THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

A STOCHASTIC MODEL OF PARALLEL AND CONCURRENT PROGRAM EXECUTION ON MULTIPROCESSORS

B. A. Makrucki
T. N. Mudge

CRL-TR-3-82

OCTOBER 1982

Electrical and Computer Engineering Department
Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1] Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

ABSTRACT

This report summerizes a model developed to allow the evaluation of parallel program execution on multiprocessors. The model is intended for MIMD algorithms in which the individual processors are coupled through their programs' interaction with memory. The model is not intended for SIMD algorithms. Specifically, estimates of processor utilization, execution times of programs or subprograms, and memory bandwidth can be obtained from the model. Earlier research has concentrated on the last of these quantities and a body of research, which might be termed "memory interference models", has evolved. The work reported here goes one step further by allowing the programs to be included in the model. Consequently questions about the performance of the processors running the programs can also be answered.

TABLE OF CONTENTS

1. Introduction.

In order to increase computing power in an economical fashion, multiprocessor computer systems have been developed. Multiprocessor architectures allow system designers flexibility and typically modularity; computing power may be added incrementally and over time as system needs grow. Multiprocessor systems allow system designers to obtain reliability in a simple manner, failure of a few processors may often be tolerated. Implementation of multiprocessors using VLSI technology makes their use even more attractive when compared to very high speed uniprocessor alternatives.

Multiprocessors used in such environments might logically be configured to run in the MIMD (multiple instruction stream, multiple data stream) mode of operation. This way, processors may share common random access memory (RAM) and disk memory space. By making RAM common to all processors in the system, memory space usage may vary dynamically as user programs require. A less versatile alternative could consist of processor-memory pairs which communicate with each other using simple transmission systems, this could be termed a tightly coupled computer network. The use of common memory also allows processors to share data files and data bases which would be stored in secondary disk memory.

Multiprocessors may also be used to increase the execution speed of algorithms which exhibit parallel or concurrent components. MIMD systems may be used to decrease the execution time (over serial execution time) of many algorithms which are inherently parallel. For example, some database searching and artificial intelligence problems lend themselves to a parallel environment. MIMD systems perform well on algorithms which entail list processing where elements of lists are to be processed similarly. SIMD (single instruction stream, multiple data stream) systems work very well on problems where each element of a list is to be processed *identically.* SIMD systems are generally not as versatile as MIMD systems.

MIMD system performance is an important characteristic for comparing alternative system designs, but MIMD system performance often depends on the programs load used to characterize the system's performance. The quantification of MIMD performance is valuable in that it may be used as an aid by MIMD hardware and software designers. For example, a mathematical model which parameterizes system characteristics such as the number of processors in the system, the number of memory modules in the system, interconnection characteristics, and processor-memory speeds could be used to evaluate the effects of systems changes. That is, a model may be used to answer questions such as: how much extra computing power is gained by the addition of another processor or memory module; if memory speed is increased by a certain amount, how much faster will the system run; etc. Such questions may be answered by a mathematical model (of sufficient capabilities) without implementing a system or writing programs to use as tests and simulators.

The complexity of MIMD systems and parallel program execution makes a mathematical model of parallel/concurrent program execution valuable as a tool for guiding program design as well as answering system design questions. For example, in multiprocessor operation, when multiple processors access system memory, interference occurs and system performance may often be modeled as being random. Due to these random effects, program design is not always clear, seemingly trivial program modifications may increase system performance. The amount of increase should be obtainable from a good model.

This document describes the present state of such a model that may be used for the quantification of program execution on a (described) MIMD system. For simplicity, a single system configuration will be described (in section 2) that is believed to be representative of a high-performance MIMD system suited for parallel program execution and a general purpose environment as would be seen in a general computer installation. Modifications to the basic model presented may be made to suit specific system configurations. For example, an Intel 432 system could be modeled with appropriate changes in the basic model presented here.

The remainder of the section describes previous work done on multiprocessor performance evaluation (for the type of multiprocessor system considered here) and general modeling concepts. Section 2 describes the system configuration and operation, including program execution on the system. Section 3 describes the model which is based on modeling program execution using stochastic finite state machines; and the performance measures of interest. Section 4 is an overview of results from stochastic process theory that will be used in analyzing the model. Section 5 derives expressions for the performance measures described in section 3 based on relationships discussed in section 4. Section 6 presents two calculations which complete the basic model (as of the date of writing) analysis. Section 7 presents example uses of the model, and shows simulation results that were used to verify and test the model's accuracy. Section 8 describes several topics for future work related to the basic model.

1.1. Modeling vs. Simulation and Program Testing.

When a model is constructed, its usefulness must be evaluated against alternative techniques that may be used to obtain problem statistics. Analytic models would be of little use if alternative techniques provided more information. This section describes alternative performance evaluation techniques that could be considered: system simulation; and running test programs in order to design a good parallel program.

Simulation of multiprocessor systems of the type considered here tends to be costly (section 7 describes a program simulation example) and provides insight only into the case simulated. Simulation of systems which possess a stochastic nature often requires several simulator runs in order to obtain confidence intervals; this can be expensive. Simulation of actual program execution, in general, is expected to be very costly. It is far easier to solve model equations then it is to simulate the same situation (assume that the model is somewhat approximate, as will be seen a model which computes full system behavior is virtually useless due to computational complexity).

A general model, and a program that solves its equations, may be used to evaluate specific situations by specifying parameters of the model and running the model solution program. The alternative simulation approach is to write a general simulator and use it as the model solution program would be used. Simulations though are generally expensive to run, especially if they are general purpose simulators and good stable statistics are desired.

Another alternative to analytic modeling of program execution on multiprocessors is to write the program and then conduct tests of the program on the actual machine. This approach is useful if the machine is available (it may still be in the design phase in which case an analytic model may be used to quantify design choice impact, this is a very useful aspect of modeling in itself) and when the algorithm is easily implemented. The implementation of algorithms on parallel machines is still an area of research and algorithm partitioning is often not obvious. An analytic model may be used to test various algorithm partitioning ideas without the necessity of writing a full program (an actual program would probably be required for a general simulator just mentioned and as such is another drawback of the simulation approach to program/system performance evaluation). For example, the model presented here does not require an actual program to be written, characteristics of the program at the flowchart level are used. Even if all characteristics of a given algorithm partition are not known, approximate values may be used where needed and meaningful comparisons may still be made.

Analytic models provide insight into system performance that is not obtainable through other means. For example, closed form special case results are often obtained from general formulations. Analytic models also provide details as to the causes of system behavior. Such understanding is not offered by experimental means. Experiments are useful for verifying and testing analytic models.

1.2. Previous Results.

Multiprocessor models have been developed in an effort to characterize system performance based on system measurable quantities such as rates of emission, the fraction of time that an event occurs, etc. Past models have typically dealt with the relationship between these measurable quantities.

Consider a multiprocessor system composed of P processors, M memory modules used for data (and possibly instruction storage) and a simple PxM crossbar connection between the processors and memories. Each processor may access any of the M memory modules. Previous models have been devised to study the problem of memory interference. Memory interference has loosely been termed the interference which processors inflict upon each other when concurrent access to a common memory module is required. The past study of memory interference can be categorized into two basic classes: discrete time models; and continuous time models.

1.2.1. Discrete Time Models.

Most previous models have been of this type; here the system is assumed to be clocked and the models attempt to obtain performance characteristics based on presumptions about program execution at the clock cycle (or basic time cycle) level. [SkA69] was (about) the first published work on a discrete time model of multiprocessors, they assumed clock cycle level of operation and modeled the effects caused by memory interference. Their model used Markov chain techniques and even modeled conflict resolution probabilities. Unfortunately their technique becomes intractable even for systems with 4 processors and 4 memories. Their model requires 65 states for a 4 processor, 2 memory system.

[Str70] introduced another model of basically the same system and derived closed form results for special cases. [MuM82a] obtained the same basic results but generalized them using a simple discrete time model. [Bha75] formulated an exact computation (for an approximate model, see section 1.3) and approximations for general sized systems by using Markov chains.

[SeD79] obtained some approximate results for systems where memory access patterns tend to latch onto their present modules, their results are rather approximate. [Rau79] presents a computation technique based on a decomposition approximation presented in [BaS76].

In the discrete time classification, in the author's opinion, the most versatile and accurate model so far has been developed in [Hoo77].

1.2.2. Continuous Time Models.

These models are often basically the same as the discrete time models but approximations are made using ideas from queueing theory [Kle75, Coo72, Tak62, GrH74]. [BaS76, Smi74] made the appropriate approximation that for large systems, memory queues may be approximated as M/G/1 queueing stations. These results noted that variation of certain system characteristics (connection time distributions) has a small effect on performance measures. See section 7.1 for a further explanation.

[McC73] used a Jackson type queueing network to obtain analytic results for a fully continuous time model. [MaG81] obtain results using Markov processes on systems which employ multiple busses as their connection facility. [MaM81] used queueing results to model MIMD systems under some simplifying assumptions relieved here.

1.3. General Modeling Ideas.

A fact becomes evident from examining previous work in the area. Known[1] work can basically be summed up as follows: approximate processor/program/memory behavior in a simple manner and then seek an accurate solution for this model. This approach leads to simplistic modeling of complex phenomenon with great care taken in solving the simple model. The model presented herein is believed to be different in that it seeks to model processor/program/memory behavior as realistically and accurately as possible, then it attempts to find an acceptable solution to the model. This basic difference leads to a more realistic and versatile model than previous ones and even compares favorably with Hoogendoorn's discrete time model. The model presented here also allows more information about program and system behavior to be obtained, this is obtained by using a more realistic model of program execution than previously used.

The model presented here may best be classified as a continuous time model and as such it seems reasonable that its greatest error (due to approximation breakdown) should occur when modeling behavior best described by discrete time models. The model presented may be used quite easily to model the same phenomenon as the simple models previously developed.

[1] To the authors.

2. System Configuration and Operation.

This section describes the basic configuration of the MIMD (multiple instruction stream, multiple data stream) multiprocessor computer system studied herein. The system is intended to serve as a general-purpose parallel processor, or possibly as a multiuser, multiprogrammed, general-purpose system.

To achieve acceptable performance from a multiprocessor designed for parallel processing, reasonably concurrent access (by all processors) to common data is required. Multiport memory systems would be the ideal solution to achieving highly concurrent memory access, unfortunately memory systems of this type would be very expensive and large. Low density multiport memory parts are available but the number of ports provided is smaller than required by system designs [MuM82d].

It is desirable to have common data appear the same to all processes/processors[2]. That is, each process should see an identical version of common data. Figure 1 diagrams the situation.

Another important aspect of parallel processor system operation and design involves the implementation of data access locking. That is, the implementation of access control structures (e.g., semaphores, monitors, guarded regions, etc. [Hoa74, Han78]). For example, it is often important for a process to have full control of list pointers, list elements, etc. This control will be seen to be implementable in hardware using certain memory mapping techniques.

[2] "Processes" and "processors" are equivalent terms if each processor runs only one process, this will be seen to be the parallel program execution mode of processor operation. They are different in the case where a physical processor runs multiple processes (in a time-multiplexed manner), this will be termed concurrent program execution. In this case the term "processor" will be used to reference the physical processor, and the term "process" or "program" will be used to reference a single task executing on a processor.

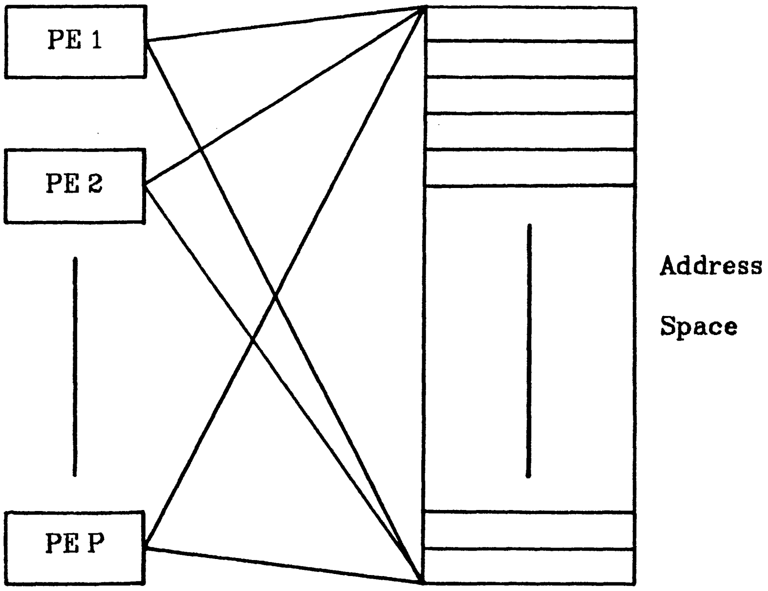


Figure 1. Global data as seen by all processors.

2.1. System Configuration.

The multiprocessor computer system to be described and modeled is shown in Figure 2. The system consists of P processors, M global memory modules, D disk units, a processor-memory interconnection network (ICN), and a memory-disk ICN (the disk units are not shown and are not of major concern here). It will be assumed that disk accesses are handled through global memory in a DMA type manner, disk data transfers are handled by the memory-disk ICN. DMA is assumed because of its efficiency and wide use in most systems. A discussion of these subsystems and their operation follows.

The term "processor" will be used to describe devices such as general-purpose processors, special-purpose processors (such as I/O processors) or other such devices which perform operations on data. The following discussion applies primarily to processors which execute programs using data stored (at least partially) in global memory.

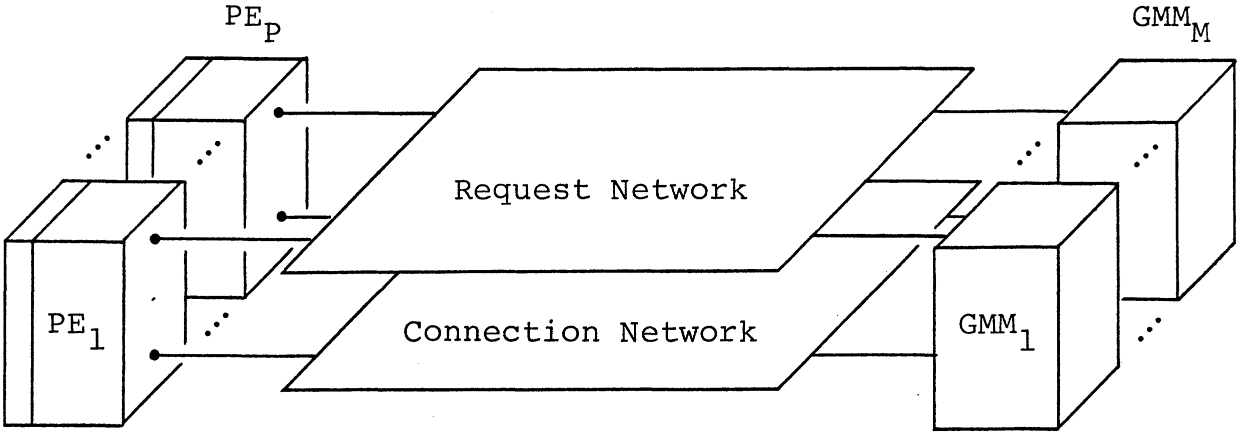


Figure 2. Multiprocessor system architecture.

Processors in the multiprocessor system read their instruction streams from local memories shown in Figure 3. Local memory i is accessible only by processor i (and possibly a global system controller). Each processor combined with its local memory will be termed a processing element (PE). PE's are suitable for implementation in VLSI, a PE may be constructed in relatively compact form.

Processors also use local memory to store temporary (or local) intermediate results of computations. The use of local memory makes global memory (which is accessible by all processors) usage efficient in that under proper data storage conditions memory interference may be minimized (it is *not* a good plan to store instructions in global memory unless absolutely required because this causes great memory interference to occur, an example use of the model in describing this phenomenon will be seen in section 7.).

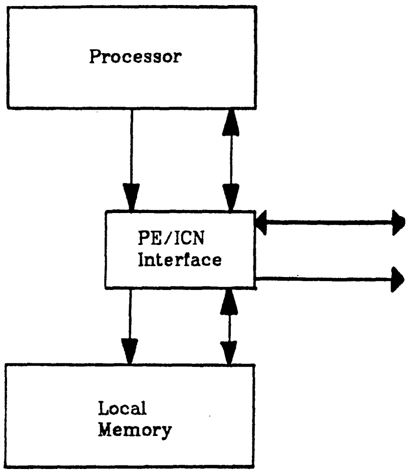


Figure 3. PE configuration.

Global memory (GM) is generally used to store global data structures such as arrays, lists, trees, etc. (as will be seen GM may be used in a general-purpose, multiuser, multiprogrammed environment; for the time being consider parallel program execution where each processor executes only one program). If, though, local memory storage capacity is exceeded by local storage requirements, processors may be forced to use GM for local storage. In this case it is best for each processor to store overflow data in a preferred GM module (GMM); processor i would store its overflow data in GMM i. This technique minimizes memory congestion (interference) due to local memory overflow effects. This fact will be evident from the model and analysis described herein.

When a processor executes a GM reference instruction (i.e. it makes a GM reference) it emits *request packet(s)*. A request packet consists of of at least a processor number (in the parallel program execution environment). That is, a request packet (termed a *request*) emitted by processor i, destined for one of the GMM's, could be as simple as a tag representing the number i. Packets emitted by PE's are sent to the processor-memory ICN's.

The processor-memory ICN's serve to: (1) transfer requests from PE's to GMM's, this ICN will be termed the request network (RN); (2) connect PE's to GMM's for the actual data transfer operations, this ICN will be termed the connection network (CN). The RN is a *packet switched* network while the CN is a *circuit switched* network [Sie80].

For simplicity[3] both ICN's will be assumed to possess the totally connected property: all 1-1 PE-to-GMM connections may exist simultaneously; the networks are assumed not to block connections with insufficient connection capabilities. The crossbar network meets these requirements and is implementable in VLSI for small to moderately sized systems, up to about 32 PE's and 32 GMM's. Consider next the structure of a GMM.

GMM's receive requests from the RN, a GMM configuration is shown in Figure 4. Requests from PE's (transmitted through the RN) are placed into a FCFS (first come, first served, or FIFO) queue. Since requests consist of simple tags (of at least width $\lceil \log_2 P \rceil$ bits), a FIFO queue may be constructed in VLSI using simple shift register structures. The length of GMM queues depends on implementation constraints (for example, if the queue and GMM controller are implemented on a single chip using VLSI technology, space factors could limit the length of the queue) and system behavior, as will be seen later.

2.2. System Operation.

Processor operation divides basically into two modes:

1) parallel program execution mode
2) concurrent program execution mode

[3] This assumption is used here in order that initial insight may be obtained without the complication arising from the use of other ICN structures. This assumption is *not* essential, as will be seen later in the section which describes the SMP model analysis.

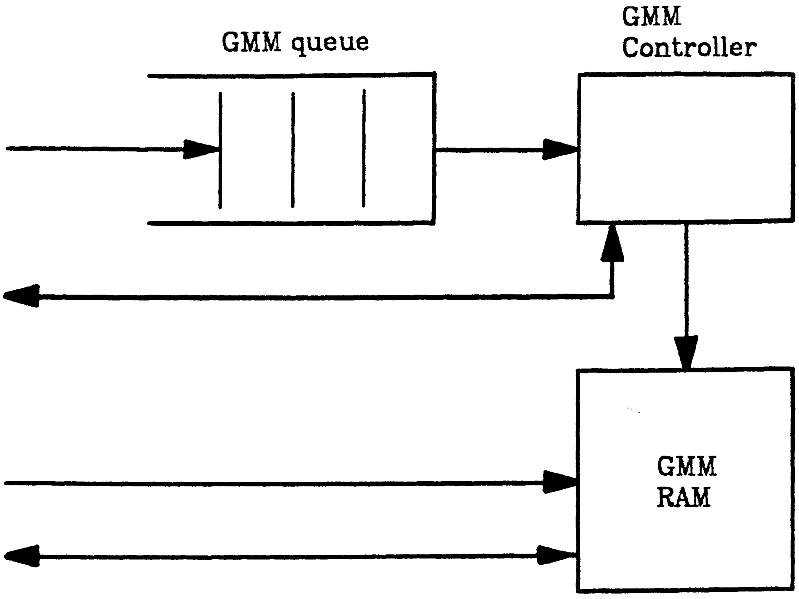


Figure 4. A global memory module architecture.

When operating in the parallel program execution mode, processors emit requests and then *wait* for requested connections (to GMM's) to be established. This mode of operation is equivalent to running one process per processor. In this mode of operation processors become idle for some period of time each time a GM reference is made. A measure of system effectiveness for parallel program execution is the *fraction of time that processors are idle.*

In the concurrent program execution mode of operation, processors are multiprogrammed, several users use each processor on a time-sliced like basis. In this mode many GM usage strategies are available, an obvious strategy is to use GM and disks as secondary memory. Page faults generated by user programs (whose working sets would be stored in the local memory belonging to the processor on which the user's tasks execute) are references to GM or disk memory storage (which would be buffered through GM).

This treatment of user program memory references requires an answer to the question as to whether or not it is best to context switch tasks when GM references are made. For example, if context switch times (the time required to complete a context switch) are small compared to mean request queueing times (processors emit requests which wait in queue for use of the GMM RAM), then it is best (on the average) to treat GM references as page faults because tasks may be swapped quickly enough that some useful processing may be done before the requested connection becomes available. See Figure 5.a for a timing diagram of this situation. If, though, context switch times are about equal to or greater than the mean request queueing time then treating GM references as page faults leads to a form of thrashing (because as soon as the context switch is complete, the previously requested connection becomes available, see Figure 5.b for average timing), this will be termed GM thrashing. An important use of a model of system behavior determines when GM thrashing will occur.

Notice that the concurrent program execution mode of operation leads to multiple requests from the same processor being present in the GM system simultaneously. Hence simple processor numbers will probably not suffice as request packets, a packet number might also be required.

At the time of writing, the model and examples developed pertain primarily to the parallel program execution mode of operation.

To the programmer[4] the GM may be viewed as a set of individually addressable memory modules. In memory systems designed for high concurrency, memory modules are often selected using the least significant $\lceil \log_2 M \rceil$ bits of memory addresses, this is the interleaved addressing scheme[5]. Other memory selection schemes exist, one application of the model presented is the quantification of effects on program execution (and system

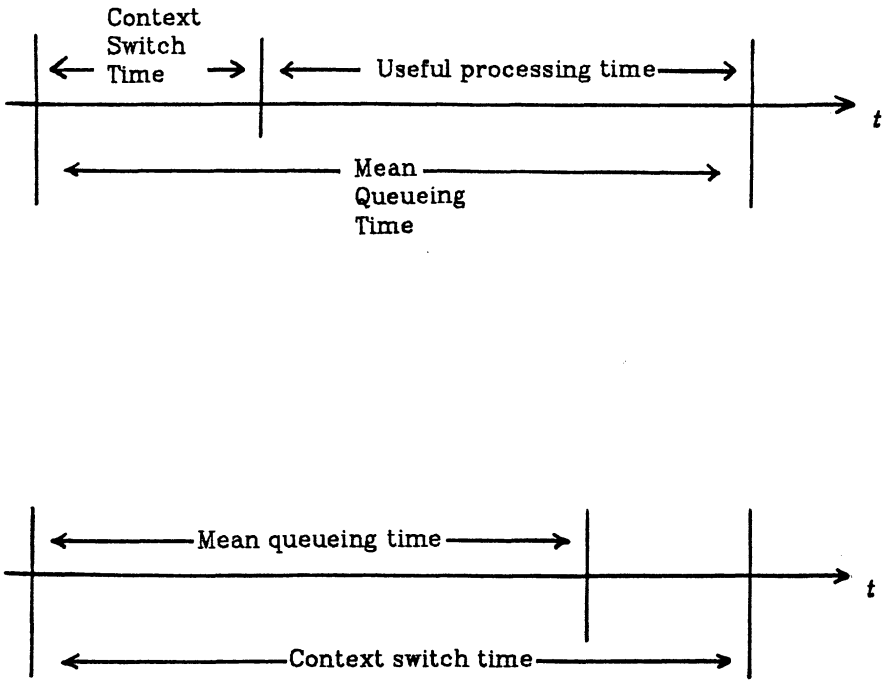


Figure 5. Global memory timing with context switching.

performance) of choices such as memory mapping and data layout in GM. The model acts as a performance indicator of program design. Two addressing schemes are obviously available: the *interleaved* mapping technique, and the ***block*** mapping technique, where the most significant $\lceil \log_2 M \rceil$ bits of the GM address bus are used to address the GMM's.

Block mapping has an efficiency advantage over interleaved mapping in that once a connection has been established (from say PE i to GMM j) all locations in GMM j are accessible in a linear manner. [Note that the same *effect* may be achieved with interleaved GM but addresses within GMM's are $\lceil \log_2 M \rceil$ locations apart, the term "interleaved" normally applies when each GMM is used for a single word access, here the two mapping techniques will be taken

[4] Or compiler for that matter.

[5] The term "interleaved" is often associated with single memory word accesses.

to be different.]

Block mapping also serves to implement data access locking. For example, elements of lists which may consist of several words may be stored efficiently by storing a single element of the list in a single GMM (obviously the list wraps around the GM system). This storage scheme allows elements of the list to be modified without concern about "semaphore like" locking of the data. The circuit switched property of PE-to-GMM connections intrinsically (along with block mapping) supplies a locking facility in hardware. An example presented later describes the use of this facility more fully.

3. The Semi-Markov Process (SMP) Model.

The SMP model describes MIMD system behavior at the processor/memory/switch (P/M/S) level of system operation. For simplicity the crossbar ICN's will be taken to have zero switching times. This is certainly an approximation but there are two reasons for its applicability:

1) To assume otherwise would complicate matters without contributing much in the way of new understanding of system behavior.

2) Simple delay times could be added to account for ICN propagation delays, this addition again simply complicates matters without yielding new results.

Due to the negligible effects of ICN operation, the model will consist of describing the system at the processor/ memory (F/M) level of operation. That is, events in the system consist of the interaction between PE's and GMM's (disk interaction is yet another level of complexity that might be studied, herein disk interaction is not *explicitly* modeled).

3.1. Processor/Program Modeling.

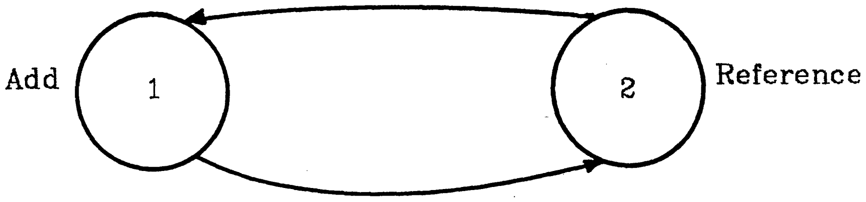
The basic technique used to model program execution (on a single PE) is to view program execution (termed processor behavior) as the sequencing of a finite state machine (FSM). From the viewpoint of processor/memory interactions, processors may be in one of two states: computation states; and GM reference states.

Computation states are states in which processors are doing purely internal operations; that is, the PE is operating as an autonomous PE. Computation states may actually consist of very complicated phases of computations where local memory is used for storage of intermediate results. After completing the local computations, the PE enters its next state (as seen at the P/M level of system operation).

GM reference states consist of states where processors reference GMM's. Upon entry to a reference state (this will be the term used for a GM reference state) a PE emits requests for connections to GMM's. After emitting a request, the PE waits for the connection to be established. Once the connection has been established, the PE uses the requested GMM as required. After completing the required GMM use, the PE *releases* the connection. It is at this point in time that the PE leaves its reference state. It can then enter a computation or reference state.

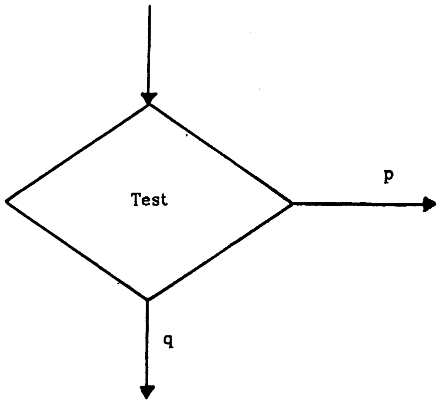
If a PE emits multiple requests upon entering a reference state, it uses connections in the order in which they become available (i.e., their acknowledges arrive, but this is not a stiff requirement of the model). Only after all requested connections have been used does the PE enter its next state.

The P/M level of program execution description reduces a program flowchart (which may be interpreted as a FSM state transition diagram) to a P/M level state transition diagram. For example, suppose it is desired for one of the P processors to add a series of numbers stored in GM. When the process is in operation, its P/M state transition diagram looks like:



The computation state (1) is occupied for the time required to complete an add operation. The reference state (2) is occupied for the time required for a GM read and the time spent by the request in the referenced GMM queue.

The model described assumes that as an approximation to program conditional transitions (i.e., conditional transitions in the program flowchart), probabilities may be determined that represent the probability that a particular branch of a conditional is taken. For example, suppose that a conditional test looks like (in the P/M or flowchart level of program description):

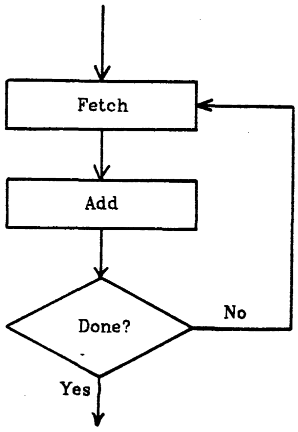


Then it is assumed that p and $q = 1 - p$ represent the probability that their respective arcs are taken upon evaluation of the test.

This assumption of probabilistic transitions is not unreasonable in that it may be representative of reality in two obvious ways:

(1) When program behavior is approximately random, i.e., when p represents the fraction of time that its arc is taken and each entry to the conditional test is relatively independent of past and future entries to this state (the amount of correlation allowed before this assumption breaks down is now unknown but might be a topic for study).

(2) When inaccuracy may be tolerated so that an answer may be obtained. For example, if each processor is used to add $k \gg 0$ numbers, a suitable approximation to a string of k read/add states might be:



This approximation reduces the number of states to 3, two computation states and one reference state. For large k, the savings is quite significant (as will be seen state space reduction techniques are valuable). Certainly this is an approximation because only k cycles through the loop will be taken by the program, but probabilistic approximations set only the *mean* number of loop executions to k. There is a non-zero variance on the number of times the loop will be executed if the probabilistic approximation is used, this leads to some inaccuracy, the tradeoff is cost (of computations as seen later) verses accuracy.

In general, it may be advantageous to use the probabilistic loop to replace linear chains (an unwrapped loop) of loop states. This is most attractive when the number of loop iterations is large. It is precisely these situations in which the model is to be used. The SMP model is designed for use in *large scale computations* (i.e., ones in which the problem is computation intensive) where the system reaches steady-state (steady-state will be described more fully later). Modeling small scale programs is not really required because (typically) the solution of a good model (of small scale programs) may actually be more expensive than running the program, this relates to the discussion of analytic models in section 1.

A good model of small scale program execution basically entails the transient solution of system description equations. Steady-state solutions will not suffice because in small scale programs (i.e., ones with short execution time) the system probably does not reach steady-state. Steady-state solutions *might* provide some insight into bounds on small scale program execution though. Small scale programs are not as worthy (compared to large scale programs) of modeling in that they are not that expensive to run.

The first assumption (the approximation that transitions are taken randomly) is believed to apply reasonably well to programs which exhibit among other characteristics a recursive nature or independent element characteristics[6]. For example, where many elements of a list are to be processed and each element is relatively independent of others. A more concrete example will be seen later and was used as a test of the basic SMP model's accuracy.

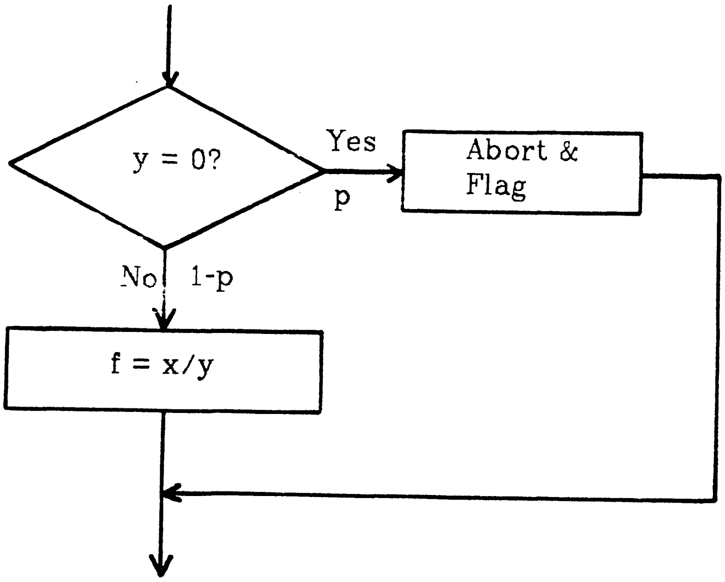
Computations that exhibit element independence, besides being candidates for stochastic modeling, can typically be partitioned to operate in the MIMD mode of operation. Examples of such computations include artificial intelligence state-space generation and searching, and database processing. In the context of concurrent program execution mode the operating system introduces stochastic (or seemingly random) characteristics into system behavior.

Remaining work to be done on the model includes testing the model on real programs that are believed to exhibit stochastic behavior, and examining the extent to which the first assumption holds. It is also (presently) believed that the model may provide relatively meaningful statistics on large scale, relatively loosely coupled (i.e., not synchronized) program executions of a non-random nature. It is believed that the MIMD mode of operation inherently "randomizes" program execution. This is another area requiring more experimentation.

The assumption of stochastic program behavior also acts as a general model of program behavior when input data is unknown. That is, when a program is designed, input data is often unspecified (i.e., the program maps input data into output data in specified ways but the program designer does not specify input data), in this case branching probabilities represent the fraction

[6] Models of hardware behavior have made a similar assumption regarding the nature of processor behavior [MuM82a], here a similar assumption is being made at a different level of system description.

of time arcs are taken when considering input values. For example, consider states which represent a simple division operation:



Then under general circumstances, p represents the fraction of time that $y = 0$ during a division, or it is the fraction of time that division by zero occurs when the routine is used.

Randomness, then, is believed to occur naturally in large scale independent element computations, and to some extent may be used to simplify state transition diagrams. For now consider there to be only *computation and reference states*. The following section describes the performance measures of interest in the parallel program execution mode of operation. They act as performance indicators of program design, GM data layout efficiency, hardware speed, etc.

3.2. Performance Measures.

This section describes performance measures of system behavior. The SMP model acts as a mapping from system configuration, program design, and GM data layout into a set of values which indicate (in an interpreted way) system performance.

3.2.1. Program Execution Time.

Important measures of program design and GM data layout are program execution times. There are basically two "types" of execution times: local execution time which is the time it takes a processor to compete its allotted task; and global execution time which is the time it takes the complete MIMD system to complete the full parallel program. Notice that often the global program execution time is the time required for *all* processors to complete their tasks; that is, the global task is complete only when all of its components are done. If local program execution times are known, then global execution time can be determined. These two *random variables* (rv's) will be termed

$$\tilde{E}_i = \textit{execution time of program } i, \quad 1 \le i \le P.$$

$$\tilde{E} = \textit{global execution time.}$$

Then,

$$\tilde{E} = \max_i \{ \tilde{E}_i \}$$

Using notation that will *always* be used, the ~ denotes a random variable. If $\tilde{A}$ is a random variable, then $A(t)$ is its cumulative probability function (CDF) and $a(t)$ is $\tilde{A}$'s probability density (or mass) function (pdf, or pmf) if it exists:

$$A(t) = Pr(\tilde{A} \le t)$$
$$a(t) = \frac{dA(t)}{dt}.$$

Often $a(t)$ will be a probability mass function where Dirac delta functions ($\delta(t)$) will be used to account for discrete components. Using this interpretation, the existence of $a(t)$ is not of practical concern.

The mean of a rv $\tilde{A}$ will be written $\bar{A}$, then

$$E[\tilde{A}] \equiv \bar{A} = \int_0^\infty t dA(t),$$

likewise, moments of $\tilde{A}$ will be written as:

$$E[\tilde{A}^k] = \overline{A^k} = \int_0^\infty t^k dA(t).$$

Denote the Laplace-Stieltjes transform (LST) of $A(t)$ as $A^*(s)$ and the LT of $a(t)$ as $a^*(s)$.

Returning to the discussion of execution time, a possible objective of program design and GM layout is to *minimize* a measure of the global execution time. A good candidate is to minimize $\bar{E}$.

Thus the objective of parallel program design and GM data layout could be to minimize $\bar{E}$. Other measures of global execution time could be used, but the mean is an obvious choice. It might, for example, be desirable to minimize the variance of global execution time in order to ensure "finite" and constant (as best as possible) program execution time. A discussion of the relationship between global and local execution times will be delayed until section 5.1

3.2.2. Processor Utilization.

A fundamental measure of system effectiveness is the fraction of time that processors are doing useful work. It is the fraction of time that processors are doing useful work in computation states (this is certainly related to the fraction of time that processors are idle). Define φ_p to be the fraction of time that processor/program p is "in" computation states.

Define system processor utilization φ to be the weighted sum of φ_p where the weighting coefficients represent relative importance of various programs:

$$\varphi = \sum_{p=1}^{P} w_p \varphi_p .$$

For example, if program j is very important relative to other programs, then one might set $w_j \gg w_i$, $i \neq j$ (assume $0 \leq w_p \leq 1, \sum_p w_p = 1$). This would be the case for example if processor j is the only general-purpose processor in the system and processors $i \neq j$ are I/O processors. Here obviously program j is most important in determining φ. It could be, though, that for I/O bound situations, weighting φ_j most heavily and optimizing φ actually makes the system run jobs more slowly.

Notice that φ_p is affected by many things including times spent in computation states, GM connection times, and times spent by requests in GM queues. The random component of these three (from the viewpoint of wasted time due to GM interference) is queueing time by requests. Time spent in queue is purely wasted time. Times spent in computation states and GM connection times are dependent only on program design, GM data layout, and the speed of system hardware.

A fundamental property of program design with its GM data layout is *potential processor utilization* $\hat{\varphi}$. This is the same as φ but is computed assuming zero queueing delay, i.e., with no memory interference. Notice that $\hat{\varphi}$ represents system processor utilization for a system employing (in some way) perfect processor synchronization. It is an intrinsic quantity that indicates maximum system utilization. For a given program and GM data layout $\varphi \leq \hat{\varphi}$. Similar to φ define $\hat{\varphi} = \sum_p w_p \hat{\varphi}_p$ simply as the weighted sum of individual potential processor utilizations.

Then to measure the amount of potential processing power lost to memory interference define ξ to be the *relative system processor utilization:*

$$\xi = \frac{\varphi}{\hat{\varphi}}.$$

Certainly $0 \leq \xi \leq 1$. Another objective of program design and GM data layout might be to maximize ξ (although it is not clear that this is as useful a minimizing $\bar{E}$).

3.2.3. Memory Utilization.

Memory utilization will be defined as:

$$\rho_m = \textit{fraction of time that GMM m is in use.}$$

Memory utilization is a seemly robust characteristic of system behavior in that relatively accurate predictions of memory utilizations may be obtained with simple approximations [MuM82a, Pat79, Str70].

3.2.4. Connection Bandwidth.

Connection bandwidth will be defined as the number of PE-GMM connections in use at time t. This definition actually defines connection bandwidth (or bandwidth for short) to be a time dependent stochastic process $\{\tilde{B}(t), t \geq 0\}$ where

$$\tilde{B}(t) = \begin{array}{l}\textit{number of PE-GMM connections}\\ \textit{in use at time } t\end{array}$$

$\tilde{B}(t)$ is a random variable because of the stochastic nature of program execution induced by those characteristics discussed in section 3. and 3.1.

An important characteristic of ICN's is their maximum attainable connection capacity. Define this to be $\hat{B}$ [MuM82a]. Then a measure of the fraction of ICN capacity in use at time t is

$$\tilde{\gamma}(t) = \frac{\tilde{B}(t)}{\hat{B}}.$$

$0 \leq \tilde{\gamma}(t) \leq 1$. If the system reaches steady-state (discussed later):

$$\lim_{t \to \infty} E[\tilde{\gamma}(t)] = \lim_{t \to \infty} \bar{\gamma}(t) \equiv \bar{\gamma}.$$

$\bar{\gamma}$ represents the steady-state mean fraction of ICN capacity that is used by the set of programs[7] running on the system.

The quantities ρ_m, ξ, $\bar{\gamma}$ characterize an operating point of the system. From these quantities, simple conclusions may be drawn regarding system bottlenecks, program/system improvements, etc. ([Kuc78] contains a similar discussion for single processor-memory-disk systems). For example, if ξ and ρ_m are small and $\bar{\gamma}$ is large then it may be concluded that the system is ICN bound.

The quantities $\hat{B}$ and $\bar{\gamma}$ depend on the type of ICN used for the PE-GM connection. Multistage networks [Sie80, WuF80, Law75] are an attempt at attaining crossbar performance while maintaining a lower rate of cost growth with size. Simple busses may be used if cost is a major concern.

A general SMP model can be used to determine the required capacity of ICN's considered for a particular system. The SMP model presented here assumes, as in the system description discussed in section 2, that the system is of small to moderate size and the ICN is a crossbar. Analysis of multistage networks is more complex but has been studied to some extent [MuM82b, MuM82c, Pat79, DiJ80].

When the ICN is circuit switched, more concise statements may be made concerning bandwidth [MuM82a]:

$$\hat{B} = \min\{P, M\}$$

$$\tilde{B}(t) = \textit{number of GMM's in use at } t.$$

[7] The term "set of programs" will be used to describe a parallel program and its GM data layout.

Then defining

$$\tilde{B}_m(t) = \begin{cases} 0 & \textit{if GMM m is not busy a time t} \\ 1 & \textit{if GMM m is busy at time t} \end{cases}$$

leads to

$$\tilde{B}(t) = \sum_{m=1}^{M} \tilde{B}_m(t), \qquad \overline{B}(t) = \sum_{m=1}^{M} \overline{B}_m(t).$$

More discussion on these measures will be presented later. Notice that for circuit switched ICN's $(\rho_m, \xi, \overline{\gamma})$ is equivalent to (ρ_m, ξ).

3.2.5. Queue Lengths.

GM queue lengths are very important in that they provide much information about improvements in GM data layout. If, for example, it is found that the average number of requests present in a single GMM queue is far greater than the average number in other queues, it would be advisable to redistribute GM data to even out mean queue lengths across all GM queues. A single large queue length indicates that a particular GMM is acting as a system bottleneck and it would be advantageous to attempt to relieve the congestion at a this queue.

Define

$$\tilde{N}_m(t) = \begin{matrix} \textit{the number of requests in GMM} \\ \textit{queue m at time t.} \end{matrix}$$

Only requests in the queue proper (not including the server) are included in this measurement.

The next section discusses parameters that the SMP model uses. The parameters serve to describe programs that execute on PE's. The parameters typically require extensive knowledge of program characteristics, actual knowledge of program behavior is probably not as extensive as the SMP model can use. The SMP model can be used as an approximate guide to program/system performance when only approximate program characteristics are used. The extent of applicability of the use of approximate program characteristics might be material for future study.

3.3. SMP Model Parameters.

This section describes the SMP model, its notation and some of its characteristics.

Define $\tilde{Z}_p(t)$ to be the state of processor[8] p at time t (state numbers are positive integers) then *the SMP model is described by saying that* $\{\tilde{Z}_p(t), t \geq 0\}$ *is a semi-Markov process.* This is an approximation but if the MIMD system reaches a steady-state, it is not an unreasonable one. Steady-state existence is a reasonable assumption for the large scale computations considered here.

The approximation that $\{\tilde{Z}_p(t), t \geq 0\}$ is a semi-Markov process amounts to making the following assumption regarding program execution:

Every time a state in the P/M transition diagram is entered (the corresponding Markov renewal process changes state), the distribution of time until the next state transition along with the next state entered is independent of all system information except the state just entered. More formally, define the following quantities:

$$\tilde{M}_p(n) = \begin{array}{l} \textit{state of processor p after} \\ \textit{the nth transition of the processor} \\ \textit{p embedded Markov chain (MC)}^{9} \end{array}$$

[8] Since we are considering parallel program execution the term processor will be used to describe both the processor and program, they are equivalent.

$$\tilde{S}_{ps} = \textit{sojourn time spent in state s by processor p.}$$

Then the approximation that processor p behaves as a SMP becomes that the following holds:

$$\begin{aligned} &Pr(\tilde{S}_{ps} \le t,\, \tilde{M}_p(n+1) = j \mid \textit{all previous program information}) \\ &\quad = Pr(\tilde{S}_{ps} \le t,\, \tilde{M}_p(n+1) = j \mid \tilde{M}_p(n) = s) \end{aligned}$$

Define the following:

$$Pr(\tilde{S}_{pi} \le t,\, \tilde{M}_p(n+1) = j \mid \tilde{M}_p(n) = i) \equiv Q_p(i,j,t)$$

$Q_p(t)$ is the semi-Markov kernel for program (SMP) p, $Q_p(t)$ is a matrix of CDF's.

If the transition probabilities of arcs in the P/M transition diagram are independent of sojourn times (that is, the probability of taking a particular next state arc is independent of the amount of time spent in the present state), then

$$\begin{aligned} Q_p(i,j,t) &= Pr(\tilde{S}_{pi} \le t \mid \tilde{M}_p(n) = i)\, Pr(\tilde{M}_p(n+1) = j \mid \tilde{M}_p(n) = i) \\ &= Pr(\tilde{S}_{pi} \le t \mid \tilde{M}_p(n) = i)\, P_p(i,j) \\ &= S_{pi}(t)\, P_p(i,j) \end{aligned}$$

Where P_p is the one step probability transition matrix for the embedded MC describing program p. $S_{pi}(t)$ is, by definition, the CDF of the sojourn time for program p in state i. The state sojourn time CDF's, $S_{pi}(t)$, are of two types:

(1) Computation state sojourn times where these CDF's are known, they are controlled by the programmer and the speed of processor/local memory hardware.

[9] The embedded MC is defined by the transition probabilities on the arcs of the P/M transition diagram.

(2) Reference state sojourn times which consist of two components.

Computation state sojourn times are known in that they may be determined directly from examining program machine code and PE hardware specifications. In their simplest form, they are the times between the completion of one GM connection, and the emission of the next request. Note that for simple constant computation time of length c units, $S_{pi}(t) = u(t-c)$. $u(t)$ is the unit step function and $\delta(t)$, the Dirac delta function, is defined as its derivative.

Reference state sojourn times consist (in their simplest form that applies to a crossbar ICN) of two components: the amount of time spent by a request in its GMM queue; and the amount of time that the program uses the GMM referenced. First consider that processor p references memory m when it enters state s (assume only one request is emitted when a particular state is entered) then

$$\tilde{S}_{ps} = \tilde{W}_{psm} + \tilde{Y}_{psm}. \tag{1}$$

Where $\tilde{W}_{psm}$ is the time spent in queue m by the request emitted by processor p when it entered state s. $\tilde{W}_{psm}$ is the waiting time rv seen by an arriving request from processor p in state s referencing memory m. In the parallel program execution mode we expect $W_{psm}(t)$ to be independent of s so we will write $\tilde{W}_{pm}$ for $\tilde{W}_{psm}$.

$\tilde{Y}_{psm}$ is the connection time required by processor p using memory m in state s. For simple unit connection times $Y_{psm}(t) = u(t-1)$. This CDF maps the length of pages transferred (in, for example, words) into a transfer (connection) time. This mapping depends basically on GMM RAM speed. This CDF can also be used to reflect secondary disk access time in the case where disk paging is modeled.

If the memory chosen upon entry to a reference state is random (as it might be, for example, in the interleaved mapping scheme), the pmf of sojourn time for a reference state becomes

$$s_{ps}(t) = \sum_{m=1}^{M} (w_{pm}(t) \; o \; y_{psm}(t)) \, \eta_{psm} \qquad (2)$$

$$s^{*}_{ps}(s) = \sum_{m=1}^{M} w^{*}_{pm}(s) \times y^{*}_{psm}(s) \times \eta_{psm}$$

Where o denotes convolution of the pmf's $w_{pm}(t)$ and $y_{psm}(t)$ ($\overline{W}_{pm}$ and $\overline{Y}_{psm}$ are independent[10]) and η_{psm} is the probability that processor p emits a request for memory m when it enters state s. If $\eta_{psm} = 1$ then this indicates an emission with certainty.

Mean reference state sojourn times become simple to evaluate:

$$\overline{S}_{ps} = \sum_{m=1}^{M} (\overline{W}_{pm} + \overline{Y}_{psm}) \, \eta_{psm} \qquad (3)$$

$\sum_m \eta_{psm} = 0$ for computation states while $\sum_m \eta_{psm} > 0$ for reference states. In particular $\sum_m \eta_{psm} = 0, 1$ for the single request per processor situation.

Concluding this section: the SMP model uses the quantities $S_{ps}(t)$, P_p, $Y_{psm}(t)$, and η_{psm} to describe program characteristics. The main point in the SMP model (and hence its name) is to take processors/programs to behave as stochastic FSM's that are describable as semi-Markov processes. The next section describes fundamental semi-Markov process relationships that will be used later.

[10] This is not actually true in that a large mean value for $\overline{Y}_{psm}$ may cause a large queue to form during this time (processor p connection time for memory m and state s) which in turn affects $\overline{W}_{pm}$, due to the indirect effect, they will often be taken to be independent.

4. Fundamental SMP Relationships.

The relationships described here are common knowledge and may be found in [Cin75, HeS82, Ros70]. Throughout let $\{\tilde{Z}_p(t),\ t \geq 0\}$ be the SMP under consideration.

Define the state space of $\tilde{Z}_p(t)$ to be A_p, A_p is the set of states in program p. Define

$$P_{ps}(t) = Pr(\tilde{Z}_p(t) = s).$$

Then $P_{ps}(t)$ characterizes the trajectory of program p. For deterministic programs (say a single processor, running a deterministic program) $P_{ps}(t)$ describes *exactly* where the program will be at time t (t = 0 marks the beginning of program execution). For a realistic program SMP, in general, $P_{ps}(t)$ is virtually impossible to find due to complexity [Cin75, HeS82, Ros70].

What is readily available about general SMP's is their steady-state distribution of state occupancy, i.e., $\lim_{t \to \infty} P_{ps}(t)$ is very easily obtained.

Define

$$C_p(i,j,t) = Pr(\textit{first entry to state } j \textit{ in time} \leq t \mid \textit{initially in state } i)$$

Then $C_p(i,j,t)$ is the CDF of the time required for program p to "cycle" from state i to state j, i.e., it is the CDF of the time between two entries to state j if $i = j$. This is certainly an indicator of the "rate" of program execution if state j is some given state. $\tilde{C}_p(i,j)$ is performance measure that arises naturally in the SMP model. Since the process is an SMP, each entry to state j constitutes a renewal process with renewal time CDF $C_p(j,j,t)$.

Then in the case of non-lattice[11] $C_p(j,j,t)$:

[11] A lattice random variable takes on values (with non-zero probability) that are integral multiples of the rv's period, i.e.: $\sum_{n=0}^{\infty} Pr(\tilde{V} = n\beta) = 1$.

$$\lim_{t \to \infty} P_{pj}(t) \equiv P_{pj} = \frac{\overline{S}_{pj}}{\overline{C}_p(j,j)}, \qquad if\ C_p(j,j) < \infty. \tag{4}$$

Since the model presented here assumes that the system reaches steady-state, P_{pj} will be used as the steady-state state occupation distribution. P_{pj} is also the (steady-state) probability that the SMP will be in state j at a *random* point in time. These will be termed the general-time probabilities [GrH74].

If the embedded MC is irreducible and positive recurrent then

$$P_{pj} = \frac{\pi_{pj} \overline{S}_{pj}}{\sum\limits_{k \varepsilon A_p} \pi_{pk} \overline{S}_{pk}} = \frac{\overline{S}_{pj}}{\overline{C}_p(j,j)} \tag{5}$$

Where π_{pj} are the stationary (steady-state) occupation probabilities for the embedded MC $\{\tilde{M}_p(n), n \geq 0\}$, π_{pj} are gotten from: $\pi_p = \pi_p P_p$, $\sum_k \pi_{pk} = 1$. [π_p is a row vector $(\pi_{p1}, \pi_{p2}, \ldots, \pi_{p|A_p|})$.] The next section describes the irreducible, positive recurrent, and regularity conditions in the SMP model.

From this simple formulation $\overline{C}_p(j,j)$ may be found from the embedded MC state occupation probabilities and *mean* sojourn times.

For the case when $C_p(j,j,t)$ is lattice, formulations are more complex, equation (4) does not hold and the left-hand side of equation (5) takes the form:

$$\lim_{n \to \infty} P_{pj|i}(t + n\beta) = \frac{\beta \pi_{pj}}{\sum\limits_{k \varepsilon A_p} \pi_{pk} \overline{S}_{pk}} \sum_{k \varepsilon S_t} 1 - S_{pj}(t - \beta_{ij} + k\beta)$$

Where,

β is the period of the SMP
β_{ij} is the first jump point of $C_p(i,j,t)$
$S_t = \{m : t + m\beta \geq \beta_{ij}\}$

and $P_{pj|i}$ is P_{pj} conditioned on the fact that i was the initial state of the SMP.

Analytic techniques for obtaining $C_p(i,j,t)$ exist and will be detailed next.

Define the Markov renewal kernel, $R_p(i,j,t)$:

$$R_p(i,j,t) = E\left[\begin{matrix} \textit{number of visits to } j \textit{ in } [0,\, t] \mid \\ \textit{process started in } i \textit{ at } t = 0 \end{matrix}\right]$$

$R_p(i,j,t)$ is related to $Q_p(i,j,t)$ and $C_p(i,j,t)$ as follows for finite state spaces:

$$R_p^*(s) = (I - Q_p^*(s))^{-1}$$

$$C_p^*(i,j,s) = \begin{cases} \dfrac{R_p^*(i,j,s)}{R_p^*(j,j,s)} & i \neq j \\ 1 - \dfrac{1}{R_p^*(j,j,s)} & i = j \end{cases}$$

In theory these quantities could be used to solve for CDF's of cycle times, etc. In reality, these computations are so complicated that it is doubtful that analytic results may be obtained for these CDF's. They might be obtained for special cases such as those SMP's used to describe approximate, low level processor models (see section 7).

Notice that to obtain such analytic results, CDF's of sojourn times are *required*, mean values will not suffice. As will be seen this requirement makes the calculation of the above CDF's even less attractive because CDF's of sojourn times (for reference states) are difficult to obtain.

5. Measure Derivation in the SMP Model.

This section describes the calculation of performance measures discussed in section 3 using the SMP relationships from section 4. First though the ideas of regularity, positive recurrence, irreducibility, and lattice distributions need to be addressed. Again define $\{Z_p(t), t \geq 0\}$ to be the SMP describing program p execution. A_p is the (finite) state space of program p.

The regularity condition required by many of the SMP results of section 4 is equivalent to the condition that the semi-Markov process can not make an infinite number of transitions in finite time with positive probability. This is certainly true in the context of an MIMD system, processors cannot make an infinite number of state changes in finite time, so the regularity condition is satisfied because we are dealing with a physical system.

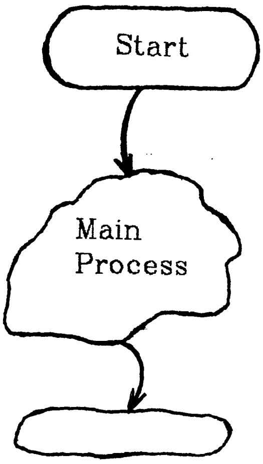
Notice that since most processors are synchronous machines, virtually *all* CDF's which are associated with time are *lattice* because processor computation times and connection times are based on a *clock at the lowest level of system operation*. GMM controllers are also based on clocks (although they would probably be different clocks than those that drive the processors). In fact, if PE's are indeed autonomous units, there may be many individual clocks in the system. Nevertheless, if the entire system is driven by a central clock of cycle time τ then we know $\beta \geq \tau$ and connection and computation times are lattice with period at least τ. Although this presents some difficulty[12] the results for absolutely continuous CDF's have been used in testing the SMP model and have been found to yield acceptable mean value results[13].

Due to the assumption that the system runs large scale programs, it is reasonable to presume that the program SMP's reach a steady-state. This assumes

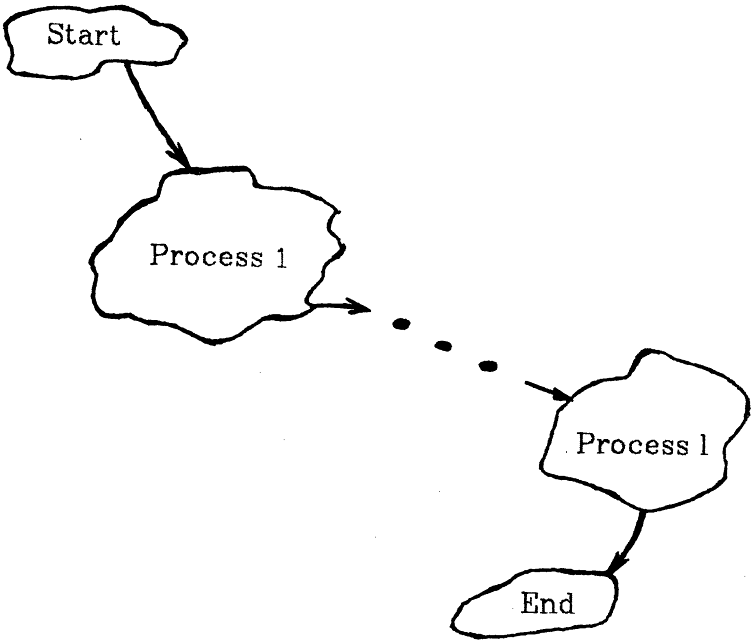
[12] This whole problem does not exist if processors are asynchronous machines.

[13] Further development of the SMP model might include examining the use of results for lattice CDF's where appropriate.

that the SMP's are irreducible. To satisfy this requirement, it seems reasonable to restrict A_p to an irreducible subspace. For example, consider a program p that looks like



If the main process is a large scale irreducible program, then it will be assumed that the SMP is restricted to this main irreducible process. The case of phases of irreducible processes (termed phased computations) will be studied in the future. An example of a phased computation might look like:



Hence it will be assumed that programs have been reduced to their irreducible subspaces by eliminating initial and final transient states (which contribute negligible or known amounts to measures of interest because the computations of interest are large scale problems).

Define A_p^r to be the state space of the restricted (or reduced) SMP describing program p execution. In general, an ^{r} will be appended to previously defined quantities to denote the corresponding quantity for the reduced SMP. The rest of this section discusses the derivation of performance measures.

5.1. Program Execution Time.

Let s_p be the starting state for program p and let f_p be the final (ending, or stopping) state for program p, then:

$$E_p(t) \geq C_p(s_p, f_p, t). \tag{7}$$

Where $C_p(s_p, f_p, t)$ is computed by assuming that all processors are busy (i.e., they are all in their irreducible sub-SMP's) during processor p's full program execution. $C_p(s_p, f_p, t)$ computed this way is lower than actual program execution time CDF's because other processors could reach their final states before processor p does. When other processors reach their final states, processors which are still running speed up (measured say as the inverse of the time between visits to a given state) because retarding effects from memory interference decrease. Hence $E_p(t)$ is greater than $C_p(s_p, f_p, t)$ computed assuming the full system is in operation during program p's execution and that the primary component of $\tilde{E}_p$ is due to occupation of the reduced SMP.

Next consider bounding $E(t)$. Since $\tilde{E} = \max_i \{\tilde{E}_i\}$,

$$\begin{aligned} E(t) &= Pr(\tilde{E} \leq t) = Pr(\max_i \{\tilde{E}_i\} \leq t) \\ &= Pr(\tilde{E}_1 \leq t, \tilde{E}_2 \leq t, \ldots, \tilde{E}_P \leq t) \end{aligned}$$

Which may be written as

$$\begin{aligned} E(t) = {} & Pr(\tilde{E}_1 \le t \mid \tilde{E}_2 \le t,\, \tilde{E}_3 \le t, \ldots, \tilde{E}_P \le t) \\ & \times Pr(\tilde{E}_2 \le t \mid \tilde{E}_3 \le t, \ldots, \tilde{E}_P \le t) \\ & \times Pr(\tilde{E}_3 \le t \mid \tilde{E}_4 \le t, \ldots, \tilde{E}_P \le t) \\ & \times \cdots \\ & \times Pr(\tilde{E}_{P-1} \le t \mid \tilde{E}_P \le t) \\ & \times Pr(\tilde{E}_P \le t). \end{aligned}$$

The basic idea is to replace each factor by a bounding quantity to find a simple bound on $E(t)$. By the same argument as above:

$$\begin{aligned} & Pr(\tilde{E}_{P-1} \le t \mid \tilde{E}_P \le t) \ \ge\ C_{P-1}(s_{P-1}, f_{P-1}, t) \\ & Pr(\tilde{E}_{P-2} \le t \mid \tilde{E}_{P-1} \le t,\, \tilde{E}_P \le t) \ \ge\ C_{P-2}(s_{P-2}, f_{P-2}, t) \\ & \cdots \end{aligned}$$

So

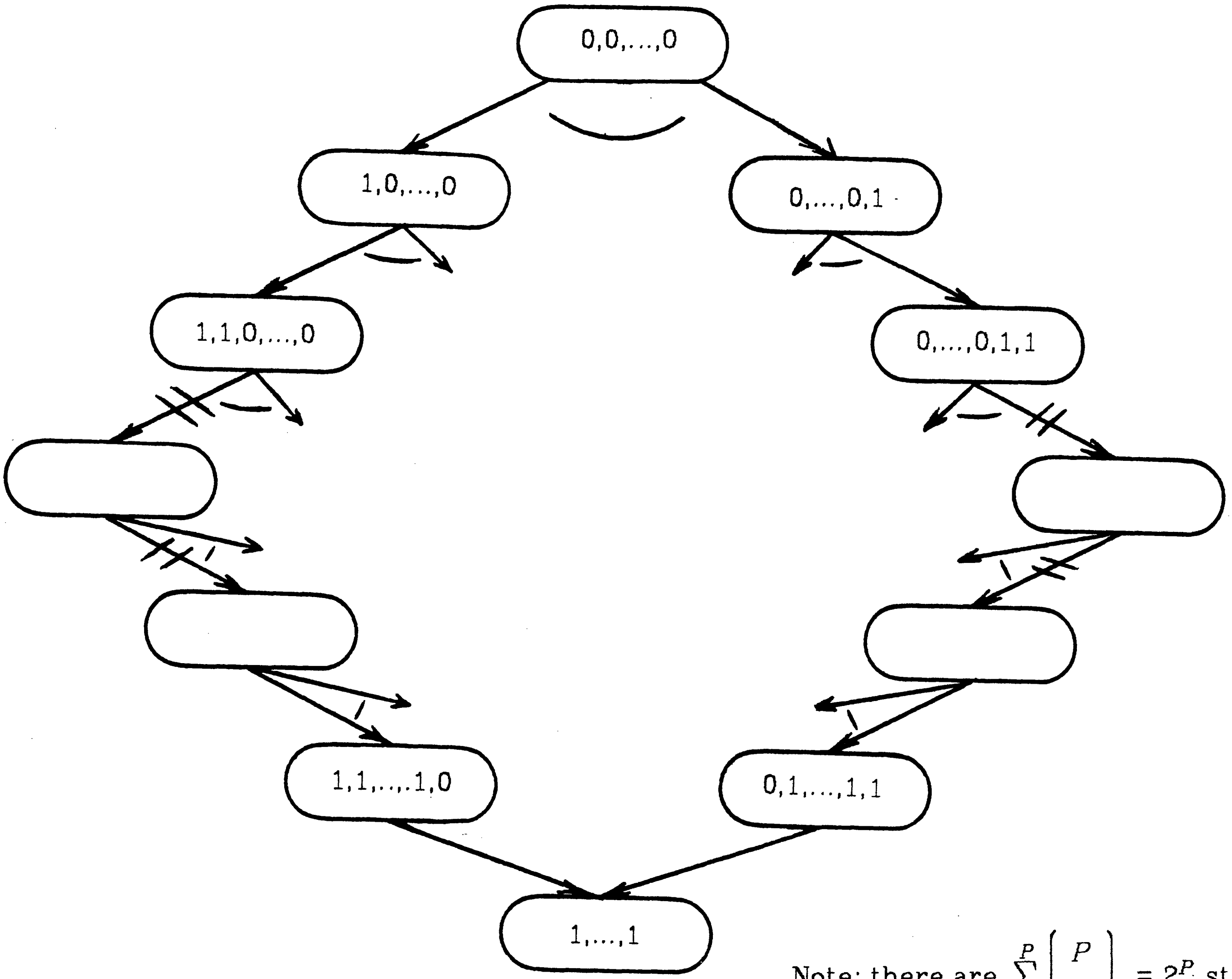
$$E(t) \ge \prod_{p=1}^{P} C_p(s_p, f_p, t) \tag{8}$$

provides a simple bound on $E(t)$.

A more accurate (though still approximate) approach would be to condition $E(t)$ on the sequence program terminations, then uncondition with the probability of the sequence. Consider a transition diagram (for an SMP) where each state is designated by a bit vector $(b_1, b_2, \ldots, b_P)$ where

$$b_i = \begin{cases} 0 & \textit{if program } i \textit{ is still running} \\ 1 & \textit{if program } i \textit{ has reached } f_i \end{cases}$$

A state diagram is shown in Figure 6. The problem with this approach is the complexity of the computations for $Q(i,j,t)$ in the SMP described by Figure 6. If $Q(i,j,t)$ are available, then $E(t) = C(\textit{top state}, \textit{bottom state}, t)$. Due to complexity this approach has not been pursued.



Note: there are $\sum_{p=0}^{P}\binom{P}{p} = 2^{P}$ states.

Figure 6. An SMP state diagram for execution time computation.

5.2. Processor Utilization.

Define

$$\eta'_{ps} = \sum_m \eta_{psm},$$

then

$$\varphi_p = \sum_{s \varepsilon A_p^r} P_{ps}^r \; 1_{\{\eta'_{ps}=0\}}, \tag{9}$$

where 1_E is the characteristic or indicator function:

$$1_E = \begin{cases} 0 & \text{if } E \text{ is not true} \\ 1 & \text{if } E \text{ is true} \end{cases}$$

Since the P_{ps}^r quantities are steady-state general-time pmf's, they represent fractions of time. Also since states are occupied disjointly, their probabilities add when evaluating the "or" condition.

Potential processor utilizations are computed similarly:

$$\hat{\varphi}_p = \sum_{s \varepsilon A_p^r} \hat{P}_{ps}^r \; 1_{\{\eta'_{ps}=0\}}. \tag{10}$$

5.3. Memory Utilization.

This is a more difficult entity to compute than is processor utilization basically because processor behavior is being modeled directly while memory utilization is a related quantity that is indirectly modeled. Due to this fact memory utilization may be approximated in at least two ways (a third will be seen later). In a circuit switched CN the following holds--

$$\rho_m = \lim_{t\to\infty} \frac{\int_0^t \tilde{B}_m(\tau)d\tau}{t} = \lim_{t\to\infty} Pr(\tilde{B}_m(t) = 1) = \lim_{t\to\infty} \overline{B}_m(t).$$

5.3.1. The Product Form Approximation (PFA).

This is based on viewing the memory module controller (server) at a random point in time (t) after the system reaches steady-state. Define a processor to be *using* GMM m if it has a request queued in queue m or it is connected to GMM m RAM.

$$\rho_m = Pr(GMM\ m\ is\ busy\ at\ t) = 1 - Pr(GMM\ m\ is\ idle\ at\ t). \qquad (11)$$

$$\begin{aligned} Pr(GMM\ m\ \ is\ idle\ at\ t) &= Pr(all\ processors\ are \\ &\qquad not\ using\ GMM\ m\ at\ t \\ &= Pr(processor\ 1\ is\ not\ using\ GMM\ m\ at\ t, \\ &\qquad and\ processor\ 2\ is\ not\ using\ GMM\ m\ at\ t, \\ &\qquad \ldots \\ &\qquad and\ processor\ P\ is\ not\ using\ GMM\ m\ at\ t) \\ &= Pr(processor\ 1\ is\ not\ using\ m\ at\ t\ |\ proc.s\ 2\ through\ P\ are\ not) \\ &\times Pr(processor\ 2\ is\ not\ using\ m\ at\ t\ |\ proc.s\ 3\ through\ P\ are\ not) \\ &\ldots \\ &\times Pr(processor\ P\text{-}1\ is\ not\ using\ m\ at\ t\ |\ proc.\ P\ is\ not) \\ &\times Pr(processor\ P\ is\ not\ using\ m\ at\ t) \end{aligned}$$

As will be often be done for simplicity the processor SMP's will be taken to be independent. This yields:

$$Pr(GMM\ m\ is\ idle\ at\ t) = \prod_{p=1}^{P} Pr(processor\ p\ is\ not\ using\ GMM\ m\ at\ t). \qquad (12)$$

Several examples of this approximation will be seen later. Obviously

$$Pr(processor\ p\ is\ not\ using\ GMM\ m\ at\ t) = \sum_{s \varepsilon A_p^r} (1 - \eta_{psm}) P_{ps}^r. \qquad (13)$$

A better way to compute memory utilization would be to use the conditioning formulation above and recognize that since the P SMP's are interlocked through memory interference, knowing (or conditioning on) where[14] one is influences the pmf of where others are. The study of this approach is planned. An example of the PFA will be seen in section 7.

5.3.2. The Sum Form Approximation (SFA).

This is based on an alternative (approximate) formulation of the probability that GMM m is busy at t.

$$\begin{aligned} \rho_m = Pr(GMM\ m\ is\ busy\ at\ t) = Pr(&processor\ 1\ is\ using\ GMM\ m\ at\ t, \\ &or\ processor\ 2\ is\ using\ GMM\ m\ at\ t, \\ &\ldots \\ &or\ processor\ P\ is\ using\ GMM\ m\ at\ t). \end{aligned} \tag{14}$$

Since the event that processor i uses GMM m at t is disjoint from the event that processor $j, j \neq i$ uses GMM m at time t, the "or" condition may be rewritten as:

$$Pr(GMM\ m\ is\ busy\ at\ t) = \sum_{p=1}^{P} Pr(processor\ p\ is\ using\ GMM\ m\ at\ t). \tag{15}$$

Notice that each reference state of a program SMP may be viewed as a succession of two states: one in which the request waits in queue; and one in which the processor *uses* the GMM RAM. Then if t is a random time it seems reasonable that

$$\begin{aligned} &Pr(processor\ p\ is\ using\ GMM\ m\ at\ t \mid processor\ p\ is\ in\ state\ s) \\ &= \eta_{psm} \frac{\overline{Y}_{psm}}{\overline{Y}_{psm} + \overline{W}_{pm}} \end{aligned} \tag{16}$$

[14] For parallel execution mode a single *token* may be used to describe where a processor's request is. That is, imagine a token owned by program p which flows through hardware facilities such as processors and GMM's. This would be the approach used if a pure network of queues model were developed. For example, we could define $D_p(t)$ to be the position of program p's token at time t. $D_p(t)=0$ iff the program p token is in processor p, $D_p(t)=k$ iff the program p token is in GMM k, $1 \leq k \leq P$. The problem here is to study the stochastic nature of $D_p(t)$.

Then unconditioning on the state of processor p at time t (again we actually know that the SMP's are not independent so the SFA is an approximation)

$$Pr(processor\ p\ is\ using\ GMM\ m\ at\ t) = \sum_{s \varepsilon A_p^r} \eta_{psm} \left(\frac{\bar{Y}_{psm}}{\bar{Y}_{psm} + \bar{W}_{pm}} \right) P_{ps}^r . \qquad (17)$$

Therefore,

$$\rho_m = \sum_{p=1}^{P} \sum_{s \varepsilon A_p^r} \eta_{psm} \left(\frac{\bar{Y}_{psm}}{\bar{Y}_{psm} + \bar{W}_{pm}} \right) P_{ps}^r . \qquad (18)$$

This is an approximation in the same sense that the PFA is; knowing that processor p is using GMM m, there is less uncertainty as to where processor $k \neq p$ is.

5.4. Connection Bandwidth.

$\tilde{B}(t)$ is very complicated unless it is studied in the steady-state, then we are primarily interested in $\lim_{t \to \infty} Pr(\tilde{B}(t) = k)$, $0 \leq k \leq \hat{B}$. Unfortunately these values are not readily available.

Returning to the original definition of $\tilde{B}(t)$ for a crossbar CN, $\tilde{B}(t) = \sum_{m=1}^{M} \tilde{B}_m(t)$. To find the mean steady-state value of $\tilde{B}(t)$ use

$$\lim_{t \to \infty} E[\tilde{B}(t)] = \lim_{t \to \infty} \sum_{m=1}^{M} E[\tilde{B}_m(t)] = \sum_{m=1}^{M} \lim_{t \to \infty} E[\tilde{B}_m(t)]$$

then

$$\lim_{t \to \infty} E[\tilde{B}(t)] = \sum_{m=1}^{M} \lim_{t \to \infty} Pr(\tilde{B}_m(t) = 1) = \sum_{m=1}^{M} \rho_m \qquad (19)$$

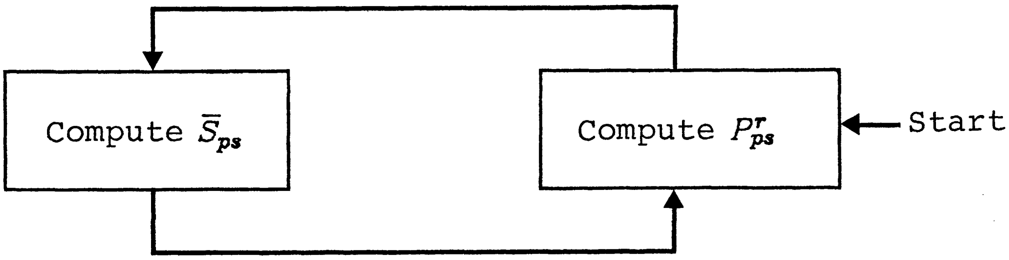
Which is the well known formula for $\lim_{t \to \infty} E[\tilde{B}(t)]$. By the bounded convergence theorem $\lim_{t \to \infty} E[\tilde{B}(t)] = E[\lim_{t \to \infty} \tilde{B}(t)]$ which would be the case if $\lim_{t \to \infty} Pr(\tilde{B}(t) = k) = Pr(\lim_{t \to \infty} \tilde{B}(t) = k)$ were available.

5.5. Queue Lengths.

Here the queue length seen by an observer is of interest, section 6 describes two computations of sojourn times, one of which computes this, the other which computes the queue length pmf for arriving requests; this is also a valuable measurement.

6. Computation of Sojourn Times.

Until now, it has been assumed that sojourn times have been available, equation (5) describes the relationship between sojourn times and P^r_{ps}, and hence the performance measures of the previous section. This section describes two approximate formulations of sojourn time computations. In general an appropriate solution algorithm for solving the non-linear equations which form the SMP model is to compute P^r_{ps}, $\bar{S}_{ps}$ as follows:



6.1. The Independent SMP Approximation for Sojourn Times.

As noted earlier, the mean value analysis requires the average time spent by a request from processor p in the GMM m queue (queue m for short). [Note that this is not simply related to $\tilde{N}_m(t)$ because here we are interested in the time spent by actual packets, not virtual packets as would be obtained with $\bar{N}_m$ [GrH74].] This formulation is based on a conditioning argument. Condition the waiting time CDF $W_{pm}(t)$ on the *sequence* of requests seen by the arriving request from processor p for queue m.

Let $\tilde{\Delta}_{mp}$ be the sequence of (processor, state) pairs seen by an arriving request from processor p referencing GMM m. Then $\tilde{\Delta}_{mp} = \delta = ((p_1,s_1), (p_2,s_2),..., (p_k,s_k))$ where (p_i,s_i) is a tuple representing the processor number queued in position i and the state s_i in which the processor

(p_i) is waiting. By convention p_1 is the processor using GMM m RAM at the arrival time and p_k is the processor whose request arrived just prior to processor p's.

Then conditioning and unconditioning on $\tilde{\Delta}_{mp}$ gives

$$W_{pm}(t) = \sum_{\delta} W_{pm}(t \,|\tilde{\Delta}_{mp} = \delta)\; Pr(\tilde{\Delta}_{mp} = \delta) \tag{20}$$

A characteristic of the steady-state assumption is that every request emitted from processor p for GMM m sees the same arrival point queueing time CDF's. Define $k = |\delta|$ = the length of δ (the number of requests in GMM m). Then

$$w^*_{pm}(s \,|\tilde{\Delta}_{mp} = \delta) = \begin{cases} 1 & |\delta| = 0 \\ e^*_{p_1s_1m}(s) & |\delta| = 1 \\ e^*_{p_1s_1m}(s) \prod_{j=2}^{|\delta|} y^*_{p_js_jm}(s) & |\delta| \geq 2 \end{cases} \tag{21}$$

Again denotes the LST of the marked CDF/pmf/pdf. Notice the new quantity whose LST is $e^*_{p_1s_1m}(s)$. $E_{p_1s_1m}(t)$ is the CDF of the *excess connection time* for the connection in progress at the time of processor p's request arrival at queue m. This connection is due to processor p_1 in state s_1 using memory m. See the Appendix for an explanation of the transform equation (21).

This excess connection time seems difficult to establish in that the stochastic process describing service (connection) times is not a renewal process and arrival times are not random (Poisson). A reasonable approximation might be to assume that arrival times *are* Poisson and that the excess connection time is that seen if service *is* a renewal process with renewal time CDF $Y_{p_1s_1m}(t)$; this is another effect that may warrant more study. Then [Kle75, Ros70, GrH74]

$$e_{p_1s_1m}(t) = \frac{1 - Y_{p_1s_1m}(t)}{\overline{Y}_{p_1s_1m}} \tag{22}$$

and

$$\bar{E}_{p_1 s_1 m} = \frac{\overline{Y^2}_{p_1 s_1 m}}{2\bar{Y}_{p_1 s_1 m}}. \qquad (23)$$

Using (20) the mean queueing time is easily obtained:

$$\begin{aligned} \bar{W}_{pm} &= \int_0^\infty t \, dW_{pm}(t) = \sum_{\delta} E[\tilde{W}_{pm} \mid \tilde{\Delta}_{mp} = \delta] \, Pr(\tilde{\Delta}_{mp} = \delta) \\ &= \sum_{\delta : |\delta| \geq 1} E[\tilde{W}_{pm} \mid \tilde{\Delta}_{mp} = \delta] \, Pr(\tilde{\Delta}_{mp} = \delta) \end{aligned} \qquad (24)$$

$$E[\tilde{W}_{pm} \mid \tilde{\Delta}_{mp} = \delta] = \bar{E}_{p_1 s_1 m} + \sum_{j=2}^{|\delta|} \bar{Y}_{p_j s_j m}, \qquad |\delta| \geq 1.$$

Where the null sum is taken to be zero.

Approximating the arrival point probabilities with the general-time probabilities gives an evaluation for $Pr(\tilde{\Delta}_{mp} = \delta)$, assuming again that the processor SMP's are independent.

$$Let \quad X = \{p_1, p_2, \ldots, p_k\}, \quad Y = \{1, 2, \ldots, P\}$$

Then

$$\begin{aligned} Pr(\tilde{\Delta}_{mp} = \delta) = {} & Pr(\textit{proc. } p_1 \textit{ is in state } s_1, \textit{ connection in progress}) \\ & \times Pr(\textit{proc. } p_2 \textit{ is in state } s_2, \textit{ in queue}) \\ & \times \cdots \\ & \times Pr(\textit{proc. } p_k \textit{ is in state } s_k, \textit{ in queue}) \\ & \times Pr(j \in Y - p - X \textit{ are not queued at GMM } m) \end{aligned} \qquad (25)$$

$$Pr(\textit{proc. } p_1 \textit{ is in state } s_1, .\textit{connection in progress}) = \eta_{p_1 s_1 m} P^r_{p_1 s_1} \left[\frac{\bar{Y}_{p_1 s_1 m}}{\bar{Y}_{p_1 s_1 m} + \bar{W}_{p_1 m}} \right] \qquad (26)$$

$$Pr(\textit{proc. } p_i \textit{ is in state } s_i, \textit{ in queue}) = \eta_{p_i s_i m} P^r_{p_i s_i} \left[\frac{\bar{W}_{p_i m}}{\bar{Y}_{p_i s_i m} + \bar{W}_{p_i m}} \right], \qquad 2 \leq i \leq k. \qquad (27)$$

$$Pr(j \; \varepsilon \; Y-p-X \textit{ are not queued at GMM } m) = \prod_{j \varepsilon Y-p-X} Pr(j \textit{ not queued at GMM } m) \tag{28}$$

Using equation (14) again:

$$Pr(j \textit{ not queued at GMM } m) = \sum_{s \varepsilon A_j^r} (1 - \eta_{jsm}) \, P_{js}^r \tag{29}$$

Although this formulation's usefulness (in a practical sense it is too complex to be used even for moderately sized systems and programs) is outweighed by its complexity it is useful for small state spaces (see section 7) and gaining insight into characteristics of memory interference.

This formulation for waiting time shows that mean sojourn times are dependent only on the first two moments[15] of connection times. Hence connection time CDF"s which display nearly the same first and second moments will yield nearly the same behavior (and hence P_{ps}^r). If connection times are constant, then $\overline{E}_{p_1 s_1 m}$ is minimized. Note that if mean queue lengths are small (<1) at arrival times, then even though $\overline{E}_{p_1 s_1 m}$ is the predominating contributor to waiting times, the waiting time is a small contributor to P_{ps}^r, if mean queue lengths at arrival times are large then $\overline{E}_{p_1 s_1 m}$ has only mild influence on waiting times and hence P_{ps}^r.

6.2. The Simple M/G/1 Computation of Sojourn Times.

This is an approach of a different type and appears to be more appropriate than the independent SMP approach of section 6.1. It provides good results for memory utilization, mean algorithm cycle times and processor utilization. Presently, it is not as accurate however at predicting mean waiting times.

If the number of processors is large and they behave independently and are also independent of the GMM system, it is reasonable to approximate the

[15] Within the framework of the approximation for $\overline{E}_{p_1 s_1 m}$ given above.

input process at GMM queues as Poisson processes. Technically this is not true unless all processor emission processes are Poisson, or there are an infinite number of them and each behaves as a renewal process with uniformly sparse emission rates. A survey of more technical aspects on the superposition of point processes may be found in [Cin72]. As a result of simulation studies and similar approximations [MuM82b] the assumption of approximate Poisson input processes seems reasonable.

Since processors emit requests and then wait for use of GM, there is a sort of "feedback" induced within the model formulation. This feedback serves to inhibit any Poisson nature of emission processes. Nevertheless, the *effective* rate of packet emission from processors will be used in the analysis.

The evaluation of sojourn times requires arrival point waiting times or the waiting time experienced by an arriving request, an approximation relating to queueing networks with a finite number of customers will be employed. [SeM81] presents a proof that in Markovian multiclass (of multichain) networks of queues, the distribution (in the steady-state) of queue length seen by an arriving customer is the same as the general-time distribution of the queue behavior without the customer in question.

The analysis here differs from these ideas in two major respects: the network of multiclass queues is certainly not Markovian; the arrival point queue length characteristics will be computed in a different manner[16].

The basic analysis is to use M/G/1 results to obtain queue statistics seen by processor p. Define the following quantities:

$$\lambda_{psm} = \begin{array}{l}\textit{effective rate of packet emission directed toward} \\ \textit{GMM m due to processor p in state s.}\end{array}$$

[16] To compute the arrival point distribution of queue characteristics, the system without processor p's contribution will be considered. A more exact result might be achieved if to find processor p's view of the system, a complete system solution without processor p is computed. This is another area for study.

$$\lambda_m^i = \begin{array}{l} \textit{rate of packet arrival at the} \\ \textit{input of the GMM m queue.} \end{array}$$

$$\lambda_{pm} = \begin{array}{l} \textit{total rate of packet flow} \\ \textit{from processor p to GMM m.} \end{array}$$

$$\lambda_m(p) = \begin{array}{r} \textit{rate of packet flow into GMM m} \\ \textit{not including processor p's contribution.} \end{array}$$

$$\lambda_p^o = \textit{ rate of packet output from proc. p.}$$

Notice that λ_m^i, λ_p^o, λ_{pm} are actually measurable quantities in the system, they are *viewable* at the bus level of the system. [As such, a test of the model on an actual system would consist of predicting these values and then measuring them as the system runs, such test may be conducted with a simulator.]

Obviously the following relationships hold among such data

$$\lambda_m^i = \sum_{p=1}^{P} \lambda_{pm}$$

$$\lambda_m^i = \lambda_m(p) + \lambda_{pm}$$

$$\lambda_m(p) = \sum_{j \neq p} \lambda_{jm}$$

$$\lambda_p^o = \sum_{m=1}^{M} \lambda_{pm}$$

$$\lambda_{pm} = \sum_{s \varepsilon A_p^r} \lambda_{psm}$$

Further define $\overline{X}_m(p)$ to be the mean connection time (computed over request types) for GMM m *excluding* processor p's contribution.

These quantities are related to the previous discussed quantities (in previous sections) as follows:

$$\lambda_{psm} = \frac{\eta_{psm}}{\bar{S}_{ps}} P^{r}_{ps}$$

Which may be seen as follows: let $r(\tilde{Z}_p(t))$ be the "reward" (emission) rate earned when processor p is in state $\tilde{Z}_p(t)$. Then

$$r(\tilde{Z}_p(t)) = \begin{cases} \dfrac{\eta_{psm}}{\bar{S}_{ps}} & \tilde{Z}_p(t) = s \\ 0 & otherwise \end{cases}$$

The time average rate of reward (emission) for processor p in state s referencing memory m becomes [Cin75, Ros70]:

$$\lambda_{psm} = \lim_{t \to \infty} \frac{\int\limits_0^{\infty} r(\tilde{Z}_p(t))\, dt}{t} = \sum_{k \varepsilon A^r_p} r(k) P^r_{pk} = \frac{\eta_{psm}}{\bar{S}_{ps}} P^r_{ps}.$$

Considering the GMM queues as M/G/1 queueing stations with arrival rate $\lambda_m(p)$ (that is, without processor p's contribution) yields the fact that the probability of a customer entering service (a request beginning a connection) is from processor $j \neq p$ in state s is $\lambda_{jsm} / \lambda_m(p)$. Connections beginning service form a Markov chain with these transition probabilities (when considering (processor, state) as the state variable). Computing the mean service time of the mth M/G/1 queueing station without processor p's contribution gives:

$$\overline{X^z}_m(p) = \sum_{j \neq p} \sum_{k \varepsilon A^r_j} \left[\frac{\lambda_{jkm}}{\lambda_m(p)} \right] \overline{Y^z}_{jkm} = \frac{1}{\lambda_m(p)} \sum_{j \neq p} \sum_{k \varepsilon A^r_j} \lambda_{jkm} \overline{Y^z}_{jkm}. \qquad (30)$$

Which may be seen graphically in Figure 7.

The simple M/G/1 approximation entails the following computations:

$$\rho_m(p) = \lambda_m(p)\, \bar{X}_m(p). \qquad (31)$$

Where $\rho_m(p)$ is the GMM m utilization "seen" by processor p.

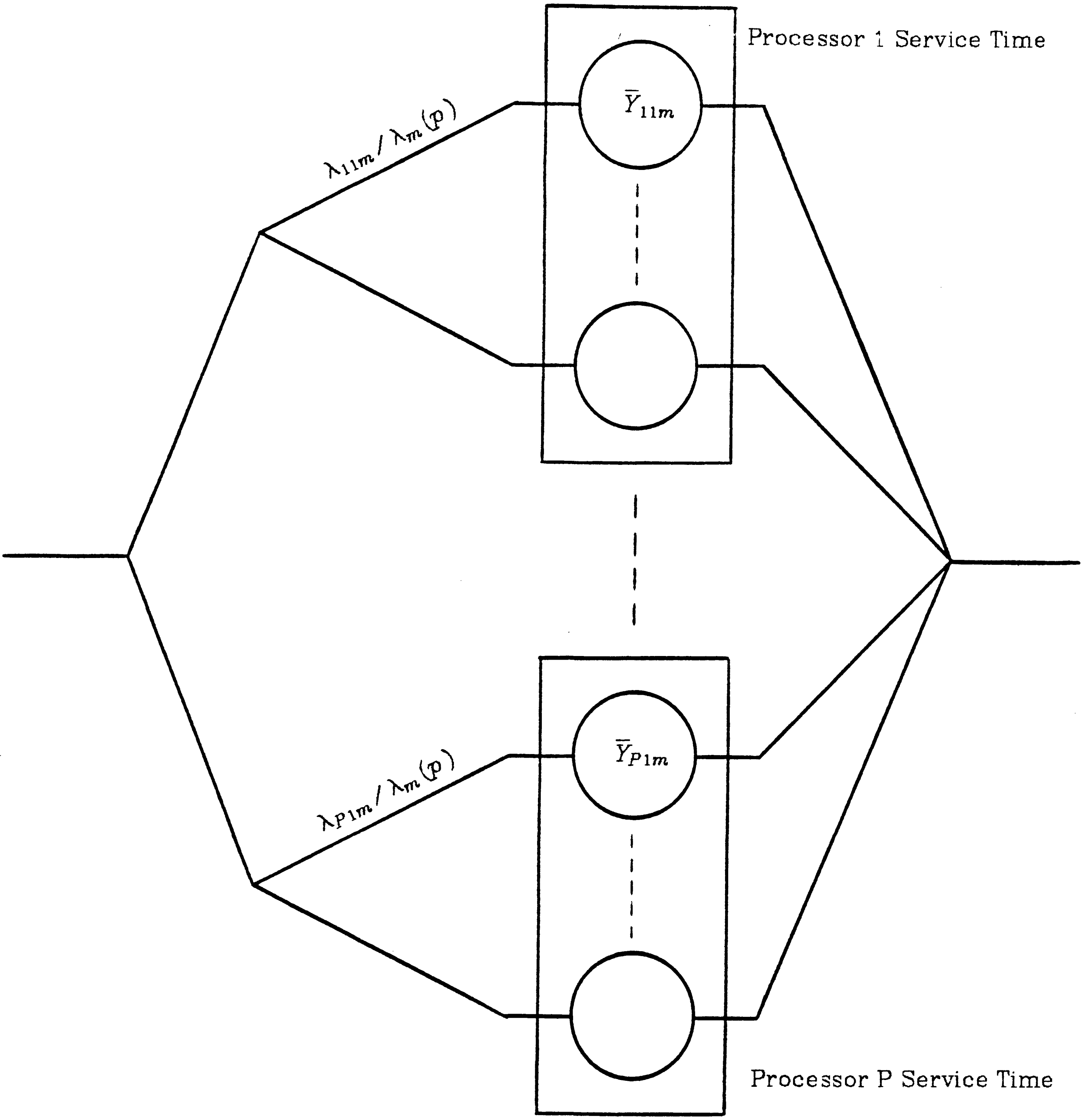


Figure 7. $\bar{X}_m(p)$ interpretation $(p \neq 1, P)$.

$$\overline{N}_{pm} = \frac{\lambda_m(p)\rho_m(p)}{1-\rho_m(p)}\left(\frac{\overline{X_m^2}(p)}{2\,\overline{X}_m(p)}\right) \tag{32}$$

Where $\overline{N}_{pm}$ is the number of requests seen in queue m by an arriving request from processor p. This is the appropriate P-K (Pollaczek-Khintchine) mean value formula, then using a modified Little's formula

$$\overline{W}_{pm} = \frac{\overline{N}_{pm}}{\lambda_m(p)} = \overline{N}_{pm}\overline{X}_m(p) + \rho_m(p)\left(\frac{\overline{X_m^2}(p)}{2\,\overline{X}_m(p)}\right) \tag{33}$$

Note that *again* only the first two moments of connection times arise in the formulation, this agrees with the independent SMP formulation with its given mean excess connection time approximation.

Note that it is plausible that $\overline{N}_{pm}$ from above may at first thought be > P-1 (which is certainly impossible in the system operation) but experience shows that the iterative solution technique[17] tends to seek a stable, feasible[18] solution to these M/G/1 equations.

The computation of memory utilizations is *particularly* simple using the queueing formulation.

$$\rho_m = \lambda_m^i\,\overline{X}_m \tag{34}$$

$$\overline{X}_m = \frac{1}{\lambda_m^i}\sum_{j=1}^{P}\sum_{k\varepsilon A_j^r}\lambda_{jkm}\,\overline{Y}_{jkm} \tag{35}$$

And the mean queue length is readily obtainable (this is the general-time outside observer's queue length, not to be confused with $\overline{N}_{pm}$):

[17] The iteration is to compute P_{ps}^r's, λ's, $\overline{S}_{ps}$'s, then P_{ps}^r's, etc.

[18] The uniqueness of the solution is yet unknown.

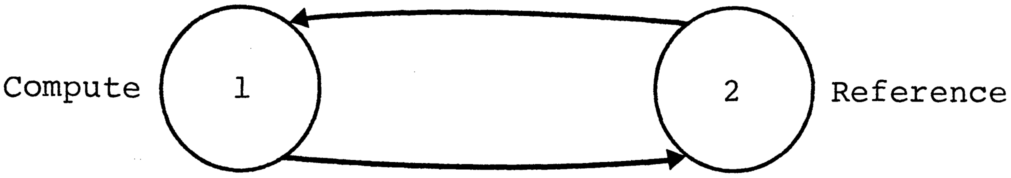
$$
\begin{aligned}
\bar{N}_m &= \lim_{t\to\infty} E[\tilde{N}_m(t)] = E[\lim_{t\to\infty}\tilde{N}_m(t)] \\
&= \frac{\lambda^i_m \rho_m}{1-\rho_m}\left(\frac{\overline{X^2}_m}{2\bar{X}_m}\right)
\end{aligned}
\qquad (36)
$$

7. Examples and Simulations.

7.1. Two State Model of Processor Behavior.

Past work in this area has concentrated on modeling processor behavior at a very low level. Past models [Hoo77, Pat79, MuM82a] have modeled processor behavior at the "clock cycle level", assuming that a processor is in a "think" state where it processes, or it is in a reference state. This is used to model hardware level behavior when the system is based on a single central clock and memory reference times are integral multiples of this clock.

In terms of the SMP model, the description of processor states becomes an alternating renewal process (ARP):



Typically computation times have been assumed to be geometrically distributed and based on the clock such that at each clock cycle, a processor (i) in its computation state emits a request with probability r_i. With this model of computation time, the mean sojourn time for state 1 is $1/r_i$.

To compare the SMP model with other models, assume that connection times are always one clock cycle in length, this is the typical assumption for idealized interleaved memory systems. [Bha75, SkA69] employed MC techniques to analyze this situation and found that for general r_i, η_{psm}, and geometric connection times, the MC state space becomes enormous even for small systems.

Hoogendoorn presents a model (termed the general memory interference, GMI model) which resembles the SMP model in that it too uses an iterative solution method for solving complicated model equations. The GMI model is believed by the author to be the most accurate of the published models of the ARP processor description and so will be used for comparison purposes.

When considering only two states, special case closed form solutions may be obtained very easily. Consider the special case where

$$\eta_{p2m} = \frac{1}{M} \qquad \textit{for all } p\,,\, m.$$

This represents the interleaved model where all GMM's are selected equiprobably. Let all processor computation state sojourn times be geometrically distributed, based on a unit time cycle (which is SMP p's period), with mean sojourn time $1/r$. That is

$$Pr(\tilde{S}_{p1} = n) = r(1 - r)^{n-1}.$$

Let all connection times have unit length (this could represent a memory cycle time) with certainty:

$$Y_{p2m}(t) = u(t - 1).$$

Due to the symmetry present in this situation all processor requests see the same waiting time CDF, all processor utilizations are the same, etc. Therefore subscripts will be dropped where they are not required.

From (23) we get

$$\overline{E}_{p2m} = \frac{1}{2}$$

$$\overline{Y}_{p2m} = 1$$

so $\overline{W} = \sum_{k=1}^{P-1}(k - \frac{1}{2})Pr(|\tilde{\Delta}| = k)$ from simplification of the waiting time formula.

Simplifying:

$$P_2 \equiv P_{p2}, \quad P_1 \equiv 1 - P_2$$

and

$$Pr(|\tilde{\Delta}| = k) = k! \begin{pmatrix} P-1 \\ k \end{pmatrix} \frac{P_2}{M}\left(\frac{1}{1+\overline{W}}\right)\left(\frac{P_2}{M}\left(\frac{\overline{W}}{1+\overline{W}}\right)\right)^{k-1}\left(\frac{M-1}{M}P_2 + 1 - P_2\right]^{P-k-1}$$

For $1 \leq k \leq P-1$.

Using thė fact that $P_2 = \dfrac{\overline{W}+1}{\overline{W}+1+\frac{1}{r}}$ gives

$$\overline{W} = \frac{(P-1)!}{M\left(\overline{W}+1+\frac{1}{r}\right)} \sum_{k=1}^{P-1} \frac{k-\frac{1}{2}}{(P-1-k)!}\left(\frac{\overline{W}}{M\left(\overline{W}+1+\frac{1}{r}\right)}\right)^{k-1}\left(1 - \frac{\overline{W}+1}{M\left(\overline{W}+1+\frac{1}{r}\right)}\right)^{P-(k+1)}$$

$$\equiv f(\overline{W})$$

Which suggests a simple solution technique that works for systems where $P \gg M$: $\overline{W}_{n+1} = f(\overline{W}_n)$. For systems with two processors a simple solution exists:

$$\overline{W} = \frac{\left\{\left(\frac{r+1}{r}\right)^2 + \frac{2}{M}\right\}^{1/2} - \frac{r+1}{r}}{2}$$

The rest of the performance measures may be derived from the calculation of $\overline{W}$ because $\overline{W}$ determines P_1 and P_2 which in turn complete the determination of the remaining measures.

Consider the use of the simple M/G/1 analysis. From the P-K formula (32):

$$\overline{W} = \frac{\rho_m(p)}{2(1-\rho_m(p))}$$

and

$$\rho_m(p) = \lambda_m(p) \times 1$$

$$\lambda_m(p) = \frac{1}{M} \times \left[\frac{1}{\overline{W} + 1 + \frac{1}{r}}\right] \times (P-1)$$

Then simplifying

$$\overline{W} = \frac{P-1}{2M\left(\overline{W} + 1 + \frac{1}{r}\right) + 2(1-P)}$$

Which has a unique solution:

$$\overline{W} = \frac{\left\{\left[M\left(\frac{r+1}{r}\right) + 1 - P\right]^2 + 2M(P-1)\right\}^{1/2} - \left[M\left(\frac{r+1}{r}\right) + 1 - P\right]}{2M}$$

Computing memory utilization using (34):

$$\rho_m = \lambda_m \times 1 = \frac{P}{M\left(\overline{W} + 1 + \frac{1}{r}\right)}$$

So

$$\overline{B} = \frac{P}{\overline{W} + 1 + \frac{1}{r}}$$

$$\varphi = \frac{\frac{1}{r}}{\overline{W} + 1 + \frac{1}{r}} = \frac{1}{1 + r(\overline{W} + 1)} \qquad \hat{\varphi} = \frac{\frac{1}{r}}{\frac{1}{r} + 1} = \frac{1}{1+r}$$

And

$$\xi = \frac{1 + r}{1 + r(\overline{W} + 1)}$$

Reviewing the equation relating $\overline{B}$ and $\overline{W}$, the sensitivity of $\overline{B}$ with respect to $\overline{W}$ may be determined. $\overline{B}$ is relatively insensitive to changes in $\overline{W}$ which means that significant error in $\overline{W}$ can yield a reasonable prediction for $\overline{B}$. $\overline{B}$ is a robust measure (perhaps this is why it is the measure that has been predicted most often in the past). The problem with the exclusive prediction of $\overline{B}$ is· that it does not yield program execution characteristics such as $\overline{C}_p(j,j)$. Assuming the two state model of computation in general causes program characteristics to be undetermined.

To understand the sensitivity of $\overline{B}$ with respect to $\overline{W}$, consider $\overline{W}$ as a function of $\overline{B}$ ($\overline{W}$ should really be written as a function of r):

$$\overline{W} = \frac{P}{\overline{B}} - \frac{r+1}{r}$$

Which is plotted in Figure 8. Also consider $|d\overline{W}/d\overline{B}|$ plotted in figure 9.

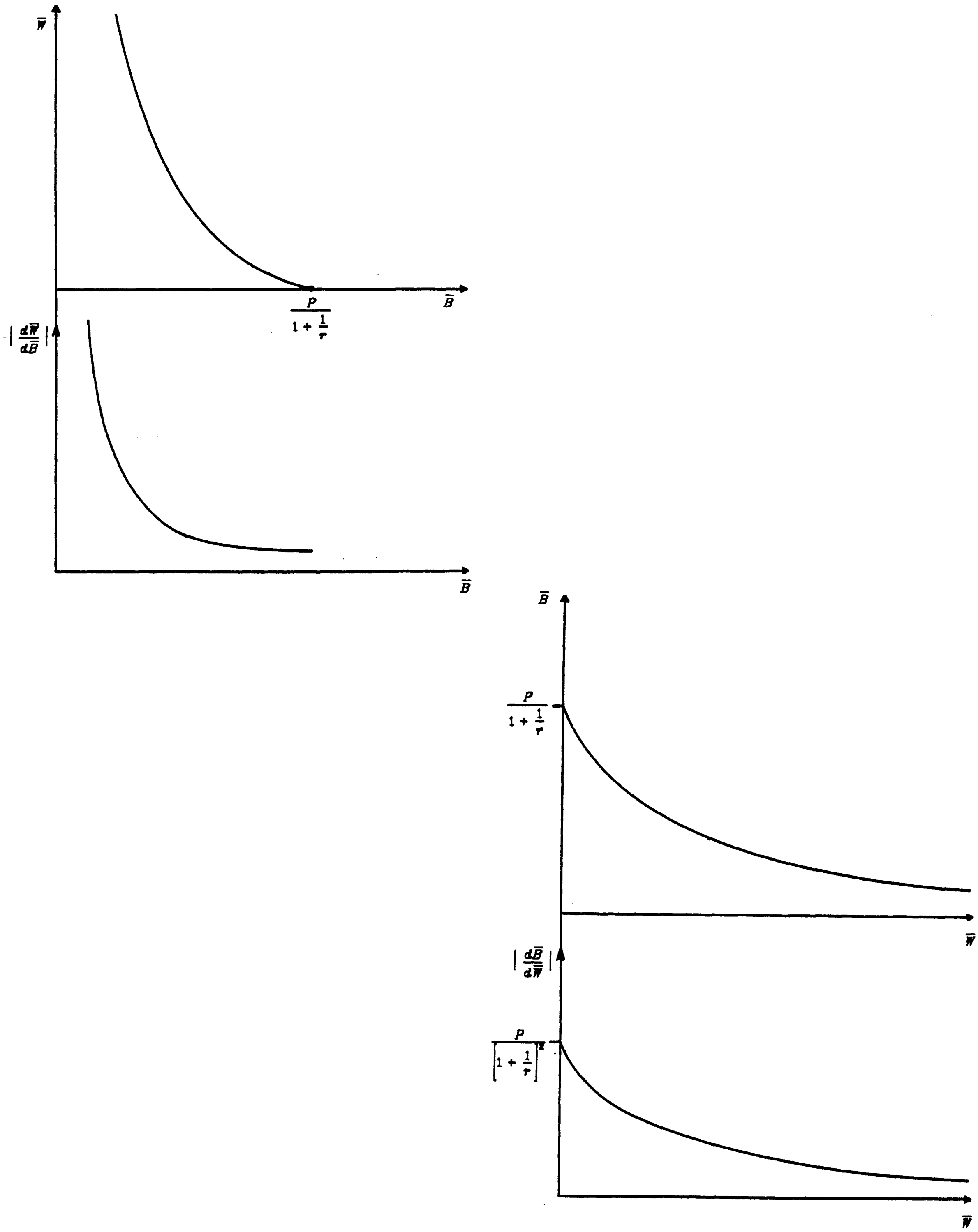
$$\frac{d\overline{W}}{d\overline{B}} = \frac{P}{\overline{B}^2}$$

Whereas

$$\frac{d\overline{B}}{d\overline{W}} = \frac{P}{\left(\overline{W} + 1 + \frac{1}{r}\right)^2}$$

When $\overline{W}$ is small with respect to $1 + \frac{1}{r}$, $\overline{B}$ remains relatively insensitive to changes in $\overline{W}$. $\overline{B}$ as a function of $\overline{W}$ and $|d\overline{B}/d\overline{W}|$ are plotted in Figure 9. As $\overline{W} \to \infty$, $|d\overline{B}/d\overline{W}| \to 0$ which again shows insensitivity to $\overline{W}$.

Notice that using the simple M/G/1 approximation enables simple closed form results to be obtained for general P and M.



Figures 8 and 9. Sensitivity analysis.

Results of using the two state model are tabulated in Tables I and II. Table I shows results for a 2x2 system. The simulator was written to simulate the ARP state diagram exactly. Notice that for $r = 1$, state 1 is occupied for precisely one time cycle and GM is used for 1 time cycle, so after an initial conflict (i.e., both processors emit requests for the same memory on the same cycle, one is delayed and one proceeds) both processors access GM *without* conflict. This phenomenon, termed self synchronization, is not modeled in the SMP (or any known general request rate model). This phenomenon is not expected to occur in reasonably general situations, this seems to be a special case. Note: all simulation values unless explicitly stated otherwise are single simulation values. Confidence intervals have not been compiled, nor have multiple run values been averaged. Simulation values for Table I should be regarded as rough values.

Table II shows results of the SMP model along with Hoogendoorn's GMI model (which only applies to the two state state diagram) and simulation confidence intervals from [Hoo77]. Here $r = 1$ or $S_{p1}(t) = u(t-1)$ along with $Y_{p2m}(t) = u(t-1)$. Note that the SFA and PFA calculations for $\overline{B}$ might be averaged to obtain a value in the confidence interval. The basic SMP model compares well with the GMI model. Notice that as the system gets larger, $\overline{B}$ computed using the simple M/G/1 approach gets better as would be expected because as P, M $\to \infty$, queue input processes approach Poisson processes (ideally) [Cin72].

7.2. The Instruction Storage Example.

Returning to the example of a system where instructions are stored in GM verses the use of LM for instruction storage. Make the following assumptions regarding instruction execution by processors[19]:

[19] The purpose here is to compare two situations and demonstrate a use of the SMP model, not to expand on instruction execution.

Analytic and simulation results for P = M = 2,
uniform (interleaved) reference patterns

r	$\overline{W}_{sim}$	$\overline{W}^1_{calc}$	$\overline{W}^2_{calc}$	$\overline{B}_{sim}$	$\overline{B}^3_{calc}$	$\overline{B}^4_{calc}$	$\overline{B}^5_{calc}$	ξ^6
0.05	.008	.012	.012	.094	.095	.095	.095	.999
0.10	.021	.023	.024	.180	.181	.181	.181	.998
0.20	.044	.041	.045	.330	.330	.331	.331	.993
0.30	.058	.057	.064	.457	.453	.456	.455	.985
0.40	.074	.070	.081	.559	.555	.560	.558	.977
0.50	.082	.081	.096	.645	.640	.649	.646	.969
0.60	.081	.091	.110	.724	.713	.725	.720	.960
0.70	.090	.099	.122	.792	.775	.791	.784	.952
0.80	.078	.106	.133	.854	.829	.849	.839	.944
0.90	.057	.112	.143	.922	.875	.899	.887	.937
1.00	.000	.118	.151	1.000	.916	.944	.930	.930

$\overline{W}^1_{calc}$ = mean queueing time for the independent SMP approximation.

$\overline{W}^2_{calc}$ = mean queueing time for the simple M/G/1 queueing approximation.

$\overline{B}^3_{calc}$ = mean memory bandwidth computed using the product-form approximation and $\overline{W}^1_{calc}$.

$\overline{B}^4_{calc}$ = mean memory bandwidth computed using the sum-form approximation and $\overline{W}^1_{calc}$.

$\overline{B}^5_{calc}$ = mean memory bandwidth computed using the simple M/G/1 queueing approximation.

ξ^6 = relative processor utilization based on $\overline{W}^2_{calc}$.

Table I. 2x2 simulation results.

Analytic and simulation results for unit processing and connection times with uniform reference patterns

P	M	$\overline{B}_{sim}$ 90% CI		GMI $\overline{B}$	$\overline{B}^1_{calc}$	$\overline{B}^2_{calc}$	$\overline{W}^3_{calc}$	$\overline{B}^4_{calc}$	$\overline{W}^5_{calc}$	ξ^6
4	2	1.5647	1.5843	1.5400	1.5226	1.5917	.513	1.5087	.651	.75
4	4	1.8194	1.8276	1.7917	1.7963	1.7846	.241	1.7778	.250	.89
4	8	1.9126	1.9244	1.9012	1.9058	1.8956	.110	1.8974	.108	.95
8	4	2.8460	2.9206	2.9789	2.9935	2.9306	.730	2.8542	.812	.71
8	8	3.4858	3.5346	3.5221	3.5754	3.4340	.330	3.4695	.306	.87
16	16	6.8140	6.9788	6.9844	7.1452	6.7068	.386	6.8513	.335	.86

$\overline{B}^1_{calc}$ = mean memory bandwidth computed using the product-form approximation and $\overline{W}^3_{calc}$.

$\overline{B}^2_{calc}$ = mean memory bandwidth computed using the sum-form approximation and $\overline{W}^3_{calc}$.

$\overline{W}^3_{calc}$ = mean queueing time for the independent SMP approximation.

$\overline{B}^4_{calc}$ = mean memory bandwidth computed using $\overline{W}^5_{calc}$.

$\overline{W}^5_{calc}$ = mean queueing time computed using the simple M/G/1 approximation.

ξ^6 = relative processor utilization based on $\overline{W}^5_{calc}$.

Table II. Simulation results with varying P and M.

(1) All processors are executing a similar instruction stream.

(2) GM is interleaved so connection times are unit length and references are directed to all GMM's equiprobably.

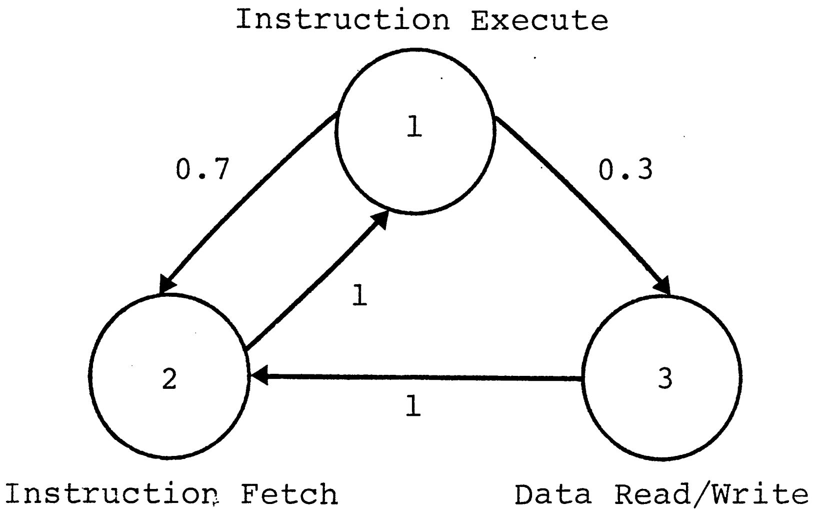
(3) About 30% of the instructions reference data (not instructions).

(4) Each non-data reference instruction requires 2 time units to execute (one instruction fetch and one instruction execution). Note that if the instruction is in GM, the a queueing delay is also present in each instruction

fetch/ execute cycle.

(5) Each GM data reference instruction requires one instruction fetch, one instruction execution time unit, and one GM connection cycle.

Consider first the case where all instructions and data are stored in GM. A state diagram is intuitively drawn as



Where

$$Y_{p2m}(t) = u(t-1), \quad \eta_{p2m} = \frac{1}{M}$$
$$Y_{p3m}(t) = u(t-1), \quad \eta_{p3m} = \frac{1}{M}$$
$$S_{p1}(t) = u(t-1)$$

An execution rate of interest is the inverse of the mean time between entries to state 1. Every entry to state 1 marks the beginning of an instruction execution. The rate of system instruction execution is:

$$\frac{P}{\bar{C}_1(1,1)}.$$

For the second situation where instructions are stored in LM, the state diagram is the same but state 2 is now a *computation* state (relative to the P/M level of interaction) so

$$S_{p2}(t) = u(t-1).$$

Table III shows the results of solving these systems descriptions using the basic SMP model with the simple M/G/1 approximation. From the table it may be seen that for a 16x16 system a speedup of 20% is achieved by storing instructions in LM (this speedup is relative to speed for a system with instructions stored in GM). Note that this result is actually conservative in that a real system would have faster (i.e., smaller access time) LM (similar to a cache) than GM so instruction storage in LM would result in better than a 20% speedup.

7.3. State Space Generator Example.

Consider the generation of a state space using a heuristic function applied to generate successor nodes [Nil71]. That is, a tree is to be generated

Instruction Storage in LM vs. GM							
P = M	LM Inst. Storage			GM Inst. Storage			% speedup
	Exec rate	$\overline{B}$	P_{p1}	Exec rate	$\overline{B}$	P_{p1}	offered by LM
2	.866	.260	.433	.792	1.030	.396	9.6
4	1.727	.518	.432	1.496	1.945	.374	15.4
8	3.450	1.035	.431	2.902	3.773	.363	18.9
16	6.894	2.068	.431	5.712	7.427	.357	20.7

Table III. Speedup with instruction storage in local memory.

with a heuristic function that is applied to all unexpanded nodes to generate successor nodes.

The implementation is based on [Nil71] but in a parallel environment where a list of nodes is kept in GM and each processor removes the next node to be expanded from an OPEN list, expands it using the heuristic function, writes the expanded node onto a CLOSED list, and writes the successors onto the OPEN list. Note that if the OPEN list becomes empty when, for example, P-1 processors operate on nodes which have no successors, P-1 processors must wait for the next node(s) to be written to the OPEN list by the single running processor.

In order that the lists OPEN and CLOSED be manipulated in a parallel environment, pointers will be kept GMM 1. Notice that GMM 1 should exhibit a higher rate of use than other GMM's. List elements (which may consist of several words) will be mapped into a single GMM. That is, each element of each list lies entirely within one GMM.

When a processor generates successors of a given node, it first modifies pointers kept in GMM 1 and the writes all the generated nodes into a single GMM indicated by a pointer in GMM 1. Pointers are also be needed in each GMM that indicate the exact position within the GMM that where new nodes should be added. Figure 10 shows a state of the OPEN list. The CLOSED list is a simple list that wraps around GMM modules. Due to these two specifications it seems that when node(s) are written onto the OPEN and CLOSED lists, the GMM chosen is chosen seemingly randomly, again provided a steady-state is reached.

A flowchart of the algorithm which each processor executes independently, is shown in Figure 11. First assume that OPEN is not empty, then the algorithm operates as follows: a processor removes the next node on OPEN, it

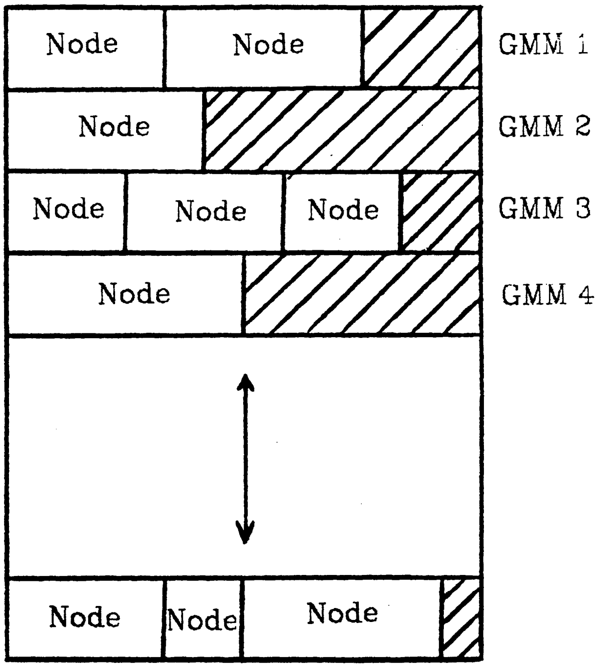


Figure 10. The OPEN list arrangement.

is indicated by a pointer in GMM 1 that points to the GMM that contains the next node on OPEN. The processor next obtains a connection to the GMM pointed to (the processor might modify the OPEN pointer in GMM 1, not all operations done on pointers are explained here; this example program was constructed to test the basic SMP model, not necessarily to create a finished parallel state space generator) and reads a node pointed to by a local pointer contained in the GMM, in this way circuit switched CN's provide a simple lock of pointers and data. Since nodes are stored uniformly in the steady-state, the GMM chosen for the read should be nearly random with uniform distribution.

Once a new node is read from the top of OPEN it is placed on the bottom of CLOSED. This is done similarly to the reading of elements from the OPEN list. Once the next node has been obtained (and stored in LM) and rewritten to

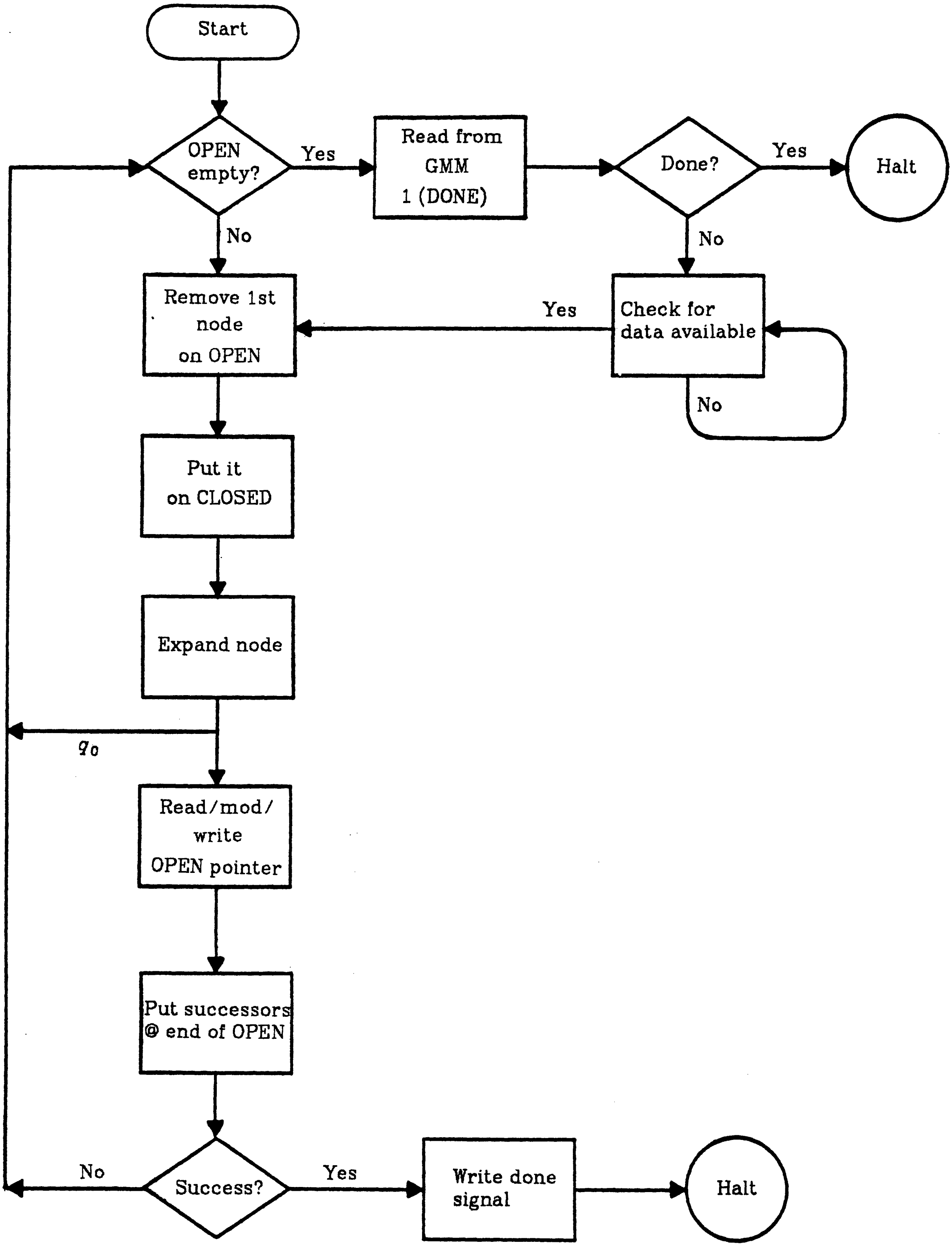


Figure 11. The state space generator example flowchart.

the CLOSED list, it is expanded. Each node is expanded using the same heuristic function (hence this fits the independent element list criterion) which has probability mass function q_k the k successors are generated, $0 \leq k \leq K$. Hence the arc labeled with q_0 leading from the "Expand node" state to the top of the cycle again.

After generating successors, the processor updates th OPEN pointer in GMM 1 (again, the semantics of the pointers are not discussed here), puts the new set of nodes at the end of OPEN and then continues.

Figure 12 shows the P/M level of description for this hypothetical state space generator. The following values have been assumed[20]:

4 processors and 4 GMM's

$$\begin{aligned}
&Y_{p11}(t) = u(t-100), \quad \eta_{p11} = 1 \\
&S_{p2}(t) = u(t-100) \\
&Y_{p31}(t) = u(t-100), \quad \eta_{p31} = 1 \\
&S_{p4}(t) = u(t-100) \\
&Y_{p51}(t) = u(t-100), \quad \eta_{p51} = 1 \\
&S_{p6}(t) = u(t-100) \\
&Y_{p7m}(t) = u(t-2000), \quad \eta_{p7m} = \frac{1}{M}, \quad m = 1,\ldots,M \\
&Y_{p8m}(t) = u(t-2000), \quad \eta_{p8m} = \frac{1}{M}, \quad m = 1,\ldots,M
\end{aligned}$$

$$\begin{aligned}
S_{p9}(t) = &\ .51\, u(t-5000) \\
&+ .25\, u(t-10000) \\
&+ .10\, u(t-15000) \\
&+ .05\, u(t-20000) \\
&+ .03\, u(t-25000) \\
&+ .03\, u(t-30000) \\
&+ .03\, u(t-35000)
\end{aligned}$$

[20] For the first simulation. Five different runs were performed with different parameters varied from run to run. Each run requires about 20 CPU seconds on the Michigan Terminal System, hence confidence intervals have not been accumulated.

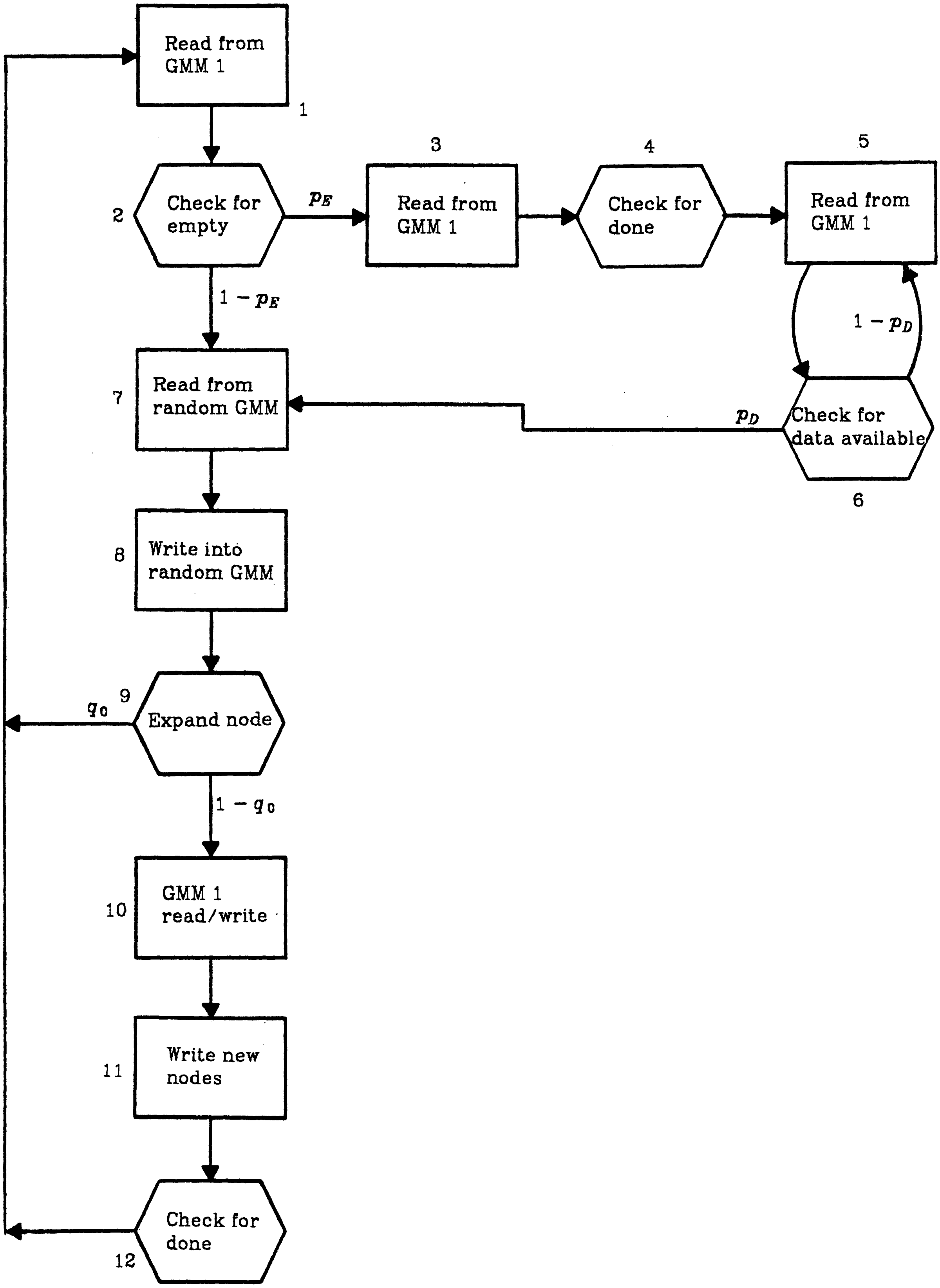


Figure 12. The P/M level description of the state space generator.

$Y_{p,10,1}(t) = u(t-200), \quad \eta_{p,10,1} = 1$

$$\begin{aligned} Y_{p11m}(t) = {} & .5051\, u(t-2000) \qquad \eta_{p11m} = \frac{1}{M}, \quad m = 1,\ldots,M \\ & + .2525\, u(t-4000) \\ & + .1010\, u(t-6000) \\ & + .0505\, u(t-8000) \\ & + .0303\, u(t-10000) \\ & + .0303\, u(t-12000) \\ & + .0303\, u(t-14000) \end{aligned}$$

$$q_0 = .01, \quad p_E = .01, \quad p_D = .05$$

$$S_{p12}(t) = u(t-500)$$

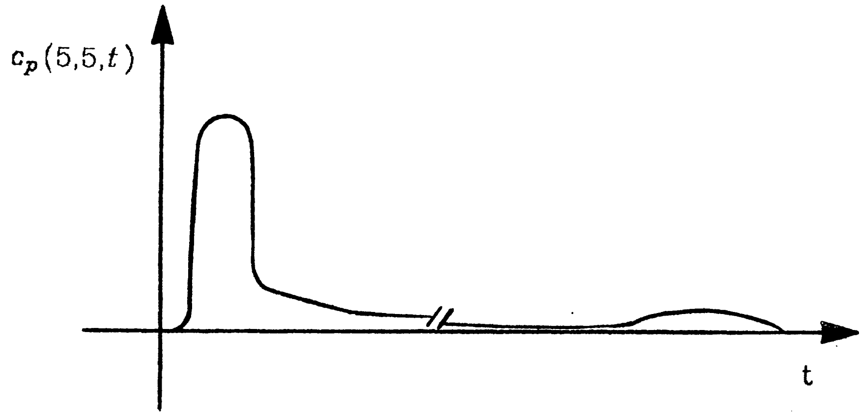
$$\overline{Y}_{p,11,m} = 4121$$

$$\overline{Y^2_{p,11,m}} = 26260400$$

From the specification of the program, it nay be seen that it takes 5000 time units to generate each successor node as given by the state 9 computation time. From the embedded MC (P/M level diagram) it may be seen that 0 successors also take 5000 time units to evaluate.

From state 11 it may be seen that it takes 2000 time units to write a new node into the appropriate GMM. This neglects the manipulation of the local pointer within the GMM.

Table IV summarizes the basic SMP model's calculated results verses simulation values for five separate simulations where q_0, p_E, $S_{p9}(t)$, and $Y_{p11m}(t)$ were varied. Again, no confidence intervals have been compiled so simulations should be regarded as rough estimates. Notice that calculated results are generally very good except in predicting $\overline{C}_p(5,5)$. For simulations 1 through 4 the simulation statistics show that $c_p(5,5,t)$ looks about like:



Where the large peak occurs at a small value of t relative to the smaller peak at a very large value of t. The problem is that a small number of entries (about 20) are seen in the simulation data pertaining to the small peak. This small number typically does not constitute a stable statistic so parameters were varied so the simulation 4 and 5 exhibit better results. Simulation 5 particularly shows better results. Multiple simulations are certainly needed for further testing of the basic SMP model.

States 5 and 6 constitute a loop that each processor uses to wait for data to become available when OPEN is empty. This *communication loop* is a important aspect of parallel program execution. In general processes must communicate through GM locations or special signaling lines that are connected from PE's to PE's. The modeling of process communication is critical. For this simple test, assumptions were made regarding the number of times the communication loop will be executed. Techniques are under development that should provide a formulation for communicating processes.

Analytic and simulation results for the
state space generator algorithm, the simple M/G/1 analysis,
and P = M = 4

Measures	Simulation No.				
	1	2	3	4	5
$\overline{B}_{sim}$	1.417	2.020	1.036	1.975	1.009
$\overline{B}_{calc}$	1.444	1.982	1.093	1.883	1.071
% error	1.9	-1.9	5.5	-4.7	6.1
$\overline{C}_p(1,1)_{sim}$	22613	16408	17417	16917	17889
$\overline{C}_p(1,1)_{calc}$	23248	16944	17290	18030	17993
% error	2.8	3.6	-.7	6.6	.6
$\overline{C}_p(5,5)_{sim}$	86919	73395	65968	15335	17595
$\overline{C}_p(5,5)_{calc}$	116289	84757	86489	18030	17993
% error	33.8	15.5	31.1	17.6	2.3
$\lambda^i_{1,sim}$	4.965×10^{-4}	7.107×10^{-4}	6.545×10^{-4}	8.857×10^{-4}	8.052×10^{-4}
$\lambda^i_{1,calc}$	5.069×10^{-4}	6.955×10^{-4}	6.816×10^{-4}	8.514×10^{-4}	8.531×10^{-4}
% error	2.1	-2.1	4.1	-3.9	5.9
$\lambda^i_{2,sim}$	1.238×10^{-4}	1.812×10^{-4}	1.630×10^{-4}	1.690×10^{-4}	1.564×10^{-4}
$\lambda^i_{2,calc}$	1.286×10^{-4}	1.764×10^{-4}	1.729×10^{-4}	1.658×10^{-4}	1.662×10^{-4}
% error	3.9	-2.6	6.1	-1.9	6.3

Table IV. State space generator example results.

8. Conclusion.

The basic SMP model presented here forms a basis for a model of MIMD systems that is believed to be more comprehensive than models previously developed. Many problems discussed need to be addressed in the future. The remaining sections describe *major* points that need to be studied in order to better model parallel program execution.

8.1. Process Communication.

Modeling process communication is of vital importance, no models yet developed (that model the level of process interaction considered here) are capable of describing system operation appropriately. At the time of writing techniques are under development that appear to offer promise. The basic idea seems to be to include more types of states in the P/M flowchart. Communication states could be included, then the SMP equations which describe process i could be mathematically "linked" to those equations describing process j.

This aspect of future work is considered to be imperative because process communication is a common phenomenon and yet no results regarding the problem have been seen. Note that modeling program/processor behavior using the two state model that predominates other work (as described in section 7) provides *no* reasonable way of modeling communicating processes.

8.2. Transient Analysis.

It seems important to predict the relationship between the steady-state SMP model results and actual system behavior on smaller scale execution times. That is, given the SMP results for a given program set; it would be useful to know how to predict the relationship between authentic short term program execution measures and steady-state values predicted. Reasonable bounds would be useful here.

8.3. Non-stochastic and Synchronized Systems.

An interesting test of the SMP model would be its use on non-stochastic problems. That is, ones in which decisions, computations, etc. are decidedly not random. In this testing, loops executed a constant, known number of times would be replaced by Bernoulli trial loops as in the example in section 3.1.

Deterministic programs also benefit from self synchronization, once they become synchronized they may stay synchronized. Future SMP model development might include modeling synchronization.

8.4. Error Analysis.

It would certainly be useful to predict when and how much error is to be expected in the SMP results. If the M/G/1 analysis is used, then a better characterization of input processes to queues would be useful. The accuracy of approximating input processes as Poisson processes is an area for study. [Cin72] presents a survey of CDF error measures that describe the "distance" between a given CDF and a Poisson process CDF (with exponential renewal times).

Another approach to approximating memory queue behavior might be to use [Fre82] approach to approximating the waiting time seen by arriving requests using an assumed functional form for the waiting time CDF and the Lindley integral equation to match moments and find appropriate coefficients in the assumed function. This technique though is an infinite independent customer analysis.

8.5. State Space Reduction Techniques.

Due to solution complexity dependence on state space size, it might prove advantageous to develop analytic techniques that map a large complicated program (or MC) into a simpler, less descriptive, state digram. It might be useful to develop more accurate processor description than the previously used two state model. The problem here is that fundamental measures are lost when state space reduction techniques are employed.

State diagrams might be developed to model operating system activity along with user program execution. This is similar to the concurrent program execution mode.

8.6. Multiple Requests.

In the case when multiple requests are emitted when a processors enters a reference state, a formulation has been derived but is presently complex and *requires* the CDF of queueing time. The M/G/1 approach might be used with the P-K transform equation of queueing time. The [Fre82] approach is appropriate here because a functional form is assumed for $W_{pm}(t)$, and hence the CDF of queueing time is present.

8.7. Lookahead Applications.

Processors may employ lookahead units in an effort to reduce queueing time effects, an important application of the SMP model is its use in evaluating the selection of lookahead times. If lookahead times are large then memory time is wasted, if lookahead times are small the waiting time will still be experienced by processors. More work is required on this topic.

In conclusion, the basic model presented here may be used to model parallel program execution and processor behavior. The basic model is a generalization of previous models although the concepts and formulation here are different than seen previously. Along with greater generality than previous models allow, the SMP model makes available more information than previous models. Preliminary accuracy tests were shown that suggest that less than 10% error on mean value predictions may be expected.

9. References.

[BaS76] F. Baskett, and A. J. Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory," *CACM*, Vol. 19, No. 6, June 1976, pp. 327-334.

[Bha75] D. P. Bhandarkar, "Analysis of Memory Interference in Multiprocessors," *IEEE TC*, Vol. C-24, No. 9, Sept. 1975, pp. 897-908.

[Cin72] E. Cinlar, "Superposition of Point Processes," *Stochastic Point Processes: Statistical Analysis, Theory, and Applications*, Peter A.W. Lewis, Ed. John Wiley and Sons, Inc. 1972, pp. 549-606.

[Cin75] E. Cinlar, *Introduction to Stochastic Processes*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1975.

[Coo72] R. B. Cooper, *Introduction to Queueing Theory*, The Macmillan Company, New York, 1972.

[DiJ80] D. M. Dias, and J. R. Jump, "Analysis and Simulation of Buffered Delta Networks," *Proc. Workshop on Interconnection Networks*, Purdue University, April 21-22, 1980.

[Fre82] A. A. Fredericks, "A Class of Approximations for the Waiting Time Distribution in a GI/G/1 Queueing System," *Bell System Technical Journal*, Vol. 61, No. 3, March 1982, pp. 295-325

[GrH74] D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*, John Wiley and Sons Inc., New York, 1974.

[Han78] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *CACM* Vol. 21, No. 11, Nov. 1978, pp. 934-941.

[HeS82] D. P. Heyman and M. T. Sobel, *Stochastic Models in Operations Research, Vol. I*, McGraw-Hill, Inc., New York, 1982.

[Hoa74] C.A.R. Hoare, "Monitors: An Operating System Structure Concept," *CACM*, Vol. 17, No. 10, Oct. 74, pp. 549-557.

[Hoo77] C. H. Hoogendoorn, "A General Model for Memory Interference in Multiprocessors," *IEEE TC*, Vol. C-26, No. 10, Oct. 1977, pp. 998-1005.

[Kle75] L. Kleinrock, *Queueing Systems Volume I: Theory*, John Wiley & Sons Inc., New York, 1975.

[Kuc78] D. J. Kuck, *The Structure of Computers and Computations*, Vol. 1, John Wiley & Sons Inc., New York, 1978.

[Law75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE TC*, Vol. C-24, No. 18, Dec. 1975, pp. 1145-1155.

[MaG81] M. A. Marsan, and M. Gerla, *Markov Models for Multiple Bus Multiprocessor Systems*, Report No. CSD 810304, Computer Science Department, UCLA, Feb. 1981.

[MaM81] B. A. Makrucki and T. N. Mudge, *A Multiple $M/D_z/1/L$ Queueing Network Model of Crossbar-based Multiprocessors*, SEL Report No. 157, Dept. of Electrical and Computer Engineering, University of Michigan, September 1981.

[McC73] J. W. McCredie, "Analytic Models as Aids in Multiprocessor Design," *Proc. 7th Annual Conf. on Information Sciences and Systems*, Princeton Univ., March 1973, pp. 186-191.

[MuM82a] T. N. Mudge and B. A. Makrucki, "Probabilistic Analysis of a Crossbar Switch," *Proc. 9th International Symposium on Computer Architecture*, IEEE, April 1982, pp. 311-320.

[MuM82b] T. N. Mudge and B. A. Makrucki, "An Approximate Queueing Model For Packet Switched Multistage Interconnection Networks," *Proc. of the 3-rd Int. Conference on Distributed Computing Systems*, October 1982, (to appear).

[MuM82c] T. N. Mudge and B. A. Makrucki, "Analysis of Multistage Networks with Unique Interconnection Paths," *Proceedings of the 14-th Southeastern Symposium on System* Theory, April 1982.

[MuM82d] T. N. Mudge and B. A. Makrucki, "Analysis of a Multiport Memory," *Proceedings of the 16-th Annual Conference on Information Sciences and Systems*, Princeton University, March 1982, pp. 639-643.

[Nil71] N. J. Nilsson, *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill, Inc., New York, 1971.

[Pat79] J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," *Proc. 6th Annual Symp. on Computer Architecture*, IEEE, April 1979, pp. 166-177.

[Rau79] B. R. Rau, "Interleaved Memory Bandwidth in a Model of a Multiprocessor Computer System," *IEEE TC*, Vol. C-28, No. 9, Sept. 1979, pp. 678-681.

[Ros70] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, Inc., San Francisco, 1970.

[SeD79] A. S. Sethi, and N. Deo, "Interference in Multiprocessor Systems with Localized Memory Access Probabilities," *IEEE TC*, Vol. C-28, No. 2, Feb. 1979, pp. 157-163.

[SeM81] K. C. Sevcik and I. Mitrani, "The Distribution of Queueing Network States at Input and Output Instants," *JACM*, Vol. 28, No. 2, April 1981, pp. 358-371.

[Sie80] H. J. Siegel, (Ed.), *Proc. Workshop on Interconnection Networks*, Purdue University, April 21-22, 1980.

[SkA69] C. E. Skinner, and J. R. Asher, "Effects of storage contention on system performance," *IBM Systems Journal*, No. 4, 1969, pp. 319-333.

[Smi74] A. J. Smith, *Performance Analysis of Computer System Components*, Ph.D. Thesis, STAN-CS-74-451, Computer Sci. Dept., Stanford Univ., August 1974.

[Str70] W. D. Strecker, *Analysis of the Instruction Execution Rate in Certain Computer Structures*, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, 1970.

[Tak62] L. Takacs, *Introduction to the Theory of Queues*, Oxford University Press, Inc., New York, 1962.

[WuF80] C-L. Wu, T-Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-29, No. 8, August 1980, pp. 694-702.

10. Appendix.

Since connection times for different processors are independent, and are added for those requests in the queue at processor p's request arrival, the pmf of waiting time is given by the convolution of connection times (and the excess connection time for the connection in progress). In terms of transforms it is given by the product. In the first case when $|\delta| = 0$, the queue is empty so the waiting time CDF is $u(t)$ which has pmf $\delta(t)$. $\delta(t)$ has an LT of 1.