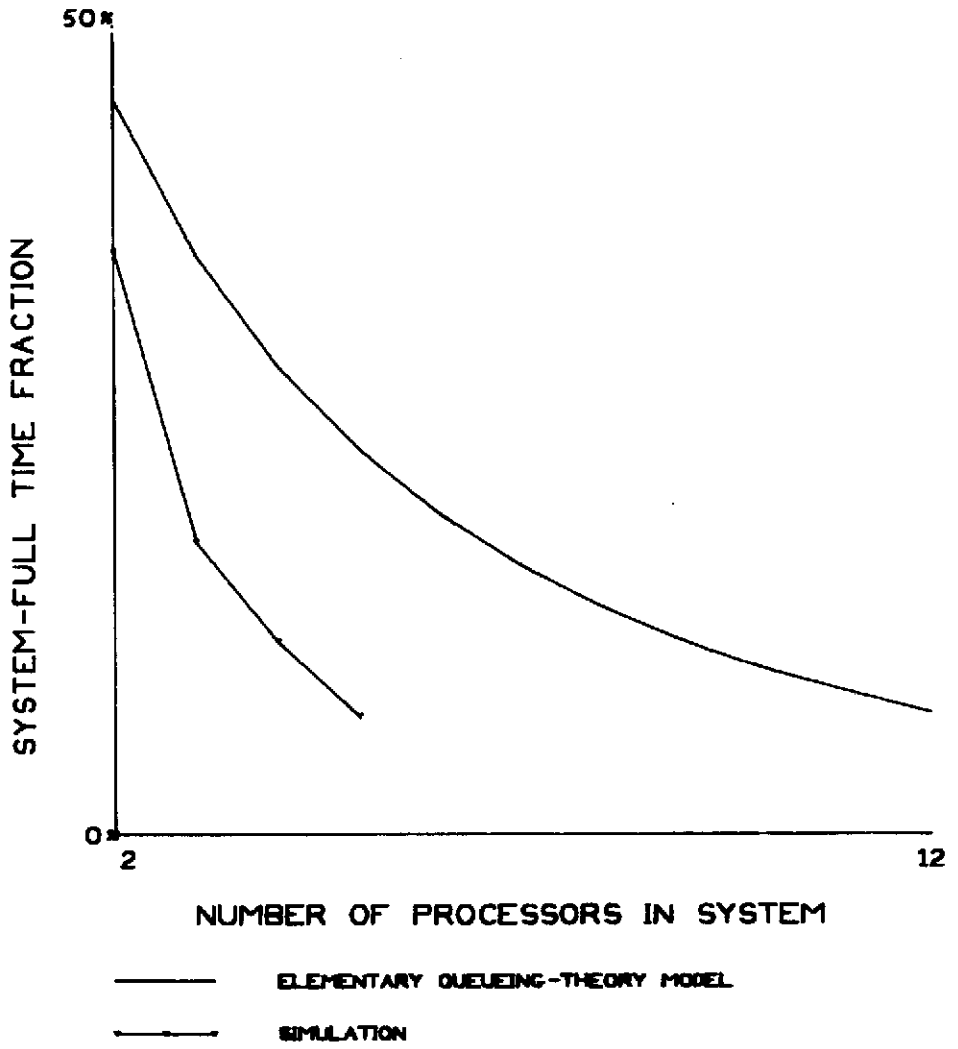


Fig. 5-15 Effect of additional processors.

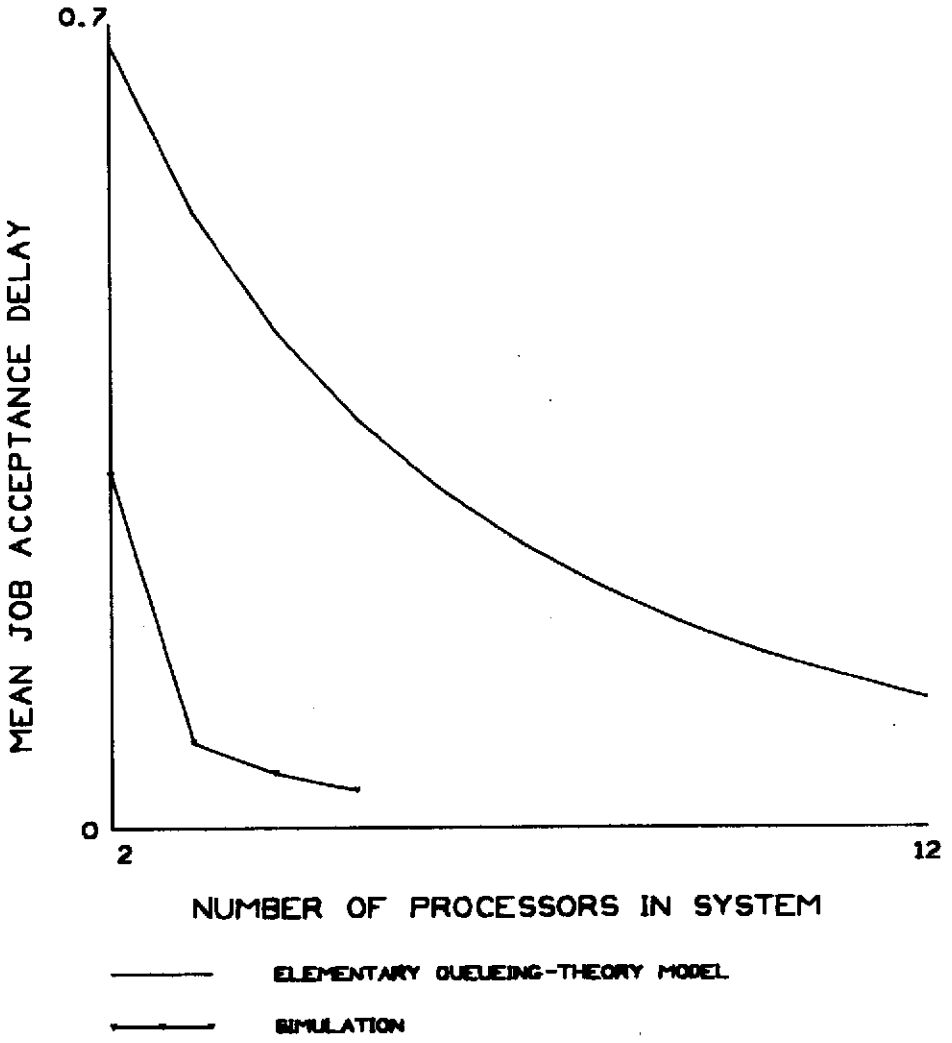


Fig. 5-16 Effect of additional processors.

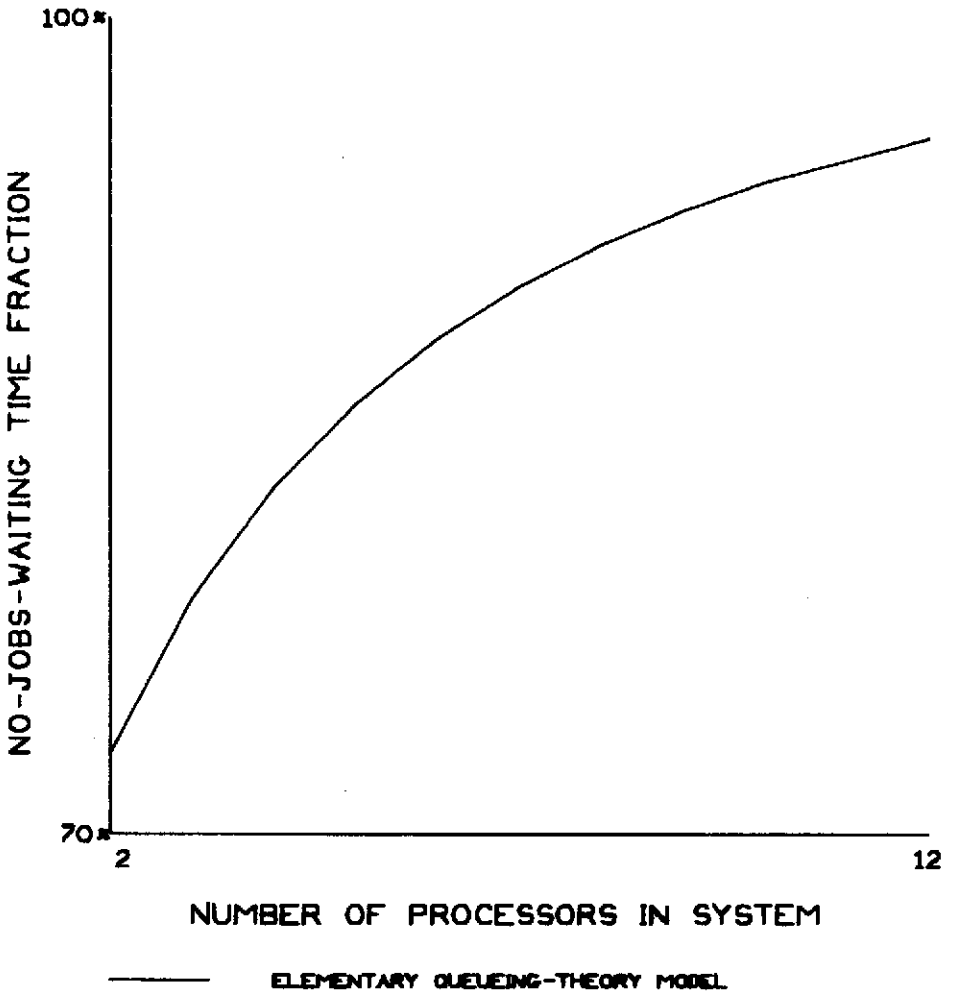


Fig. 5-17 Effect of additional processors.

having many slow processors is, for a given computational capacity, more efficient in every way than one having few fast ones.

5.5 Effect of Changing the Priority Scheme

The scheme used by the multiprocessor hardware to resolve simultaneous demands for use of the bus should clearly have an effect on the performance of the system. Four possible bus access priority schemes were mentioned in Chapter 1. Two of these are compared in this section:

a. Priority scheme 1, where the job stack has priority in bus access whenever it and a processor both want to use the bus. When the job stack sends out a job request message, the processor accepting the new job obtains the use of the bus immediately, irrespective of its position in the processor chain. Conflicts among processors wanting to use the bus at other job execution phases are resolved by having each processor enable its "neighbor" on the bus.

b. Priority scheme 4, where all devices attached to the bus — processors and the job stack — are treated equally. Each device enables its "neighbor" to use the bus. Thus, the job stack will be able to use the bus only once for each circuit of all the processors made by the "enabling pulse". When a processor accepts a new job, it too must wait its turn in the ring to be able to use the bus to obtain the data it needs to begin computations.

These two schemes differ in their orientation. Scheme 1 is oriented toward quick acceptance of new jobs by a processor, whereas scheme 4 sacrifices this objective and is oriented more toward timely completion of those jobs already in progress in the various processors. One would therefore expect that, for given processor speed, bus speed, and number of processors, a priority scheme 1 system would exhibit smaller job starting delays, while a priority scheme 4 system would exhibit higher information transfer (and other) efficiencies.

The validity of these expectations was tested by a series of computer runs using the simulation and the 191-state two-phase Markov model selected earlier in this chapter. A five-processor system was used. The ratio of computing time to bus use time was held to an average of 9:1, varying the overall speeds of the components so as to vary system load. The load can therefore be represented by any single load parameter, such as the percent of time that the bus is in use.

Results of these runs are plotted in Fig. 5-18 through 5-21. Figures 5-18 and 5-19 show, respectively, information transfer efficiency and job starting delays as predicted by the Markov model. Figures 5-20 and 5-21 are similar, but display simulation results.

The expected tendencies can in fact be seen, both in the Markov model predictions and in the simulation results. (Reasons for discrepancies between the numerical results of the two approaches, particularly in the area of job starting delays, have already been discussed.) The differences in information transfer efficiency are appreciable, but it must be remembered that overall efficiency, which includes computation time, is higher than information transfer efficiency. Therefore, the differences in overall efficiency would be small. The difference between the two schemes in job starting delay is small, except for simulation results beyond the "critical point" of the system.

In a real situation, the approach to be taken would depend both on hardware implementation costs and on theoretical considerations. It appears clear that a system implementing priority scheme 4 would be less difficult and less expensive to construct than one implementing priority scheme 1. The difference in performance between the two appears, for most conditions of interest, to be small. The simpler approach of priority scheme 4 would therefore be more desirable under most circumstances.

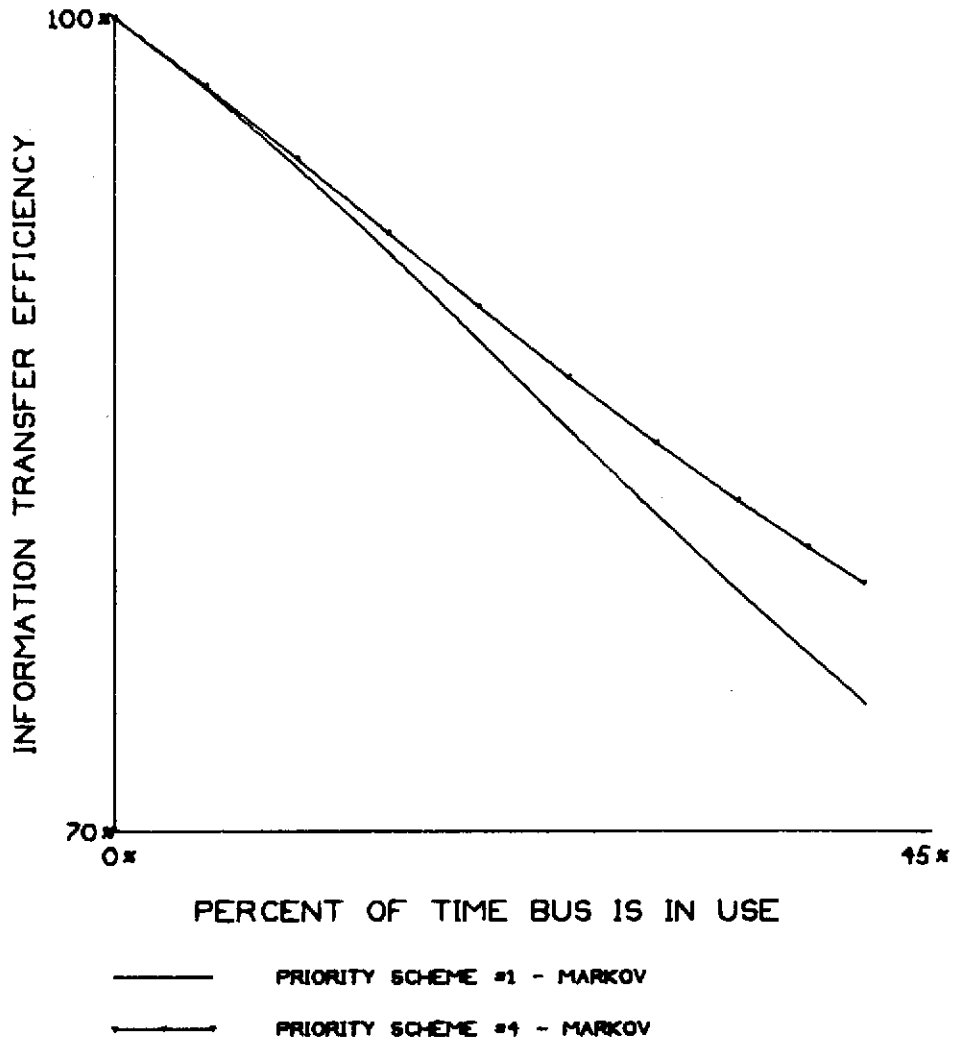


Fig. 5-18 Effect of priority scheme.

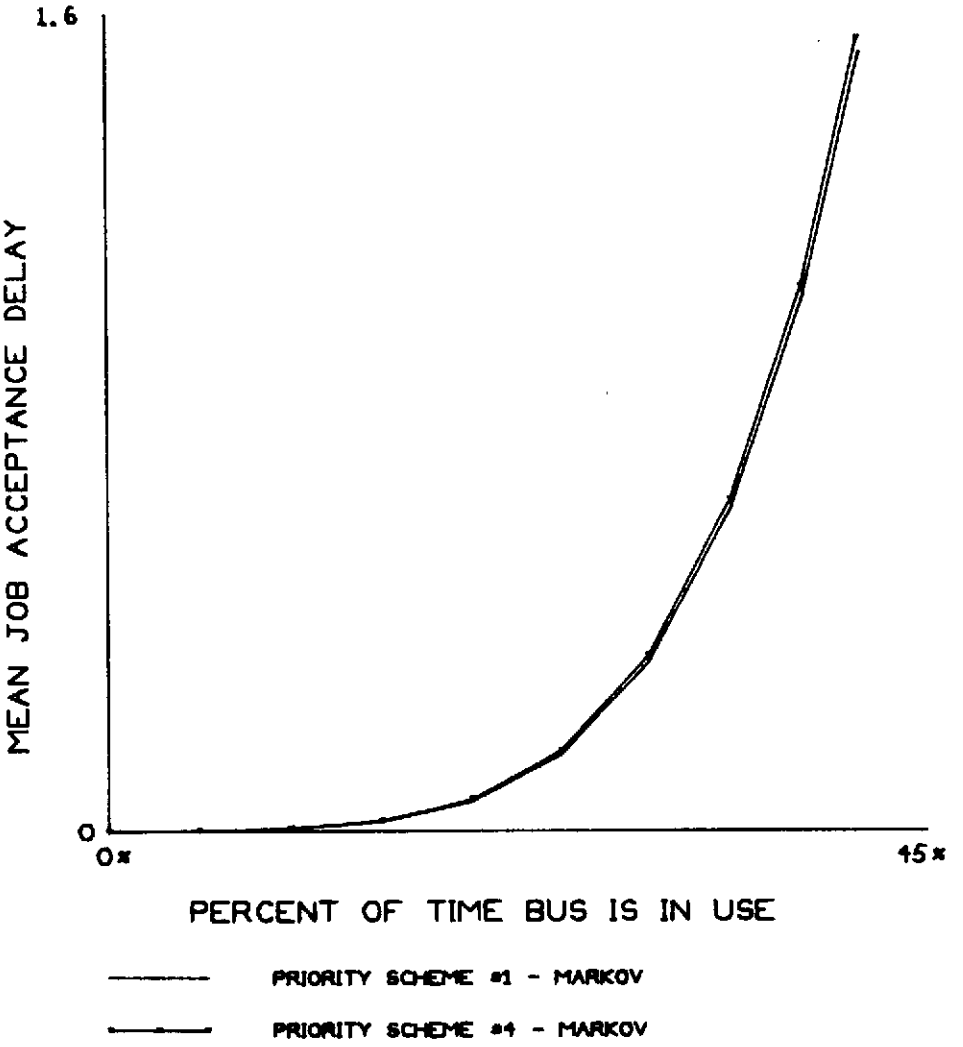


Fig. 5-19 Effect of priority scheme.

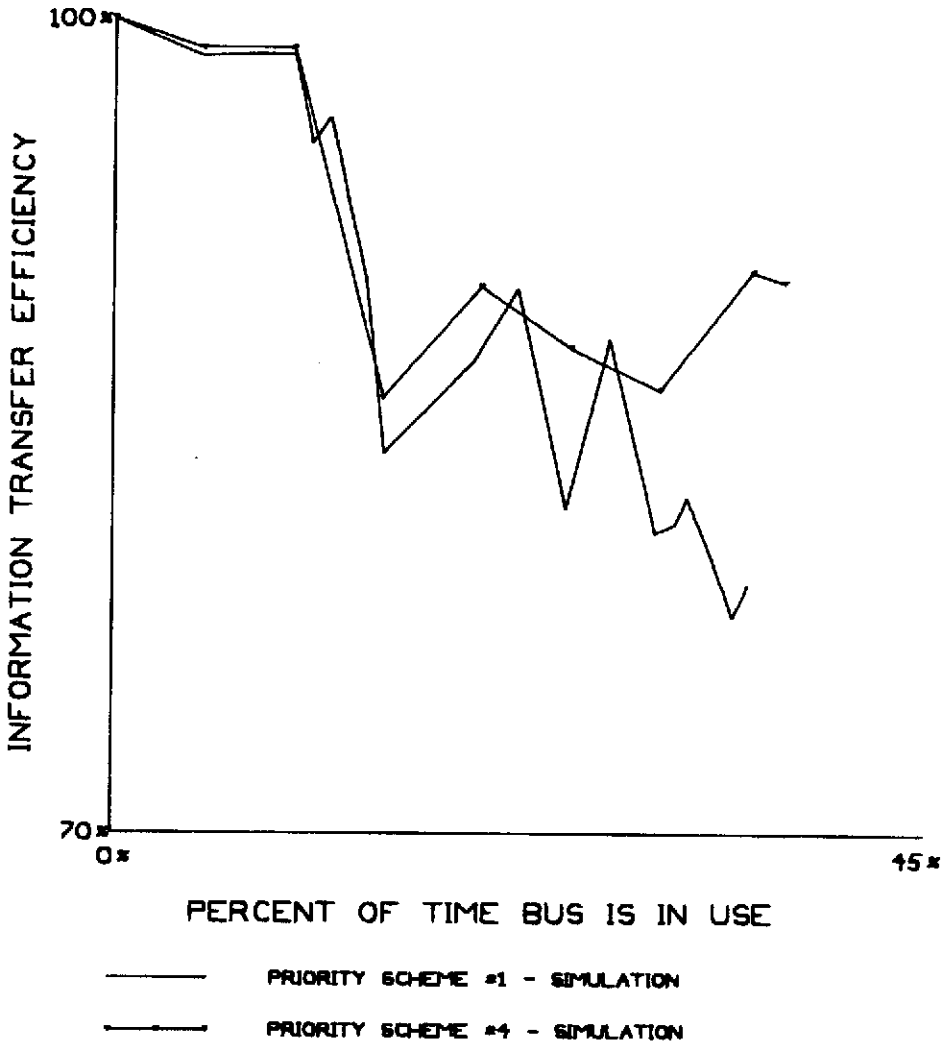


Fig. 5-20 Effect of priority scheme.

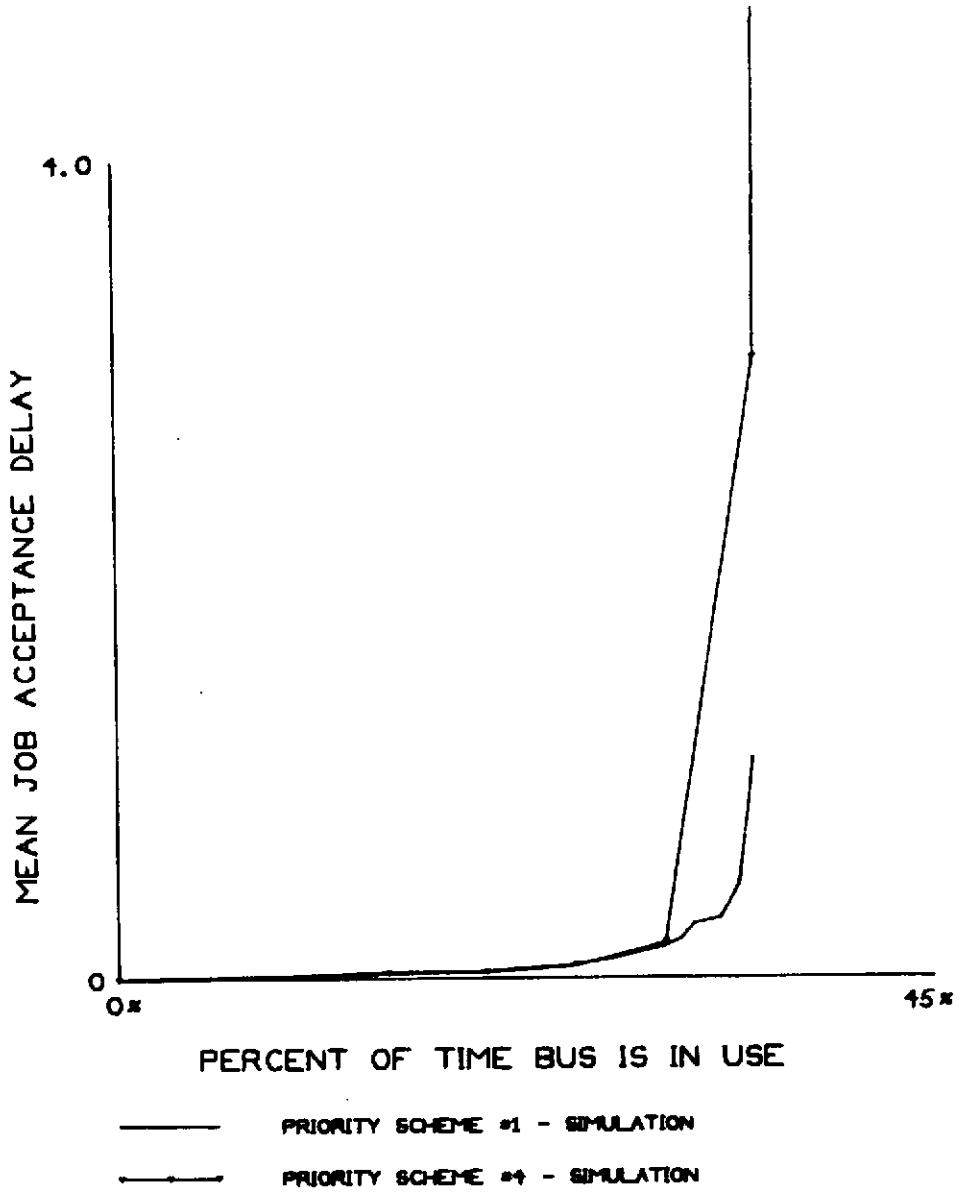


Fig. 5-21 Effect of priority scheme.

CHAPTER 6

CONCLUSIONS

The present chapter summarizes various conclusions with respect to the analysis of the multiprocessor. These have already largely been presented within the previous portions of the text, primarily in Chapter 5. The purpose of the present chapter is to localize them for ease of reference. The various points mentioned are discussed more fully in the appropriate place in the preceding chapters. Suggestions for further research are also made.

6.1 Lunar Landing Job Analysis

The computer programs representing computational tasks in the Lunar Module Guidance Computer range from the very short to the very long. Short jobs predominate numerically, but long jobs take up a majority of the computer's time: in an average two-second period, during which about 135 jobs are executed, over one-half the computing time is occupied with one job (the navigation loop). An additional 22% of the computing time isused by the digital autopilot, executed 20 times in a two-second interval. The other approximately 115 job executions (about 85%) take up about one-fourth of the computer's time.

For efficient use of a multiprocessor, it is necessary to subdivide a very long job such as the navigation loop into a set of shorter jobs. It would be desirable to permit the programer to concentrate on the subject matter of his programs, and not be concerned with the efficiency of the computer on which they will run. It would therefore be desirable to develop a method capable of automation for performing this subdivision. This remains an area for further research.

6.2 Queuing-theory Analysis

It is possible to model the multiprocessor usefully as a Markov process or, somewhat less usefully, as a simple queuing-theory problem. The simple queuing-theory analysis permits predictions of the effects of overall system load, job arrival rate and time distribution, and number of processors. The Markov analysis permits, in addition, predictions of the effects of data bus load and bus access priority scheme.

6.3 Minimum Adequate Markov Model

The job duration and bus use time distributions in the lunar landing data are not well matched by an exponential distribution, the simplest one to use in a Markov

or queuing-theory analysis. A two-term hyper-exponential distribution provides a better match. However, the improvement in matching gained by using such a distribution does not result in a corresponding improvement in the accuracy of the predictions. In fact, for reasons discussed in the body of the text (Sections 5.2.1 and 5.2.3), this improved matching can result in less-accurate predictions. The use of exponential distributions is therefore considered adequate to represent actual aerospace programs.

It was determined that it is essential, for accurate predictions, that the Markov model keep track of the purpose for which the bus is being used at any given time, rather than simply noting the fact that the bus is being used.

In reaching a compromise between the number of states in the Markov model (which should be small) and the number of jobs past-due for starting that the model will accomodate (which should be large), a "job queue" limit of five jobs was felt to be reasonable.

Such a Markov model of a multi-processor, for a system with P processors and permitting a maximum of Q past-due jobs, has the following number of states:

$$\text{Number of states} = \left[P(P+1)+1\right](Q+1)+P \qquad (6.1)$$

6.4 Effects of Job Arrival Scheduling

The scheduling of job arrivals has an important effect on the efficiency of the multiprocessor. This efficiency improves as the arrival of jobs becomes more regular in time, as shown by Fig. 5-12 and 5-13, and as discussed in Section 5.3.

It is possible to schedule jobs so that the multiprocessor performs more efficiently than it would with random scheduling. (It is probably possible to schedule jobs so that it performs better than it would with perfectly regular scheduling. The question of what the "optimum" in this area might be is open for further research.) This scheduling cannot be done by the programer as he writes one program, and could probably be done only with difficulty by a person analyzing the overall system load. The development of a means for doing this scheduling automatically (before the programs are run, since the time spent doing such scheduling as they are running would probably exceed the time gained through the added performance) is an open area.

6.5 Effect of Number of Processors

As the number of processors increases, for a given total system computing capacity, the system efficiency increases. In other words, a system with many slow processors is more efficient than a system with a few fast ones.

This conclusion does not necessarily apply to a system in which conflicts for use of a specific memory location (to modify a certain item of data) are significant. In fact, the entire question of the effect of memory conflict was not investigated in the present study. Memory conflict would be an important question when one pro-

gramer-written job is broken down into smaller ones, or when a number of programers are working on independent programs that modify the same data base.

There is a limit to the degree to which one can replace a fast processor with a number of slow ones: it must always be possible for each iterative task in the system to complete its work and return its results within its permitted iteration time. This provides a lower bound on the speed of the individual processors on which these tasks must be executed. It is largely this consideration that argues for the breaking down of large jobs into smaller ones; they could then be executed on slower processors.

6.6 Effect of Priority Scheme

Two priority schemes, through which it is determined which of many possible users obtains access to the data bus when more than one user requests it, were studied. They were the following (the numbers "1" and "4" refer to a set of schemes described in Section 1.2):

> Priority Scheme 1: The job stack obtains the use of the bus whenever a new job wants to start. The new job, if it is accepted by a processor, can use the bus immediately to obtain data.
>
> Priority Scheme 4: The job stack waits its turn in a "ring" of bus users when a job wants to start. The processor accepting a new job likewise waits its turn.

It was found that a system with Scheme 1 has shorter delays in getting jobs started, but a system with Scheme 4 completes jobs more efficiently. For the cases studied, the differences were quite small. It was concluded that the priority scheme may safely be designed on the basis of considerations of hardware simplicity and reliability.

6.7 Validity of Queuing Theory vs Simulation

In every instance, the Markov models of the multiprocessor predicted the same trends that were predicted by the simulation. It can therefore safely be concluded that the Markov analysis is valid for examining various system architectures.

This analysis is, however, limited to the prediction of overall trends. Many details of system behavior, and the reaction of a system to a particular set of jobs, cannot be obtained by this method. When detailed performance information for a given situation is needed, simulation is the only workable method of analysis.

6.8 Additional Areas for Further Work

Many aspects of the design of the multiprocessor were intentionally not investigated here. The question of program storage, for example, remains open. Hardware-software tradeoffs in the implementation of the job stack could be studied.

The list is long, and includes virtually all details of the system architecture.

Other areas of investigation deal with the nature of the computer job mix. Questions of job scheduling and job breakdown have already been discussed. Another potentially intriguing question is the following: should a job having very large data transfer requirements voluntarily relinquish the bus before completing the transfer? Such action would delay its own completion, but would speed the acceptance and completion of other jobs. The potential benefits and drawbacks of this approach should be weighed.

Further study is also needed in the area of error detection. A fundamental tenet of the multiprocessor approach is that hardware errors can be detected before erroneous data is emplaced in the system. Within this area, there are opportunities for studies of hardware design, hardware-software tradeoffs (error detection by comparing two results, supposedly identical, arrived at in different ways) and automatic (compile-time) generation of error-detection code. The question of signaling a failure, once it is detected, to the system (so that the job can be restarted) is also pertinent. Perhaps jobs should be restarted automatically if they do not issue successful termination messages within a certain interval. What interval? Measured from when? Fixed or variable? If variable, how specified? What advantages and disadvantages do this and other approaches to job restarting have?

How does the job stack know when a job is not accepted by a processor? What should it do about such situations? This entire area is open to the specification of possible schemes and the analysis of their advantages. It is not simply a question of designing something that will work, since this can clearly be done; it is a question of designing the approach that will work best.

There are many areas of research still open with respect to this multiprocessor structure, to say nothing of the many possible alternative multiprocessor structures. To most of the questions, there are no pat "right" or "wrong" answers. There are, rather, tradeoffs to be investigated and methods to be developed. The analyses and methods will be of direct usefulness in the design of the next generation of spaceborne computers.

APPENDIX A

DETAILED DESCRIPTIONS OF THE LUNAR LANDING JOBS

This appendix lists the 43 jobs extracted from the lunar landing programs, with brief functional descriptions. Starred jobs (such as JOB01 below) are those which are active during the visibility phase steady-state. Table A-1 lists the numerical characteristics of each job.

*JOB01 controls the relays that operate the crew displays. This job cycles at a rate of 0.12 seconds, testing for program commands to change the display. When such a command is found, it changes the display, two digits at a time, at a rate of 0.02 seconds per cycle until the changes are completed. Assumed statistics were: if a cycle is a "non-display" cycle, the next cycle has an 0.15 probability of being a "display" cycle; if a cycle is a "display" cycle, the next cycle has an 0.50 probability of being a "display" cycle. These assumptions result in 77% "non-display" cycles and 23% "display" cycles, corresponding to an average of 2.37 "display" cycles per second or 4.74 changed display characters per second. This is felt to be sufficiently realistic for the present purposes.

*JOB 02 monitors the PROCEED button on the computer keyboard. It is executed every 0.12 seconds. The timing of the job model assumes that the button is never pushed, which is true in the steady state and introduces negligible inaccuracies during the previous phases.

*JOB 03 monitors the performance of the inertial measurement unit, testing for conditions such as incipient gimbal lock. It is executed every 0.48 seconds. The analysis assumed that gimbal lock never occurs.

*JOB 04 monitors the status of the rendezvous radar, cycling every 0.48 seconds. The analysis assumed that the radar continues to operate properly.

*JOB 05 computes coordinate transformation matrices used by the autopilot programs. It is executed every 0.24 seconds.

*JOB 06 updates the analog meters that display landing data (such as altitude) to the crew. It is executed every 0.48 seconds. The analysis assumed that the crew has selected the display of the maximum available amount of data.

*JOB 07 monitors jet failure indicators in the spacecraft attitude control system, with a period of 0.48 seconds. The analysis assumed a 10% chance of jet failure on each cycle, which is somewhat higher than would be expected. (It should be pointed out that there are sixteen jets arranged in redundant groups on the LM, so a failure is not catastrophic.)

*JOB 08, executed every 0.50 seconds, monitors the ABORT and STAGE ABORT buttons on the control panel. The analysis assumed that they are never pushed.

*JOB 09, which is executed every two seconds, monitors the MODE and THROTTLE controls. The analysis assumed that they are never moved, which is normally true during the visibility phase.

*JOB 10 is the start of P63, the crew-initiated braking phase computer program. Its final action is to examine the exact time, in order to synchronize certain displays with the clock. The job computes the wait needed to achieve this synchronization and enters JOB 11 into the job stack (the LGC waitlist) to start after this delay. The analysis assumed a delay uniformly distributed over 0.02 to 1.01 seconds, which reflects the actual distribution.

JOB 11 continues the braking phase computations. It sets up a display with which the crew are supposed to set their event timer. JOB 14 is entered into the job stack to update this display.

JOB 12 is initiated by the crew keying in PROCEED after having reset their event timer as requested by JOB 11. It terminates by issuing a display and waiting for crew response to it.

JOB 13 is initiated by crew response to the display initiated by JOB 12, and continues the braking phase computations. At this point in the actual mission, the crew goes through the lengthy and complex procedure of aligning the inertial measurement unit, which incorporates many optical sightings, angle measurements, crew displays and responses, together with lengthy computations. Since the jobs executed during the following mission phases do not depend on the details of this procedure, and since incorporating it in detail would extend the running time of the simulation inordinately, the fine alignment programs were replaced by a completely arbitrary and artificial job, which terminates in a request for crew response to a display.

JOB 14 cycles every second, to update the display started by JOB 11, until ignition.

JOB 15 performs the final pre-descent state vector update. It is initiated by crew response to a display created by JOB 13. It is, computationally, the longest

job performed during the lunar landing, lasting about 2.2 seconds in the absence of other computer tasks. It computes the time until ignition and sets up JOB 16 to start 35 seconds before scheduled ignition of the descent engine. For purposes of the simulation, a fixed interval of 2.5 seconds was assumed for the time until the start of JOB 16, since the actual time of ignition depends on environmental conditions that were not simulated.

JOB 16 performs no computations. It is performed 35 seconds prior to ignition (TIG-35) and its LGC function is to update a "phase table" used to determine where to restart the computer in case of an error. It sets up JOB 17 to start in 5 seconds, at TIG-30.

JOB 17 is performed 30 seconds before ignition, and initializes the subsequent computer programs. It also enters five jobs into the job stack (nos. 18-22, which include the main navigation loop.)

JOB 18 is executed at TIG-5. It updates the phase table and enters JOB 29 into the job stack to start when ignition is due.

JOB 19 is executed 7.5 seconds before ignition. It sets flags which, when tested by the digital autopilot program, will cause that program to turn on the "ullage" for the main engine (a small amount of thrust, from the control jets of the spacecraft, used to settle the engine propellant in the tanks during weightlessness).

JOB 20 performs final accelerometer measurements before thrusting begins. It is entered into the stack for execution "as soon as possible" by JOB 17, and is therefore executed shortly after TIG-30.

JOB 21 is also executed shortly after TIG-30. It performs certain initialization functions for the accelerometers.

*JOB 22 is first started about two seconds after JOB 17, or at about 28 seconds to ignition; thereafter, it cycles with a period of two seconds. Its main function is to read the accelerometers. It also paces the main navigation loop; on completion, it enters two other jobs in this loop - JOB 23 and JOB 24 - into the job stack.

*JOB 23 instructs the landing radar to make an altitude measurement for use in updating the LM state vector. When the measurement is completed, in about 0.1 seconds (the analysis assumed exactly 0.1 seconds; in fact, there is a small variation about this figure), the radar will interrupt the computer and cause the execution of JOB 26.

JOB 24 performs the major navigation computations. This is the longest job that is active during the visibility phase; it can take up to 0.9 seconds to complete in the LGC. It enters three jobs into the job stack: JOB 28, entered only once;

JOB 31, each time; JOB 43, each time. (In the actual program, JOB 43 is called only when the rotation of the spacecraft requires that two or more gyro pulses be sent to the inertial measurements unit. Here, it is assumed that this is true every time, which is not seriously in error.)

*JOB 25 reads gimbal angles as required.

*JOB 26 records landing radar altitude readings.

*JOB 27 torques the gyros in the inertial measurement unit as commanded by JOB 43. It torques each of the three gyros in order, by sending out a string of pulses to it. A skewed distribution of pulses with a peak at 300 pulses was assumed for the output distribution.

JOB 28 commands the landing radar to its descent position.

JOB 29 is performed at the scheduled descent engine ignition time. It verifies that the crew has permitted ignition and turns the engine on. The job analysis assumed that the crew has, in fact, permitted ignition; this would normally be the case.

JOB 30 is performed 26 seconds past ignition. It calls for full engine thrust, sets flags for use by other programs, and initiates the periodic monitoring jobs JOB 08 and JOB 09.

*JOB 31 takes landing radar velocity measurements. It is performed every two seconds, as part of the navigation loop paced by JOB 22. This job starts JOB 25 directly, and the landing radar hardware interrupts the computer about 0.1 seconds after this job to initiate JOB 42. (An interval of exactly 0.1 seconds was assumed; there is actually a slight variability in this interval.)

JOB 32 through JOB 35 actually represent the same section of the program by different names. This section tests the position of the landing radar to determine if it has reached its descent position yet (as commanded by JOB 28). The test is repeated at one-second intervals, starting seven seconds after the command was issued. In the actual program, the test is repeated fifteen times, and an alarm is issued if it is not successful. In the modeling, it was assumed that the landing radar will have reached its goal by the time of the fourth test (11 seconds after the command). (This sequence of jobs could have been represented equally well by one job model with a counter.)

*JOB 36 is the digital autopilot (DAP) job. It repeats at 0.1-second intervals. It initiates JOB 38, to start after an interval which is a function of spacecraft control requirements (here assumed a random variable over 0 to 0.085 seconds); JOB 39, every 20th time (i.e., every 2 seconds) and JOB 40, every other time (i.e., every 0.2 seconds.)

*JOB 37 is performed each time the crew exercises its option to redesignate the site at which the LM will land. It sets flags used by JOB 24 to determine if computation of a new trajectory is necessary. The statistics of iteration rate were chosen to model the crew as making major changes in the landing site, by moving the controller many times, at the start of the visibility phase and then tapering off to a lower, approximately constant rate until landing. The distribution used in the simulation was triangular, with mean interval rising from 0.4 seconds to 5 seconds by steps of 0.05 seconds each time the controller is used. The base of the triangle starts at 0.3 seconds and the distribution is symmetric about its mean.

*JOB 38 turns off the reaction control system (RCS) jets, which control spacecraft attitude, as required by the autopilot.

*JOB 39 calculates spacecraft acceleration from data about engine thrust and spacecraft mass.

*JOB 40 controls the gimbals that direct the thrust axis of the descent engine.

*JOB 41 computes altitude and velocity landing radar beam direction vectors after the LR has been moved to its descent position.

*JOB 42 reads landing radar velocity data. It is initiated by an interrupt, at completion of the radar reading commanded by JOB 31. It stores the radar data, and commands another reading immediately, which in turn causes another interrupt after an interval of 0.1 seconds. (This interval was assumed for the simulation; in the actual mission the interval can vary by about 0.01 seconds from this figure.) After five readings, the program performs calculations on the data and terminates.

*JOB 43 computes required torques for the gyroscopes that measure spacecraft attitude, and initiates JOB 27 that will send these torque commands out.

Table A-1 summarizes the numerical characteristics of these jobs. For each job, it lists the number of words obtained from central memory (GET) and returned to it (PUT); the job duration in basic AGC instructions (INST) and interpretive milliseconds (MSEC); and, when a job can vary in its numerical parameters, a description of the variability.

TABLE A-1

LUNAR LANDING PROGRAM STATISTICS

#	GET	PUT	INST	MSEC	COMMENTS
1	16	5	30	0	when displaying
			23	0	when not displaying
			21	0	when finishing a display
2	1	0	6	0	
3	6	0	36	0	
4	6	0	57	0	
5	2	7	130	0	
6	13	9	49	0	half the time (random)
			66		half the time (random)
7	2	1	26	0	nine-tenths of the time (random)
	5	5	72	0	one-tenth of the time (random)
8	2	0	84	0	
9	3	0	85	0	
10	4	11	162	0	
11	5	3	22	0	
12	0	1	7	0	
13	25	15	100	0	arbitrary assumptions (see text)
14	5	3	23	0	
15	67	26	499	2182.99	
16	0	0	7	0	
17	0	29	112	0	
18	1	1	8	0	
19	1	1	7	0	
20	11	10	300	0	
21	8	5	56	26.18	
22	3	16	57	0	
23	3	4	47	0	

TABLE A-1 (CONT)

LUNAR LANDING PROGRAM STATISTICS

#	GET	PUT	INST	MSEC	COMMENTS
24	100	56	4839	323.86	before LR move
	100	56	4840	323.86	during LR move
	100	56	4839	323.86	after LR move
	100	57	4842	323.86	after ignition
	121	56	8529	429.32	visibility phase, no retarget
	125	63	10681	646.93	visibility phase, with retarget
25	3	3	7	0	
26	6	4	122	0	
27	4	4	67	0	first 3 times (torquing gyros)
	1	1	36	0	last time (cleaning up)
28	1	2	29	0	
29	8	13	98	0	
30	3	12	93	0	
31	1	6	46	0	
32	0	1	5	0	
33	2	1	10	0	
34	0	1	7	0	
35	0	1	17	0	
36	52	43	617	0	every 20th time
	52	42	569	0	other even-numbered times
	52	42	547	0	odd-numbered times
37	1	1	8	0	
38	6	7	37	0	one-third of the time, (random)
	6	8	37	0	two-thirds of the time, (random)
39	22	14	212	0	
40	10	4	464	0	
41	26	21	3499	32.61	
42	10	6	67	0	first four times
	11	4	100	0	fifth and last time of cycle
43	8	8	165	0	

APPENDIX B

JOB MODEL SUBROUTINES

The program generator input deck for a typical job model subroutine (not actually one from the lunar landing, but still representative of them) is shown in Fig. B-1, which is reproduced from the output of the program generator. The corresponding Fortran subroutine, as produced by the program generator, is reproduced as Fig. B-2.

Line by line, the program generator input describes the following job:

1. The job is identified as Job 12. This identification is required by the simulation system.
2. A descriptive comment, which will be inserted in the Fortran subroutine without change.
3. The job is to cycle (i.e., to be executed repetitively) at intervals of 0.5 seconds (i.e., twice per second).
4. The job obtains ("gets") five words of data from central memory before execution.
5. It reads five words of data from external devices.
6. It executes 35 instructions.
7. When done, it returns ("puts") 6 words of data back in central memory.
8. It inserts Job 15 into the job stack, for execution one second after the present job was called for execution (which may not, if delays intervened, be precisely the time at which it actually executes).
9. It writes 7 words of data onto external devices. (This figure is perhaps atypically high for the lunar landing.)
10. The input deck terminates.

The same job is described by the Fortran listing of Fig. B-2, but in a format which is less clear. By groups of lines, this program does the following:

Line 1 identifies the subroutine to the system.

INPUT CARDS:

```
JOBNO      12
COMMENT    THIS IS A TEST JOB
CYCLE      0.5
GET        5
READ       5
INST       35
PUT        6
INSERT     15          CALL       1
WRITE      7
END
```

Fig. B-1

Typical Job Model Subroutine: Program Generator Input

JOB MODEL SUBROUTINE:

```
      SUBROUTINE JOB12(*,*,*)
C THIS IS A TEST JOB
      REAL*8 TCALL
      COMMON/ALL/I
      GO TO (11,12,10,14,15),I
   10 RETURN
   40 RETURN 2
   11 CALL ACSAVE(640,TCALL)
      CALL GET(5)
      CALL READ(5)
      CALL RELBUS
      CALL RUNJOB(35,0)
      RETURN 1
   12 CONTINUE
      CALL PUT(6)
      CALL WRITE(7)
      CALL INSERT(12,TCALL+0.5D0)
      CALL INSERT(15,TCALL+1D0)
      CALL TERMIN
      RETURN 3
   31 FORMAT(I2,3Z16)
   14 READ(5,31) ITEST,TCALL
      IF(ITEST.NE.12)CALL FOULUP(12,ITEST)
      RETURN
   15 ITEST=12
      WRITE(7,31)ITEST,TCALL
      RETURN
      END
```

Fig. B-2

Typical Job Model Subroutine: Program Generator Output

From there through the line labeled with the number "40", miscellaneous bookkeeping functions required by the system are performed.

From the line labeled "11" to just before the line labeled "12", the subroutine performs the functions associated with starting execution. It saves the time for which it was called in a register referred to as "TCALL". It then, by means of a mnemonically-named subroutine calls, gets 5 words of data, reads 5 words, releases the bus, and informs the simulation that it will run for 35 instructions. Finally, it returns control to the central simulation loop.

From the line labeled "12" to just before the line labeled "31", it performs the functions associated with terminating execution. It returns 6 words to central memory, writes 7 words, inserts itself (JOB12) into the job stack for execution one-half second from the time for which it had been called, and inserts JOB15 into the job stack for execution one second from that time. It then terminates and returns control to the central simulation loop.

From the line labeled "31" through the end of the program, the instructions perform functions associated with the creation of a "restart deck" to permit restarting the simulation from the point where a previous run terminated, and with performing such a restart.

Additional functions not included in this sample job include simulating an external interrupt, stochastic variation of job parameters (such as a job which might vary its execution time in a way that depends on the values of certain variables, which cannot be determined from examination of the program listing) and more. These, too, can be handled within the program generator language. For the few cases in which this language is inadequate—such as complex statistical functions, or communication with other subroutines—it can still be used to provide a "base" which incorporates the required bookkeeping instructions, and onto which the additions may easily be grafted.

REFERENCES

1. Vacca, Gene A., Philip L. Phipps and Thomas E. Burke, "Mission Influences on Advanced Computers", Aeronautics and Astronautics, 5 no. 4, p. 36-40, April, 1967.

2. MIT Instrumentation Laboratory, Vol II-"Multiprocessor Computer Subsystem", Control, Guidance and Navigation for Advanced Manned Missions, MIT Instrumentation Laboratory report R-600, January, 1968.

3. Alsonso, Ramon L., Albert L. Hopkins, and Herbert Thaler, A Multiprocessing Structure, MIT Instrumentation Laboratory report E-2097, 1967.

4. Alsonso, Ramon L., Glenn C. Randa, "Flight Computer Hardware Trends", Aeronautics and Astronautics, 5 no. 4, p. 30-34, April, 1967.

5. Seamans, Robert C., "Into Space - and Why" Interavia, 21, p. 1479-1494, October, 1966.

6. MIT Instrumentation Laboratory, Vol. II-"PGNCS Operations", Guidance Systems Operations Plan, MIT Instrumentation Laboratory report R-577, August, 1967.

7. NASA, Apollo Guidance Program Symbolic Listing Information for Block 2, Working Paper NAS 9-4810, Manned Spacecraft Center, Houston, Texas, January, 1967.

8. MIT Instrumentation Laboratory, Apollo Group: Program SUNDANCE for LM Guidance Computer, revisions 224 and 230, October, 1967.

9. Savage, Bernard I., and Alice Drake, AGC4 Basic Training Manual, MIT Instrumentation Laboratory report E-2052, January, 1967.

10. Muntz, Charles A., Users Guide to the Block II AGC/LGC Interpreter, MIT Instrumentation Laboratory report R-489, April, 1965.

11. Katz, Jesse H., "Simulation of a Multiprocessor Computer System", AFIPS Conference Proceedings, 28 (1966 Spring Joint Computer Conference, New York, New York, AFIPS , 1966), p. 127-139.

12. Wallace, Victor, and Richard S. Rosenberg, "Markovian Models and Numerical Analysis of Computer System Behavior", AFIPS Conference Proceedings, 28 (1966 Spring Joint Computer Conference, New York, AFIPS, 1966), p. 141-148.

13. Merikallio, Reino A., and Fred C. Holland, "Simulation Design of a Multiprocessor System", AFIPS Conference Proceedings, 33 (1968 Fall Joint Computer Conference, New York, New York, AFIPS, 1968), p. 1399-1410.

14. Scherr, Allan L., An Analysis of Time-Shared Computer Systems, MIT project MAC Report MAC-TR-18, June, 1965. (Also: Ph.D. thesis, MIT Department of Electrical Engineering).

15. Fishman, George S., and Philip J. Kiviat, "The Statistics of Discrete-Event Simulation", Simulation, 10, p. 185-195, April, 1968.

16. Huesmann, L. Rowell, and Robert P. Goldberg, "Evaluating Computer Systems Through Simulation", Computer J, 10, p. 150-156, October, 1967.

17. IBM, Application Program Manual, General Purpose Systems Simulator III Introduction, Form B20-0001-0, 1965.

18. Markowitz, Harry M., Bernard Hauser, and Herbett W. Karr, SIMSCRIPT-A Simulation Programming Language, Englewood Cliffs, N.J.: Prentice Hall, 1963.

19. Morse, Philip M., Queues, Inventories, and Maintenance, New York: John Wiley and Sons, 1958.

20. Howard, Ronald A., Dynamic Programming and Markov Processes, Cambridge, Mass: MIT Press, 1960.

BIOGRAPHICAL NOTE

Efrem G. Mallach was [redacted]. He attended elementary and high schools in White Plains, N.Y., Israel and Greece, returning to the United States to graduate fron White Plains High School in 1960.

He entered Princeton University in the fall of 1960, and obtained the B.S.E. degree with high honors in aeronautical engineering in 1964. At Princeton, he was appointed a Unitersity Scholar and elected to Phi Beta Kappa.

Entering M.I.T.'s department of Aeronautics and Astronautics with a National Science Foundation fellowship in 1964, he concentrated in the area of guidance and control. He found summer and part-time employment at the M.I.T. Instrumentation Laboratory, which eventually led to his choice of a thesis topic. He was elected to Sigma Xi, Tau Beta Pi and Sigma Gamma Tau, and graduated in the spring of 1969.

Dr. Mallach was married to the former Linda [redacted] on June 12th, 1966; they have one son, Douglas, [redacted]. Following his graduation, he will work with the Boston office of Computer Usage Co., Inc.