NASA Contractor Report 3920

Modeling and Measurement of Fault-Tolerant Multiprocessors

Kang G. Shin, Michael H. Woodbury,
and Yann-Hang Lee

GRANTS NAG1-296, NAG1-492, and NGT 23-005-801
AUGUST 1985

NASA

NASA Contractor Report 3920

Modeling and Measurement of Fault-Tolerant Multiprocessors

Kang G. Shin, Michael H. Woodbury,
and Yann-Hang Lee
University of Michigan
Ann Arbor, Michigan

Prepared for
Langley Research Center
under Grants NAG1-296, NAG1-492, and NGT 23-005-801

National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1985

TABLE OF CONTENTS

1. INTRODUCTION 1

2. PERFORMANCE MODELING OF REAL-TIME MULTIPROCESSORS 2

   2.1. Introduction 2

   2.2. System Architecture and Operation 5

   2.3. Stochastic Petri Net Model 9

   2.4. Queueing Model Description 16

   2.5. Solutions to the Queueing Model 21

   2.6. Description of Experimental System: FTMP 24

   2.7. Queueing Model Representation of FTMP 28

3. MEASUREMENT OF FAULT LATENCY 34

   3.1. Introduction 34

   3.2. Methodology for Measurement of Fault Latency 40

   3.3. Experimental Results and Analysis on FTMP 41

4. CONCLUSION AND DISCUSSION 51

   ACKNOWLEDGMENT 54

   REFERENCES 55

LIST OF FIGURES

Figure 1. System Architecture 6

Figure 2. Modified SPN Model 11

Figure 3. Queueing Model 17

Figure 4. A Block Diagram of FTMP (from [15]) 27

Figure 5. Probability of Bus Contention vs. Bus Service Rate (μ_S) 35

Figure 6. Probability of Bus Contention vs. Service Rate of Job Class 1 (μ_1) 36

Figure 7. Probability of an Idle Cluster vs. Idle Service Rate (μ_1) 37

Figure 8. Probability of an Idle Cluster vs. P_I 38

Figure 9. The Experimental Results and Estimated Distributions for Stuck-at-0 Faults 46

Figure 10. The Experimental Results and Estimated Distributions for Stuck-at-1 Faults 47

Figure 11. The Experimental Results and Estimated Distributions for Inverted Signal Faults 48

Figure 12. The Experimental Results and Estimated Distributions of Fault Latencies at System Bus Controller 49

LIST OF TABLES

Table 1. Place Descriptions 13

Table 2. Transition Descriptions 14

Table 3. Experimental Measurements 29

Table 4. Markov State Descriptions and Steady State Probabilities 31

Table 5. Parameter Values 32

Table 6. Idle Processors and Bus Contention Probabilities 33

Table 7. Experimental Results and Estimated $h_f(t_j)$ 44

Table 7. Experimental Results and Estimated $h_f(t_j)$ (cont'd) 45

Table 8. Least-Squares Estimation of the Distributions of Fault Latencies 50

1. INTRODUCTION

This report deals with both the modeling and measurement of fault-tolerant multiprocessors. A detailed analysis of systems of this type is desired because of the increasing number of mission-critical situations in which they are used. One would like to be able to predict the performance of such systems for various workloads and how well they recover from system errors. The speed and effectiveness of the recovery procedures for a fault-tolerant multiprocessor have a direct effect on its performance.

In the first part of this report we present a model to analyze the performance of a unibus[1] multiprocessor. A closed queueing network is developed to study the effects of workload variation on bus contention, processor utilization, and performance. This development entails representing the computer system with a modified Stochastic Petri Net(SPN). This aids in illustrating the operation of the specific system and determining which factors have the most significant effect on performance.

A second component of this report pertains to the measuring of fault latency in a multiprocessor environment. This entails explicitly determining the distribution of fault latency and its significance in system modeling and analysis. The result of this research shows that fault latency is significant and that the common assumption of a negligible fault latency may be incorrect.

An existing system, the Fault-Tolerant Multiprocessor (FTMP) located at the NASA AIRLAB[17-20], is used as a modeling example. Many experiments have been made on this system to measure fault latency and performance related factors, such as bus contention and idle processors. It is the results of some of these experiments that justify the conclusions drawn concerning fault latency.

---

[1]This unibus can consist of redundant buses which logically act as a unibus.

The rest of this report is organized as follows. Section 2 deals with the modeling of fault-tolerant unibus multiprocessors and is divided into seven subsections. In Subsection 2.1 the performance modeling is introduced. Subsection 2.2 describes the specific architecture being addressed, a real-time unibus multiprocessor, and its operation. Subsections 2.3 and 2.4 describe the SPN model and the closed queueing network model, respectively. The results of the queueing model and closed form solutions are presented in Subsection 2.5. The experimental system, FTMP, is described briefly in Subsection 2.6. Subsection 2.7 shows the queueing model representation of FTMP and some measured experimental results pertaining to its performance.

In Section 3, we present the technique of characterizing fault latency, which is an important system parameter for modeling computer systems. Subsection 3.1 introduces the concept and approach to measuring fault latency. A methodology for measuring fault latency is outlined in Subsection 3.2. An example of the application of the method on FTMP is shown with experimental results in Subsection 3.3. Finally, the report concludes with Section 4.

2. PERFORMANCE MODELING OF REAL-TIME MULTIPROCESSORS

2.1. Introduction

Representing the operation of a computer system by a structured model is a popular and natural approach to the study of a computer's performance. Many factors need to be incorporated into the model so that it accurately describes the system that is being modeled. The type of analysis desired dictates which factors of the computer's operation need to be incorporated into the modeling framework. A factor that is almost always included, especially in the study of computer performance, is the representation of the

workload handled by the computer system being analyzed. The workload is an essential part of the performance evaluation of any computer system, because how well a computer performs is directly related to the type of workload it is handling.

First, we present the beginning stages in the development of a model to study the workload effects on performance for a specific computer architecture and application. The type of system being addressed is a highly reliable unibus[2] multiprocessor that is used in real-time control. Dealing exclusively with real-time systems in the evaluation of multiprocessor performance is an approach that has not been largely addressed in the literature. Usually, a general purpose multiprocessor is discussed, as in [1-3]. This type of approach is difficult because of the largely varied workload general purpose systems handle. Trying to represent a system of this type with its workload becomes unreasonably complex, if one wants to properly describe the workload effects on performance. It appears that a number of interesting results can be obtained if one only considers the structure of a real-time system and its workload.

The detailed analysis of this type of system is desired because of the increasing number of critical situations it is used for, e.g., control of aircraft, spacecraft, nuclear reactors, etc., where the failure of the controlling computer would result in catastrophic losses. A failure could be the result of a physical malfunction or the result of the system not reacting quickly enough as required[4].

Many authors have presented designs for synthetic workloads [5-8]. They have usually relied on heuristic methods that seem to provide an adequate workload for a general class of computing systems. Recently, Ferrari [9] has made the point that a more systematic method is necessary, because of the fundamental correlation between work-

[2]As mentioned earlier, this can be redundant buses.

load modeling and any performance evaluation. Developing such a method is more complicated than it might first appear. One first needs to define what the workload model should cover in its representation, and what standard should be used to determine if a workload model is a "good" model.

We view a real-time computer system as the combination of two closely dependent components: the *controlled process* and the *controlling computer* [4]. Because of this close dependency, we feel that the development of a synthetic workload for this type of system should not only rely on the actual workload being modeled, but it should also depend on the type of system handling the workload. It is this basic association that sets our work apart from those of others. Having a specialized synthetic workload of this type provides us with a means for producing more useful results relating to the performance evaluation of real-time computing systems.

Typically, the workload of a real-time system is a fixed group of tasks that have to be performed at certain intervals, repeatedly. There is usually a group of short, frequently initiated tasks that monitor internal and external conditions and continually compensate for their change. There are also tasks that are initiated less frequently that require more computation time. The relative frequencies of the initiation of tasks, and the number of tasks that need to be completed in a certain time frame lead to strict performance criteria.

It would be desirable to be able to determine if a computer system with the architecture mentioned above could handle a given workload and set of performance criteria. If it can, one would like to know how this might be best accomplished. And finally, it would be useful if this optimal performance could be measured. The model presented here will hopefully aid in solving some of these problems.

Vital factors can be determined directly from the model, such as the amount of processor idle time, the degree of contention for the single bus, and the tasks that have the most significant effect on performance. These details will be discussed in a later subsection. The model can also be used as a tool for determining the optimal workload distribution to reach a certain level of performance.

2.2. System Architecture and Operation

As mentioned earlier, the hardware system addressed here is a highly reliable unibus multiprocessor. The general structure of such a system is shown in Figure 1. It consists of four major components *processing clusters, input/output links, a time-shared system bus,* and *system memory.* A description of each of these will be discussed as well as their assumed interdependencies.

A *processing cluster* is an entity that is capable of operating on one task at a time. It consists of one or more pairs of a processing unit and its local memory. The degree of redundancy is considered immaterial to the performance of the cluster for a given task. Although, the redundancy does have a significant impact on the reliability and configuration aspects of system operation. What is important is that regardless of how many pairs there are in a cluster, they all work together on a single task. For example, a cluster may represent a triple modular redundant (TMR) system of three processing units and their local memories. It is also assumed that all the clusters in the system are of the same type, i.e., they all contain the same number of processor-memory pairs.

An *input/output link* is a component that enables data to be transmitted to or from an external device. These allow the system to read data from sensors and transmit data to actuators and displays. These links are also the channels used for human interface through terminals or other similar devices.

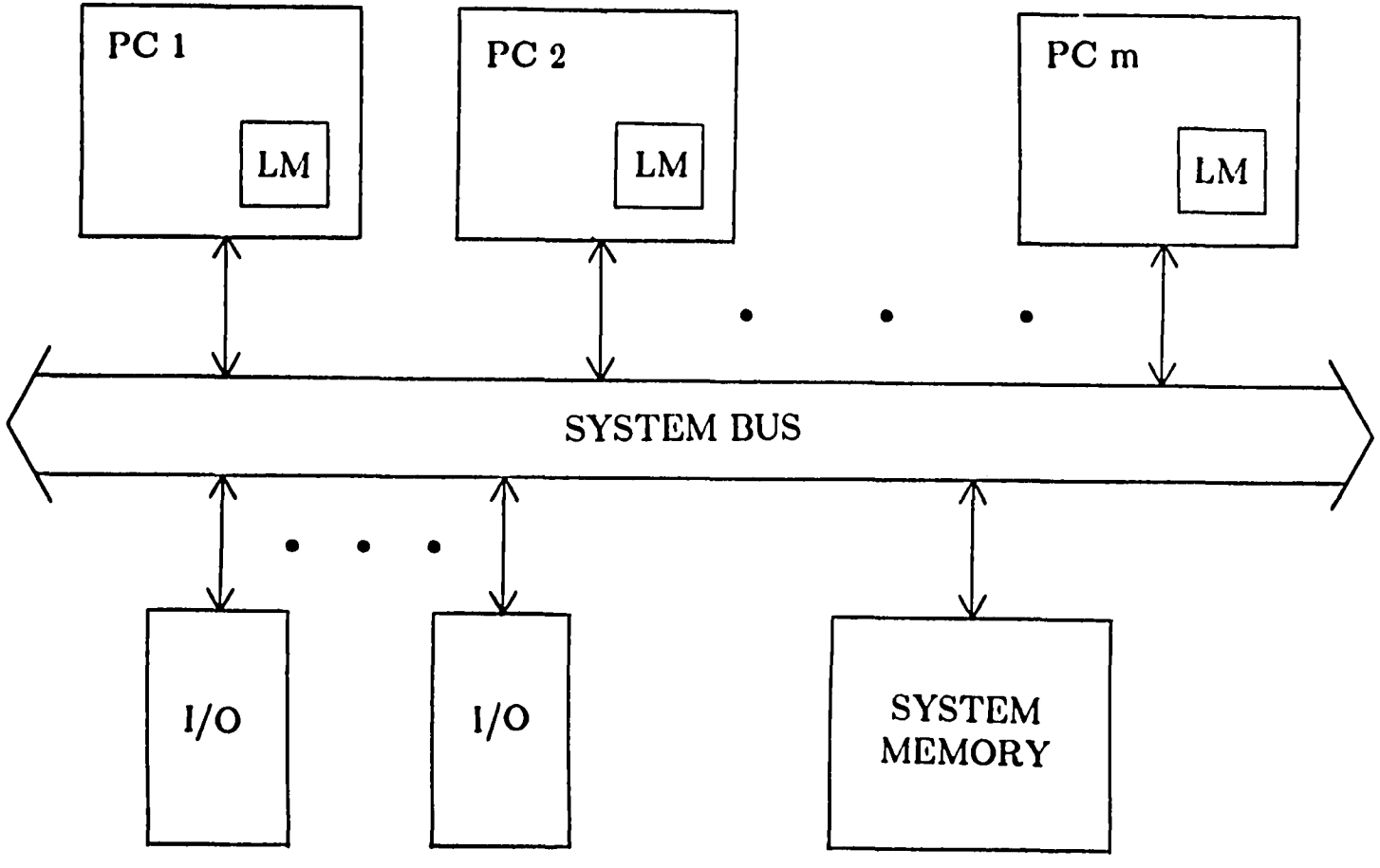

Figure 1. System Architecture

PC = PROCESSING CLUSTER
LM = LOCAL MEMORY
I/O = INPUT/OUTPUT LINK

The *time-shared system bus* interconnects all the processing clusters, I/O links, and system memory. It is the medium for exchanging all data and control signals. Again, this bus may be redundant for reliability reasons, but only one cluster transmits and receives data over all copies of the bus at a time. Therefore, the redundant system bus logically acts as a unibus. A cluster communicating over the bus is said to *control* the bus.

Finally, there exists a single *system memory* that is addressable over the system bus. This memory usually consists of a collection of dynamic RAMs. The system memory may be redundant with the restriction that only one system memory location may be addressed at a time.

The basic operating principles of this multiprocessor system can be explained as follows. All tasks to be executed by the system are stored in system memory. These tasks can be divided into n *job classes,* where a job class consists of tasks that are required to repeatedly execute at the same relative frequency. More specifically, tasks of job class i are executed every r_i seconds, where $\frac{1}{r_i}$ is the frequency of initiation of a task of job class i. There may be more than one job class having the same relative frequency for its tasks. The set of job classes is a partition of the set of system tasks, where a task is in one and only one job class.

Each job class is given a priority. This priority is used to determine which processing cluster may use the system bus when there is a contention among clusters for bus control. A cluster about to work on or currently working on a task from job class i has priority over another cluster to control the system bus, if the other cluster is about to work on or is currently working on a task from job class j, where $1 \leq i < j \leq n$. Priority of clusters working on tasks of the same job class is determined by a first come

first served (FCFS) policy. Task queues are kept for each job class and these reside in system memory also.

An idle cluster wishing to process a task from job class i must first gain control of the system bus. It does this by waiting for inactivity on the bus and proceeds to participate in a *polling sequence*. A polling sequence is a decision process to determine which cluster has the highest priority. This is conveniently done by requiring each of the clusters to transmit their priority number over the system bus and having a voting mechanism determine which cluster has the highest priority. As a result of the polling sequence, the cluster with the highest priority is given control of the bus.

At this time the cluster reads the task queue for job class i from system memory and determines the next task to be executed. It then reads in the task and *all* data necessary to process that task. This data can be obtained from I/O link reads or more system memory reads. After obtaining all the information necessary to internally execute the task, the cluster updates the job queue in system memory and releases the bus. There are other mechanisms such as counters, queues, and interrupt timers to aid a cluster in determining which job class to request. When a cluster completes a task, it will again request bus control and transmit its results to the relevant addresses, determine which job class to work on next, and proceed as before.

At any particular instant, all the clusters could be processing tasks simultaneously resulting in peak performance. Performance dwindles when a cluster becomes idle waiting for control of the system bus. There is also a penalty in performance, or system failure, if all the clusters are not able to keep up with the required frequency of task execution for each job class.

A reasonable question to address is how are the job classes formed? More specifically, given a system workload, what is the best number of job classes and the distribution of the tasks among these classes? For a general purpose computing system's workload, this is difficult to determine [8]. Some of the main problems in representing the workload in a general purpose multiprocessor system model are (1) showing the interdependencies among tasks in the workload, (2) the fact that the workload may not be stationary, i.e., tasks of one type might occur at different rates at different times, (3) the unlimited number of tasks possible, and (4) the contention for physical components needed to execute tasks operating concurrently. Providing a model that is able to represent all these features would be extremely difficult, if not impossible. Fortunately, when one only considers real-time applications on a unibus system these problems become relatively easier to address. The workload of a real-time system is usually a fixed set of tasks that have to be executed in a prescribed order at regular intervals. This makes determining the physical and logical interdependencies more tractable. It also implies a stationarity among the relative frequencies of different tasks. Therefore, natural job classes can be formed and parameterized. However, this still is not an easy task.

2.3. Stochastic Petri Net Model

In the development of the model, it was first necessary to represent the overall operation of the system at some level of abstraction that would be amenable to the type of performance analysis desired. This representation is needed to depict the various states a processing cluster might be in. The features that have a significant effect on performance are system bus contention, transmission delays, and possible idle periods of a processing cluster. By modeling at the system level, where the components of concern

are the tasks, clusters, and system bus, we are able to describe the stages a processing cluster will go through and how its actions affect the operation of the other clusters.

A useful tool for showing synchronization among system components is a *Stochastic Petri Net* (SPN) [10-11]. Figure 2 is an example of a modified stochastic Petri net which describes the synchronous actions of the system referred to in this report. This is a modified SPN because of the presence of the three function blocks F1, F2, and F3.

A SPN is a structure consisting of *places, transitions,* and *directed arcs* connecting transitions and places. A place is usually represented in a net drawing as a circle, while transitions are shown with bars. Directed arcs connect these places and transitions in a way that there is no arc going directly from a place to another place, or from a transition to a transition. *Tokens* or dot markings in a place represent collectively the state of the SPN.

A transition will *fire* when it becomes *enabled.* A transition is enabled when there exists at least one token in each input place to the transition. The process of firing a transition results in one token being removed from each place for each arc entering the transition, and a single token placed in all of the places that have input arcs emanating from that transition. A transition may fire *instantaneously,* such transitions are represented by solid bars (T1 - T9 in Fig. 2), or have an *exponentially distributed random duration,* such transitions are called *timed transitions* and are represented by hollow vertical bars (T10 - T21). When an instantaneous transition is enabled, tokens are immediately removed from input places and sent to output places. When a timed transitions is enabled, there is an exponentially distributed delay before tokens are removed from input places and immediately sent to output places.

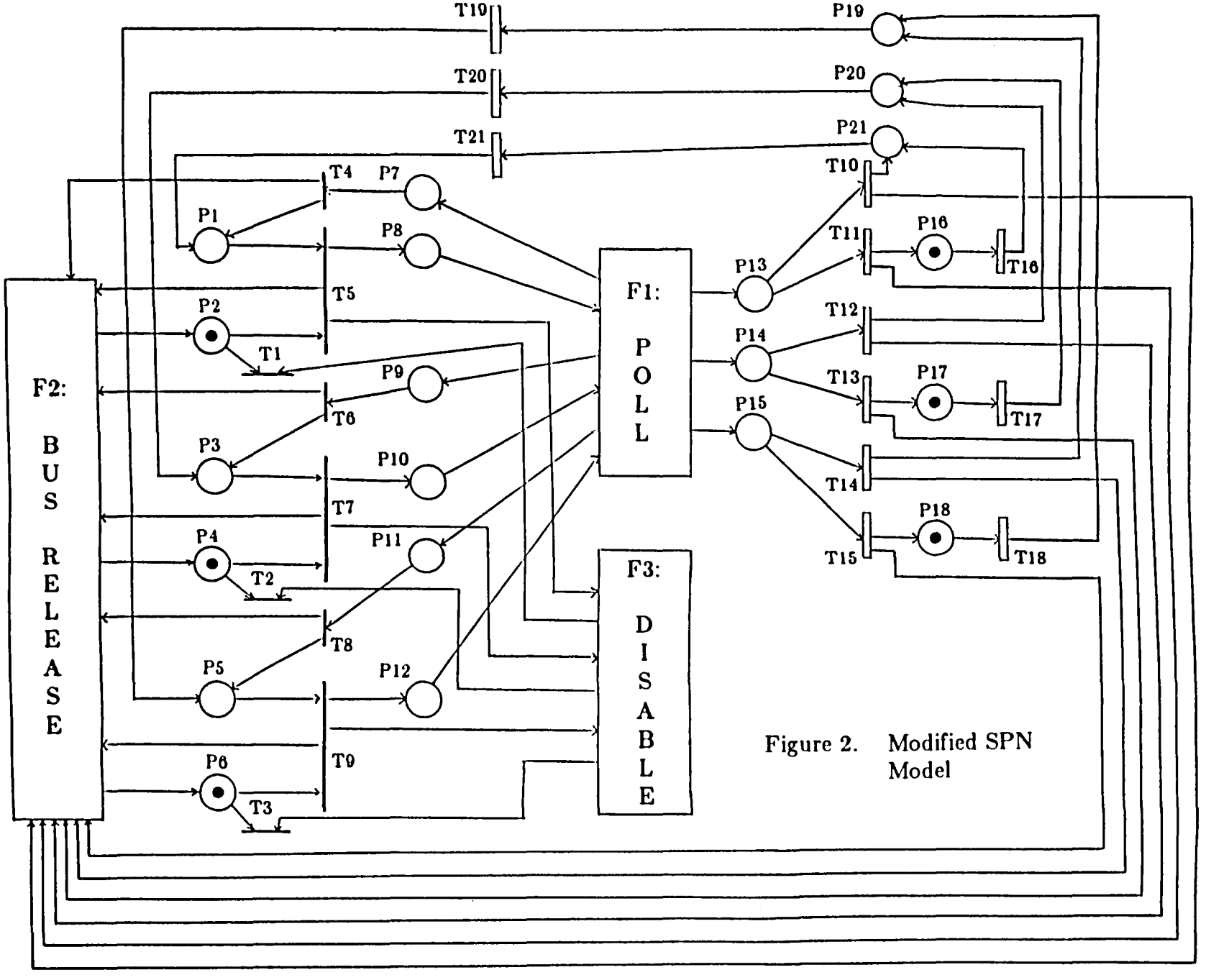

Figure 2. Modified SPN Model

The function blocks in Figure 2 are not defined components of a true SPN. They are used here to simplify the appearance of the figure. The functions represented by each block have been expressed as SPNs themselves. They act on the input arcs to the block and produce tokens at the output arcs. The functions they represent are trivial in nature, but the SPNs are complex and cloud their simplicity. For example, F3 has been expressed using 13 places, 11 transitions, and 46 directed arcs.

Figure 2 is an SPN for a three cluster, unibus multiprocessor. How a single cluster is incorporated in this model will now be explained. The extension from one cluster, to three, and more will be simple to envision. There are seven places (P1, P2, P7, P8, P13, P16, and P21), three instantaneous transitions (T1, T4, and T5), and four timed transitions (T10, T11, T16, and T21) necessary to describe the operations of a single cluster. What they represent is described in Table 1 and Table 2.

The F1 (Poll) function block is activated whenever there is a token present in places 8, 10, or 12, i.e., when there is a poll request. It performs the action of removing these tokens if they are present and deciding which of the requesting clusters should obtain control of the bus. On output one token will be placed in either place 13, 14, or 15, depending on which cluster has gained control of the bus. There will also be tokens placed in places 7, 9, and 11, if that cluster has lost the poll sequence. For example, suppose cluster 1 and cluster 2 both initiate a poll sequence and cluster 1 is to succeed in the poll. Initially, there would be a token in places 8 and 10. This indicates that cluster 1 (place 8) and cluster 2 (place 10) wish to initiate a poll sequence. The Poll function would remove these tokens, and after a delay representing the time it takes to perform a poll, will place a token in place 13 (cluster 1 has succeeded) and one in place 9 (cluster 2 has lost the poll).

Place	A token in this place means that ....
P1	a system bus request has been made by the cluster.
P2	the system bus is free as seen by the cluster.
P7	the cluster has lost a poll sequence.
P8	the cluster is initiating a poll sequence.
P13	the cluster has succeeded in a poll sequence and has been granted bus control.
P16	the cluster has completed its bus transactions and is to become idle.
P21	the cluster is ready to begin processing a task.

Table 1. Place Descriptions

Transition	The firing of this transition represents ...
T1	the cluster determining that the bus is busy.
T4	the cluster acknowledges that it has lost a poll sequence and must wait to make another request for the system bus.
T5	the cluster initiating a poll sequence.
T10	the cluster transmitting on the system bus.
T11	the cluster transmitting on the system bus.
T16	the cluster remaining in an idle state.
T21	the cluster internally executing a task.

Table 2. Transition Descriptions

Functions F2 (Bus Release) and F3 (Disable) act to indicate that the bus has become free or is busy. Function F2 acts by keeping track of which clusters are in a poll sequence, thus transmitting on the system bus, or which are communicating over the system bus. When all activity is completed by all the relevant clusters, the F2 function will indicate that the bus is free by placing a token in places 2, 4, and 6. Function F3 acts to disable other clusters from initiating a poll sequence if the bus is currently busy. Therefore, when a poll request is made the F3 function will determine which of the clusters should be disabled.

Figure 2 completely describes the system we are interested in. One is able to follow the actions of a single processing cluster and observe the effects of these actions on the rest of the system. The model serves the purpose of enabling us to see which actions of a computing cluster have the greatest effect on system performance. For example, by supplying transition rates for the timed transitions, one could determine how often a bus request is made. Combining this with information on the duration of a typical transmission will give us an idea of how often the bus is busy. With this result, it can be intuitively stated that the higher the bus request frequency is, the greater the possibility of bus contention.

It can be observed that this model, were it completely expressed with valid SPN components, would be cumbersome and confusing. Malloy [10] has shown that SPNs are isomorphic to continuous parameter Markov chains. An SPN can be converted to a Markov chain and completely analyzed. One drawback of this method is that the state space for such a Markov chain is large. It is unmanageably large for the example of Figure 2 (keep in mind that the SPN for a function block is larger than the rest of the model shown). Therefore, it is obvious that using this model directly as a tool for per-

formance evaluation of the system of interest is inappropriate. A simpler model has to be derived that expresses the same relationships. Such a model is introduced in the next section.

2.4. Queueing Model Description

The model presented in this section is designed to represent the states of a unibus multiprocessor system. The state of the system is defined by a combination of the states of all the processing clusters. With the aid of the model outlined in the previous section, the states that were determined to be relevant to system performance are when a processing cluster is (1) competing in a poll sequence, (2) communicating on the system bus, (3) processing a task from job class i, or (4) idle, i.e., not processing a task. The relationship between these states of a processing cluster and the relationship between clusters can be inferred from Figure 2.

These relationships are incorporated into the closed queueing network shown in Figure 3. This model has a number of advantages over the SPN model, besides the obvious of being simpler to understand. First, it reduces all the actions of bus contention and the polling sequence into a single *non-preemptive priority queue*. A non-preemptive priority queue is one where each of the arriving customers has an associated priority. A customer entering the queue will move ahead of all the customers in the queue that have lower priorities, and behind those of equal or higher priority. In this manner, customers of the highest priority in the queue are served first on a FCFS basis. The second advantage is that the separate job classes can be explicitly parameterized in this model, whereas in the SPN model they were all grouped together. Third and most importantly, this model can be easily solved for a given set of parameters.

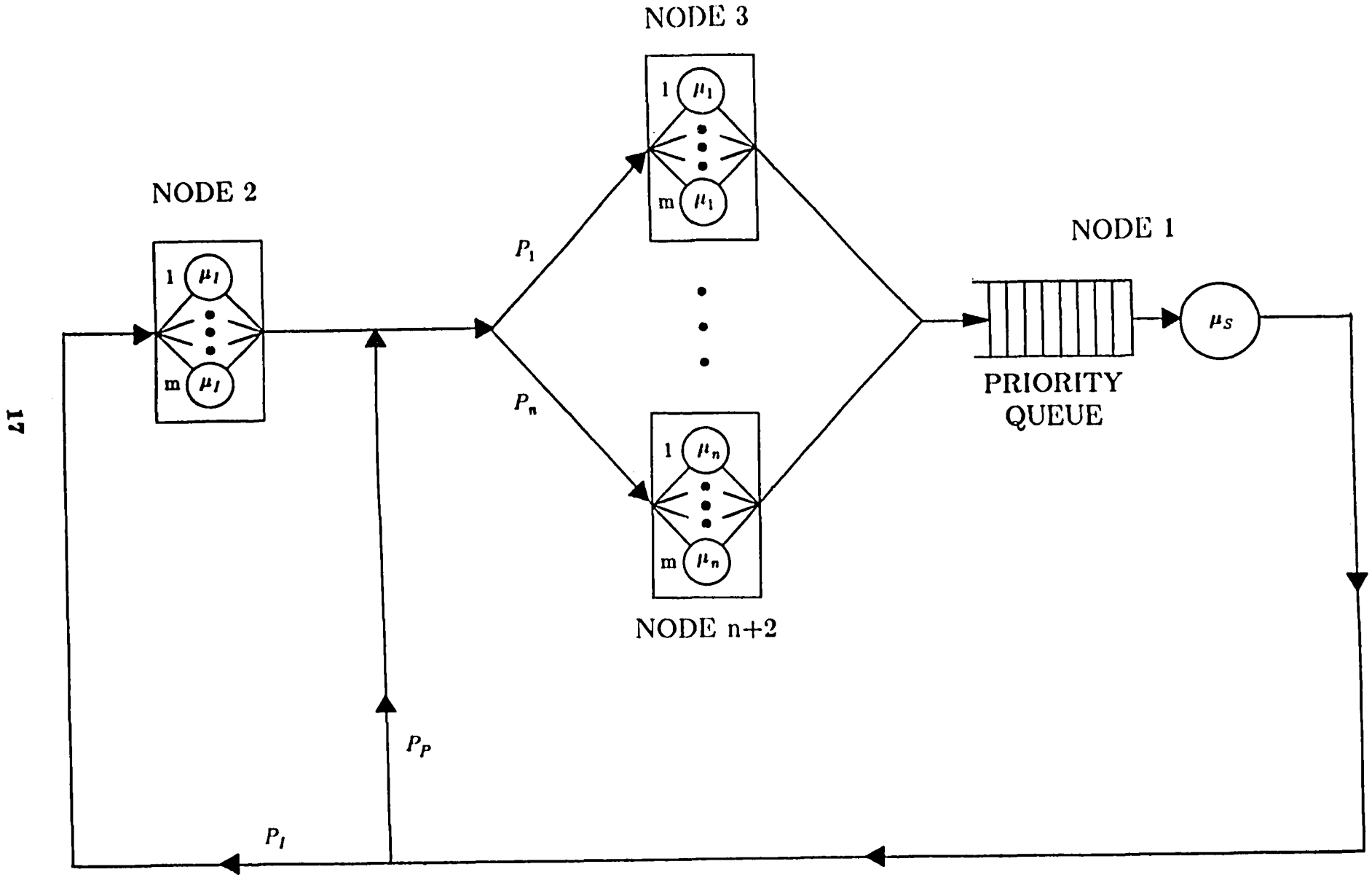

Figure 3. Queueing Model

Before describing the details of this queueing model, it should first be noted that the parameters and node representations of this model differ from those of conventional queueing models. Typically, the nodes of a queueing model represent servers of some type, e.g., processors, workers, etc. The associated parameter for each node usually describes the exponential service rate for the server. The tokens or markings moving about the model represent customers that desire service, e.g., programs, jobs, etc. The actions of a closed queueing model can be described as a token arriving at a node, waiting, if necessary, a certain length of time for service, being served for a length of time, and moving on to the next node. The model described here reverses the conventional meanings of node and token. In this model, a node represents a customer that needs service, and the associated exponential service rate describes how long it takes to complete that service. The tokens on the other hand represent servers, where all the servers are identical. Therefore, this model represents servers moving from customer to customer and performing the service requested by that customer. This unorthodox convention is used because (1) it simplifies the model, and (2) it explicitly shows the state the system is in by showing what state each processing cluster is in.

It is the goal to determine the steady state probabilities for the distribution of clusters among the different states.[3] Since it is safe to assume that the system will reach steady state before a cluster fails, the number of clusters remains constant in the analysis. Typical values for the mean time between failures (MTBF) are in the order of 10^3–10^4 hours. Whereas, steady state can be reached in a matter of minutes at most.

Once steady state is reached, a cluster may fail. At that point we have a system with one less cluster, and it is reasonable to assume that *this* system will reach steady

[3]This will be shown in Section 2.5.

state before another failure occurs. The performance of this degraded system will be less than that of the previous system. To obtain the overall performance of the system operating over a certain length of time, the performance contributions of each of the configurations are combined, weighted by their relative time of operation. Therefore, in the following analysis, we will assume that no cluster fails and that the number of clusters remains constant.

In this model m equals the total number of homogeneous clusters in the system. Since the number of tokens in this closed queueing model remains constant, it is justifiable to have each token represent a cluster. Therefore, there are exactly m tokens present in the system at all times. The nodes represent the activities that are performed by a cluster, e.g., a cluster is in the idle state if it is idle.

There are $n + 2$ nodes in this model. Again, n is the number of different job classes in the workload. As stated before, tasks that belong to the same job class are assumed to each have the same distribution of internal processing time. It is assumed that this processing time is an exponentially distributed random variable. The number of tasks in a job class has to be greater than or equal to one. Each of these job classes is given a priority level, where all tasks of the same job class have the same priority and a task from class i has priority over a task of class j when $1 \leq i < j \leq$ n.

Each of the nodes will be described below.

NODE 1 : This node represents the transmission activity over the system bus. It consists of a non-preemptive priority queue and a transmission server. A token at this node represents a cluster that is either waiting to transmit on the system bus or currently transmitting. The parameter μ_S describes the the transmission rate of a cluster, i.e., $\frac{1}{\mu_S}$ is the average transmission duration.

A non-preemptive priority queue is used to show that a cluster that has just completed a task from class i is given priority to transmit over a cluster that has completed a task of class j, where again $1 \leq i < j \leq$ n. Clusters completing tasks of the same class are able to transmit on a FCFS basis.

NODE 2 : A token at this node represents a cluster that is idle, i.e., performing no useful computations. It is a multiserver node with m servers. A node of this type is used to indicate that all the clusters may be served at this node with no queue forming. This is equivalent to saying that all the clusters may be idle at the same time. The sojourn time in this idle state for a cluster is assumed to be exponentially distributed with rate μ_I. The rate at which clusters leave this node is $k\,\mu_I$, where k is the number of tokens being served by the node.

NODES 3 through n+2 : These n nodes represent the different job classes. Node $i+2$ represents a processing activity on a task of class i. Again, as with node 2, these are multiserver nodes with m servers. Thus, no queue forms at any of the nodes. This type of node is used to indicate that all the clusters could be working on tasks from the same job class. The parameter μ_i is the rate describing the processing duration of a task of class i. Typically, $\mu_i \geq \mu_j$ when $i < j$. The rate at which clusters leave the node $i+2$ is $k\,\mu_i$, where k is the number of tokens being served by the particular node.

The final parameters in the model that need explanation are the branch probabilities. When a cluster completes a transmission, it either drops into the idle state or continues processing. The probability that the next state is the idle state is P_I and similarly, the probability that the next state is a processing state is P_P. Obviously,

$P_I + P_P = 1$. When a processor is to enter a processing state, there is the probability P_i of it being the processing of a job of class i, where $\sum_{i=1}^{n} P_i = 1$. Typically, $P_i \geq P_j$ when $i < j$.

2.5. Solutions to the Queueing Model

The common approach to solving for the steady state probabilities of a queueing model is to convert the model to that of a continuous parameter Markov chain [12]. This approach will be used to solve the queueing model presented here. For the construction of a Markov chain, we make the following definitions.

Definition 1 : A *cluster state* is a pair (c_i, n_i), where $c_i \in \{1,2,...,m\}$ is a number labeling a particular processing cluster, and $n_i \in \{1,2,...,n+2\}$ is the number of the node where the token representing the cluster is located. There are $m \cdot (n+2)$ cluster states.

Definition 2 : A *system state* is an m-tuple $(s_1, s_2, \ldots, s_m) \in S_1 \times S_2 \times \cdots \times S_m$ where S_i is the set of all cluster states whose first component is c_i. There are $(n+2)^m$ system states.

An example of a system state for a system with three clusters and three job classes is ((1,1),(2,3),(3,1)). This represents the configuration when clusters 1 and 3 are waiting to communicate on the system bus or are currently communicating, and cluster 2 is processing a task from job class 1.

From an analysis standpoint, a system state contains more information than is necessary. We are only concerned with how many clusters there are at a particular node. We do not need to know which they are, because they all require the same amount of time to process the task at a particular node. It is the number of clusters

that determines how fast tasks are completed or delayed at a node. This motivates the following definition.

Definition 3 : A *reduced system state* is the n+2-tuple $(a_1, a_2, \ldots, a_{n+2})$, where $a_i \in \{0,1,\ldots,m\}$ is the total number of tokens representing clusters at node i. There are m·(n+2) reduced system states.

We can define a formal mapping, Φ, from a system state to a reduced system state as follows: $\Phi(s_1, s_2, \ldots, s_m) = (a_1, a_2, \ldots, a_{n+2})$, where a_i = number of s_j's (j=1,...,m) whose second component is i. Referring to the example above, we note that the system state ((1,1),(2,3),(3,1)) is represented by the reduced system state (2,0,1,0,0). It should also be noted that the system states ((1,1),(2,3),(3,1)), ((1,1),(2,1),(3,3)), and ((1,3),(2,1),(3,1)) are all represented by the same reduced system state.

We use the reduced system states as the states of the Markov chain. The transitions between these states is defined by the relevant service rates of each of the nodes in the closed queueing network. It has been stated by Kleinrock [13] that a closed queueing model of this type with K customers and N nodes has $J = \binom{N+K-1}{N-1}$ states in its Markov chain representation. For our model, we have m customers (clusters) and $n+2$ nodes. From this Markov chain, a J × J transition rate matrix, A, can be formed and used to derive the steady state probabilities for each state in the Markov chain. This involves solving the matrix equation $\mathbf{Ax}=\mathbf{0}$, where $\mathbf{x} = (x_1, x_2, \ldots, x_J)^T$ and x_i represents the steady state probability of the system being in state i. A nontrivial solution results when the constraint $\sum_{i=1}^{J} x_i = 1$ is considered. The existence of such a solution is based on the fact that we have constructed a finite state, irreducible, and

recurrent Markov chain.[4] Since it is possible for a token in the queueing model to move from one node to any other node, either directly or through some intermediate nodes, *and* there is a non-zero probability that a token leaving a node will return to that node, the Markov chain is indeed irreducible and recurrent.

Once the steady state probabilities are determined, two useful results concerning the multiprocessor system can be quickly obtained. One is the probability that a cluster is idle. This is simply the sum of the probabilities for each of the Markov chain states that represent having one or more clusters at node 2. The other result is the amount of system bus contention. When there is more than one cluster at node 1, there is a cluster waiting to obtain bus control. Again, all that has to be done is to sum the probabilities for each of the Markov chain states that represent having more than one cluster at node 1. Recall that node 1 includes both the priority queue and the transmission server. These two results are necessary to produce a performance measure of any type.

A third result can also be easily obtained. It would be interesting to know how long a cluster would have to wait, on the average, if there is contention for the system bus. It has been shown by a number of authors that the average queueing time for customers of a given priority class in a non-preemptive priority queue can be determined [14-16]. The average queueing time for a customer of priority class i is

$$W_i = \frac{\frac{1}{2}\sum_{j=1}^{k} \alpha_j c_j}{\left(1 - \sum_{j=1}^{i-1} \frac{\alpha_j}{\mu_j}\right)\left(1 - \sum_{j=1}^{i} \frac{\alpha_j}{\mu_j}\right)}$$

where

k = the number of priority classes.
α_j = the probability that an arriving customer is of class j.

[4]A unique steady state solution exists for this type of Markov chain.

μ_j = the mean service rate of a customer of class j.
c_j = the second moment of the service-time distribution for customers of class j.

The mean queueing time of all customers is $W_q = \sum_{j=1}^{k} \alpha_j W_j$.

For the model described here, all clusters requesting service at node 1 require the same amount of service time. Therefore, for this example we have $k = n$, $\mu_i = \mu_S$ for all i, and $c_i = \frac{2}{\mu_S^2}$ for all i. We then arrive at the average queueing time for a cluster about to work on a task from job class i, W_i.

$$W_i = \frac{1}{\left(\mu_S - \sum_{j=1}^{i-1} \alpha_j\right)\left(\mu_S - \sum_{j=1}^{i} \alpha_j\right)}$$

It should be noted that W_i is the average *queueing time* only. The total time a customer spends at node 1 is the sum of the queueing time *and* the service time.

The only difficult part of deriving W_i is determining the values of each of the α_i's. To do this, let p(s) equal the steady state probability of being in state s of the Markov chain. Let S_{ij} be the set of states of the Markov chain representing j clusters at node i. The rest of the clusters, if any, may be at any of the remaining nodes. Then,

$\alpha_i = \frac{r_i}{\sum_{j=1}^{n} r_j}$ where $r_i = \mu_i \sum_{j=1}^{m} \sum_{s \in S_{i+2,j}} j \cdot p(s)$.

2.6. Description of Experimental System: FTMP

FTMP is a highly reliable multiprocessor installed in the AIRLAB at NASA Langley Research Center. This machine is intended to be used for real-time control of commercial aircraft of the next decade. Because of the disastrous effects that could occur if this computer should fail while in use, NASA has determined that the probability that

this system could fail should be less than 10^{-9} for a 10 hour flight. This obviously calls for extremely rigid performance criteria.

The hardware structure of FTMP consists of ten identical Line Replaceable Units (LRU's)[17]. Each LRU includes a processor module which contains local cache memory, a shared 16K word memory, a 1553 I/O port, two bus guardian units, a clock generator, and a power subsystem. Any three processors can be grouped together into a triad. The processor remaining after forming three triads is reserved as a spare processor. Ten memory modules are also formed into three triads and a spare. Communications between processors and the shared memory are accomplished through serial system buses: that is, a data transmit bus(T-bus), a data receive bus(R-bus), and a polling bus (P-bus) for resolving bus contention. The system buses are also arranged as triads by activating three out of five. Therefore, from the programmer's viewpoint, there is only one system bus.

System configurations are controlled by bus guardians which assign the connections between processors and the P-bus or T-bus, and between shared memory and the R-bus. Two bus guardians at each LRU form a dyad such that any transmission to system buses will be enabled only when both guardians agree. The bus guardians are also used as a *voter* for any processor or memory triad. Since three processors in one triad are operating in tight synchrony, their respective bus guardians should receive three identical data under a fault-free condition. When there is a disagreement, an error is considered to have occurred, but masked, and the task execution will continue. Meanwhile, the disagreement will be recorded at an error latch for later identification of the faulty module or bus. From the user's or software's standpoint, the FTMP is regarded as a three processor system and has a shared 48K system memory among the three as shown

in Figure 4. The interested reader is referred to [17] for a complete architectural description of FTMP. What has been stated here is sufficient for the present discussion.

The software of FTMP is divided into five groups. They are the Executive Software, Facilities Software, Acceptance Test/Diagnostic Software, Applications Software, and Support Software [18]. Most of the tasks in each of these software groups have to be dispatched at regular intervals to handle repetitive applications such as flight control, configuration control, fault detection, recovery, as well as system displays. To do this, FTMP has a dispatch algorithm that initiates tasks at their required frequencies. Taking into account the type of application, the FTMP developers determined that tasks had to be executed at three different frequencies, and the type of action performed by the task determined which rate group the task belonged in. They termed the three rate groups R1, R3, and R4. Their respective nominal frequencies are 3.125, 12.5, and 25 Hz. Tasks required to execute at a particular frequency are given priority to access system components over tasks that are initiated at lower frequencies. This implies that tasks in the R4 rate group have priority for bus access over tasks from rate group R3, etc.

Fault detection, identification, and system reconfiguration are handled by an executive program called the *System Configuration Controller* (SCC) which is dispatched at the slowest rate R1. This is done so the execution of the SCC will have a minimal effect on the system workload, and the errors generated by a single fault will have an appropriate system response. For experimental purposes, there are two application tasks installed on the FTMP: auto-pilot and display programs.

The associated fault injection system is controlled by a host VAX-11/750 computer. The injection extenders can be inserted into any chips at LRU3 and their respective

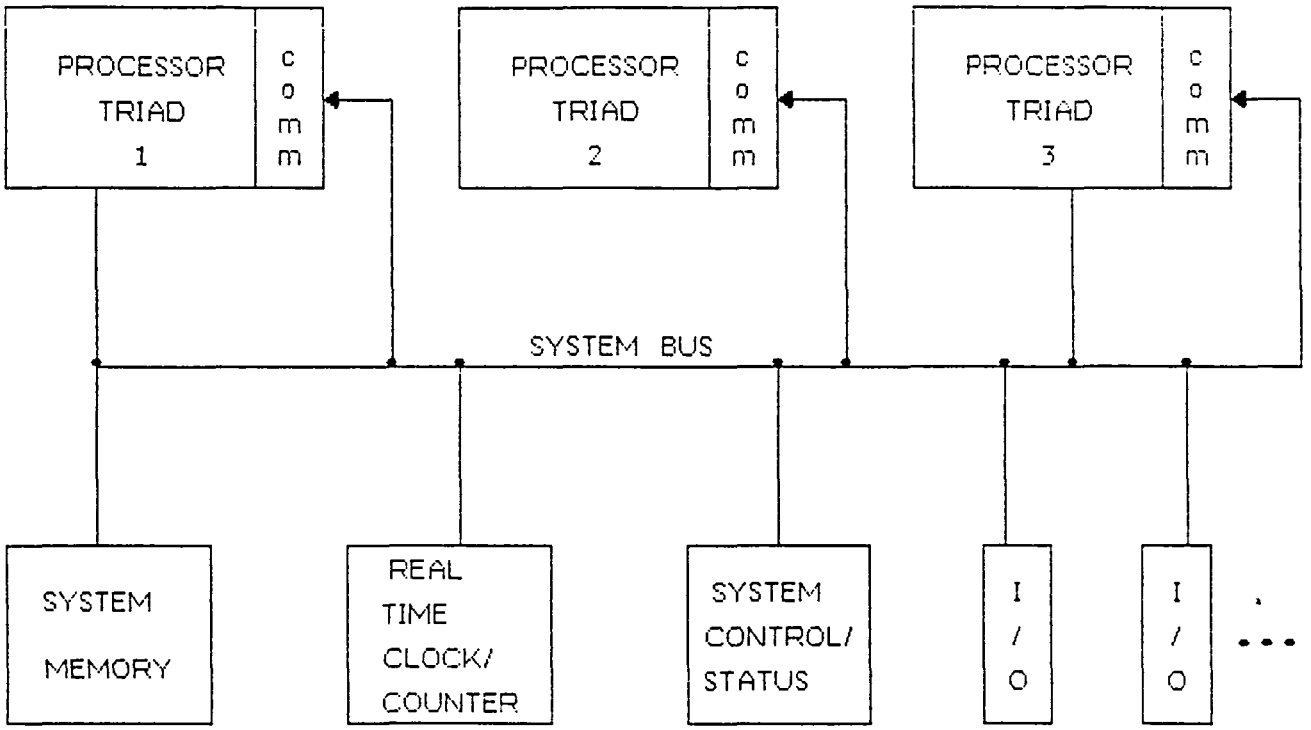

Figure 4. A block diagram of FTMP (from [15]).

socket holes such that the electrical connection between pins and the circuit board becomes controllable. Thus, three types of faulty signals, i.e., *inverted signal, stuck-at-1,* and *stuck-at-0,* can be injected at the pin level. Before any injection, the host computer will signal the FTMP to activate LRU3 for the fault injection. The detection, identification and reconfiguration intervals are measured by reading a real-time clock and the responses from the FTMP. This information is then transferred from the FTMP to the VAX-11/750 via a 1553 I/O port and a communication interface. Fault injection operations are processed by the FIS (Fault Injection System) on the VAX-11/750. The FIS consists of a command interpreter, an injection handler, and an FTMP-VAX interface program.

Recently, we have conducted some experiments on FTMP to measure some factors relating to bus contention, and the polling sequence. The results are summarized in Table 3. These results pertain to the fault free system with three operating triads. As can be seen, with the software presently on the system, there is a large amount of bus contention. Although a triad usually succeeds in its first poll sequence, it must wait 47% of the time for the bus to become free. However, it was noticed in performing the measurements that the bus was usually busy for only a very short period. The busy period was of a significant duration in only a few instances. It is also interesting to note that the duration of a bus transaction is one quarter the time between bus requests. This is probably why the bus is busy so often when a bus request is made.

2.7. Queueing Model Representation of FTMP

It is obvious that the architecture and software structure of FTMP fit nicely into our queueing model. One can represent the three triads as clusters, and each of the rate groups as a job class. Job class 1 is rate group R4, because of the relative priorities of

P(Bus is busy when a bus request is made)	= 0.47
P(Bus is free when a bus request is made)	= 0.53
P(Succeed in first poll sequence)	= 0.92
P(Lose first poll sequence)	= 0.08
P(Succeed in second poll sequence)	= 1.00
Ave. idle time waiting for free bus, if lost poll sequence	= 32.2 μs
Ave. idle time waiting for free bus, if busy when request was made	= 21.0 μs
Ave. duration of bus transaction	= 36.4 μs
Ave. time between bus requests	= 140.9 μs

Table 3. Experimental Measurements

the rate groups and job classes. Likewise, job class 2 is R3, and job class 3 is R1. There is some dependence when tasks from a rate group are executed based on the state of tasks of a higher priority rate group. However, these can be handled by the model by increasing the number of job classes. For the purpose of illustration, these dependencies are assumed to be negligible. In the queueing model representation of FTMP we, therefore, have five nodes and three tokens representing clusters, i.e., n=3 and m=3. By the formula mentioned earlier, the Markov chain representation of this specific model has $J = \binom{7}{4} = 35$ states. These states and their respective reduced system states are described in Table 4.

To solve the Markov chain, the values for the parameters of the queueing model have to be determined. Sample values are outlined in Table 5. The value for μ_S was obtained from the experimental data. P_I was arrived at from the documentation on FTMP[18]. The other parameters were arrived at through reasonable assumptions, or realistic relations among the service rates. The computed steady state probabilities for the states of the Markov chain using these parameter values is shown in column 3 of Table 4. Columns 4, 5, and 6 of Table 4 are the steady state probabilities when the parameter μ_S is varied, and the rest of the parameters remain constant.

Using the information supplied by Table 4, some simple results can be stated. The probability that there is an idle cluster is the sum of the steady state probabilities for the Markov states where there are one or more clusters at node 2. These are states 2, 6, 7, 8, 9, and 16 thru 25. The idle probabilities for the different values of μ_S are shown in Table 6. These numbers are extremely low, implying that rarely is a triad idle. The probability that there is bus contention is the sum of the steady state probabilities of states 1 thru 5 (states representing more than one cluster at node 1). These results are

Markov States		Computed Steady State Prob.			
State	Reduced System State	μ_S =0.0275	μ_S =0.00275	μ_S =0.0138	μ_S =0.055
1	(3, 0, 0, 0, 0)	0.022	0.589	0.096	0.004
2	(2, 1, 0, 0, 0)	0	0.002	0.001	0
3	(2, 0, 1, 0, 0)	0.039	0.106	0.087	0.013
4	(2, 0, 0, 1, 0)	0.039	0.106	0.087	0.013
5	(2, 0, 0, 0, 1)	0.037	0.099	0.081	0.013
6	(1, 2, 0, 0, 0)	0	0	0	0
7	(1, 1, 1, 0, 0)	0.001	0	0.001	0.001
8	(1, 1, 0, 1, 0)	0.001	0	0.001	0.001
9	(1, 1, 0, 0, 1)	0.001	0	0.001	0.001
10	(1, 0, 2, 0, 0)	0.036	0.010	0.039	0.024
11	(1, 0, 1, 1, 0)	0.071	0.019	0.079	0.048
12	(1, 0, 1, 0, 1)	0.067	0.018	0.074	0.045
13	(1, 0, 0, 2, 0)	0.036	0.010	0.039	0.024
14	(1, 0, 0, 1, 1)	0.068	0.018	0.073	0.045
15	(1, 0, 0, 0, 2)	0.031	0.008	0.034	0.021
16	(0, 3, 0, 0, 0)	0	0	0	0
17	(0, 2, 1, 0, 0)	0	0	0	0
18	(0, 2, 0, 1, 0)	0	0	0	0
19	(0, 2, 0, 0, 1)	0	0	0	0
20	(0, 1, 2, 0, 0)	0.001	0	0.001	0.001
21	(0, 1, 1, 1, 0)	0.002	0	0.001	0.003
22	(0, 1, 1, 0, 1)	0.002	0	0.001	0.003
23	(0, 1, 0, 2, 0)	0.001	0	0.001	0.001
24	(0, 1, 0, 1, 1)	0.002	0	0.001	0.003
25	(0, 1, 0, 0, 2)	0.001	0	0.001	0.001
26	(0, 0, 3, 0, 0)	0.021	0.001	0.012	0.029
27	(0, 0, 2, 1, 0)	0.064	0.002	0.035	0.087
28	(0, 0, 2, 0, 1)	0.060	0.002	0.033	0.081
29	(0, 0, 1, 2, 0)	0.064	0.002	0.035	0.087
30	(0, 0, 1, 1, 1)	0.120	0.003	0.066	0.163
31	(0, 0, 1, 0, 2)	0.056	0.002	0.031	0.076
32	(0, 0, 0, 3, 0)	0.021	0.001	0.012	0.029
33	(0, 0, 0, 2, 1)	0.060	0.002	0.033	0.081
34	(0, 0, 0, 1, 2)	0.056	0.002	0.031	0.076
35	(0, 0, 0, 0, 3)	0.018	0	0.010	0.024

Table 4. Markov State Descriptions and Steady State Probabilities

μ_S	$= \frac{1}{36.35}$	$= 0.0275$	P_P	$= 0.95$
μ_I	$= 5 \cdot \mu_1$	$= 0.0458$	P_I	$= 0.05$
μ_1	$= \frac{1}{3} \cdot \mu_S$	$= 9.17 \times 10^{-3}$	P_1	$= 0.6$
μ_2	$= \frac{1}{6} \cdot \mu_S$	$= 4.58 \times 10^{-3}$	P_2	$= 0.3$
μ_3	$= \frac{1}{16.87} \cdot \mu_S$	$= 1.63 \times 10^{-3}$	P_3	$= 0.1$

Table 5. Parameter Values

μ_S	Idle Prob.	Cont. Prob.
0.00275	0.002	0.902
0.0138	0.010	0.352
0.0275	0.012	0.139
0.055	0.015	0.043

Table 6. Idle Processors and Bus Contention Probabilities

also shown in Table 6. Figure 5 shows the effect of changing the service rate at node 1 on bus contention. The probability of bus contention increases dramatically as the service rate approaches zero, as expected. Figures 6 - 8 show the effects of varying μ_1, μ_I, and P_I, respectively. One could derive from these graphs the sensitivity to a change in performance caused by a change in a service rate or branch probability.

There are numerous other conclusions that could be drawn from the results of this example. The sensitivities of varying other parameters, average queueing times, and degraded system performance are just a few. It can be seen that this model is useful in analyzing many of the aspects that are vital to any performance evaluation. It is important to note that *all* the parameters of the queueing model are ones that can be measured.

3. MEASUREMENT OF FAULT LATENCY

3.1. Introduction

A hardware *fault* is defined as an incorrect state caused by the physical change in a component, whereas an *error* is defined to be the erroneous information/data resulting from the manifestation of a fault. Even after a hardware fault occurs in a computer system, the system will remain error-free until the fault manifests itself. Before its manifestation, the fault is *latent* and is not harmful to any system operations. Thus, there are two time intervals of interest between fault occurrence and error detection: *fault latency* and *error latency* (see [21] for a detailed description of these). Obviously, error latency depends on the detection mechanisms[5] used. Fault latency is dependent on the location and the type of the fault, and the degree of usage of the faulty unit. In other words,

[5]which we termed the function-level detection mechanisms in [21].

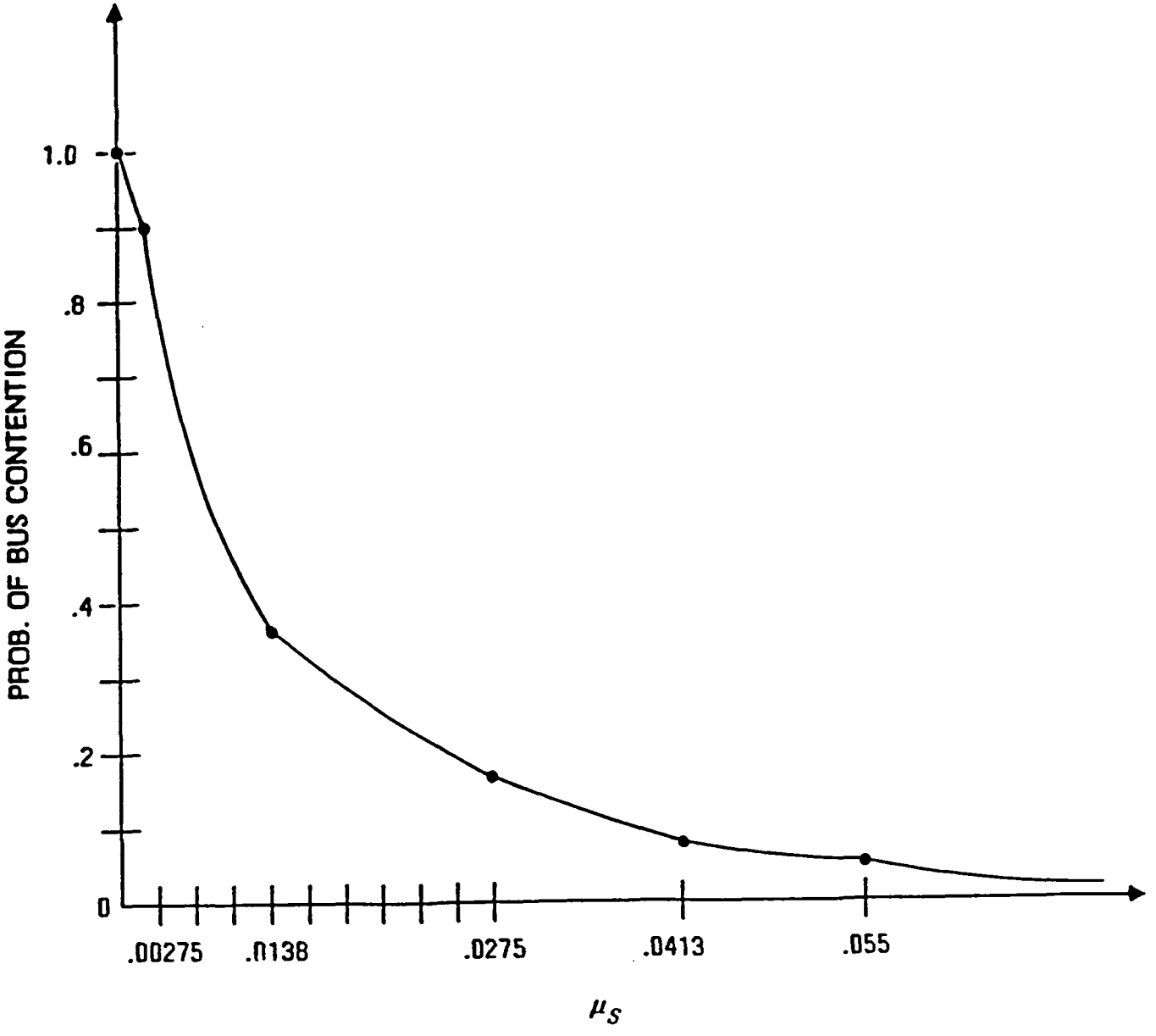

Figure 5. Probability of Bus Contention VS. Bus Service Rate (μ_S)

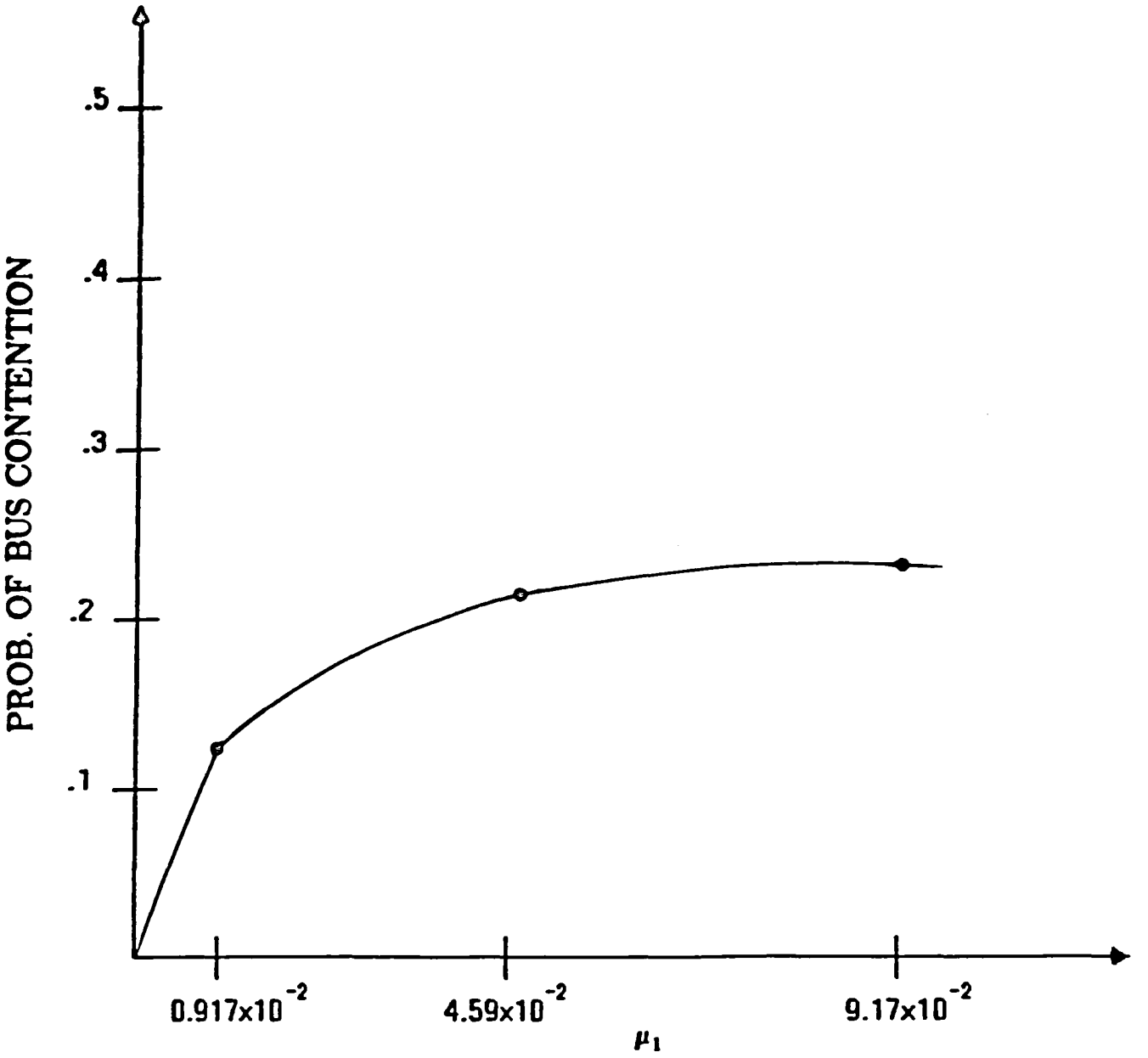

Figure 6. Probability of Bus Contention VS. Service Rate of Job Class 1 (μ_1)

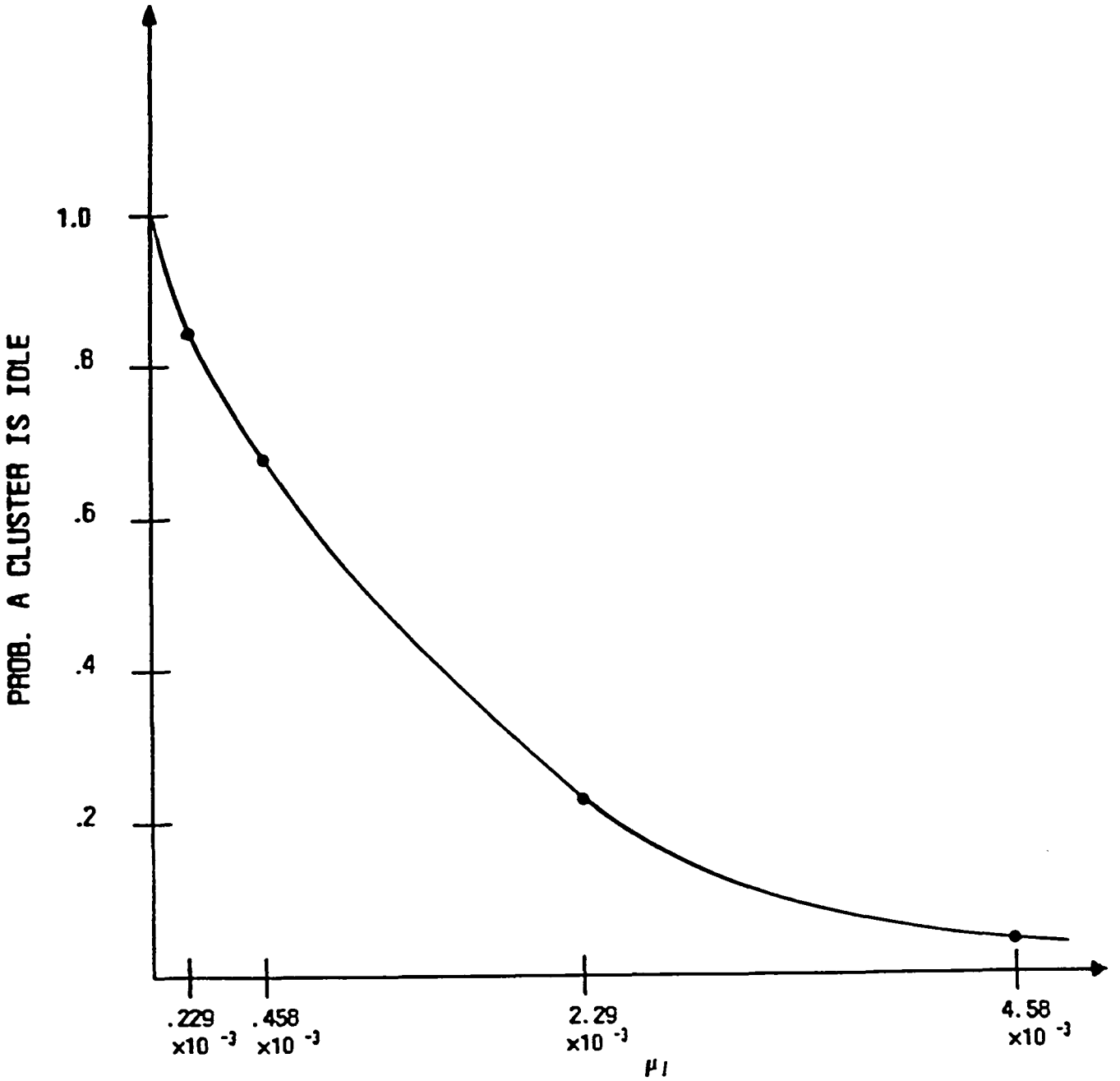

Figure 7. Probability of an Idle Cluster VS. Idle Service Rate (μ_I)

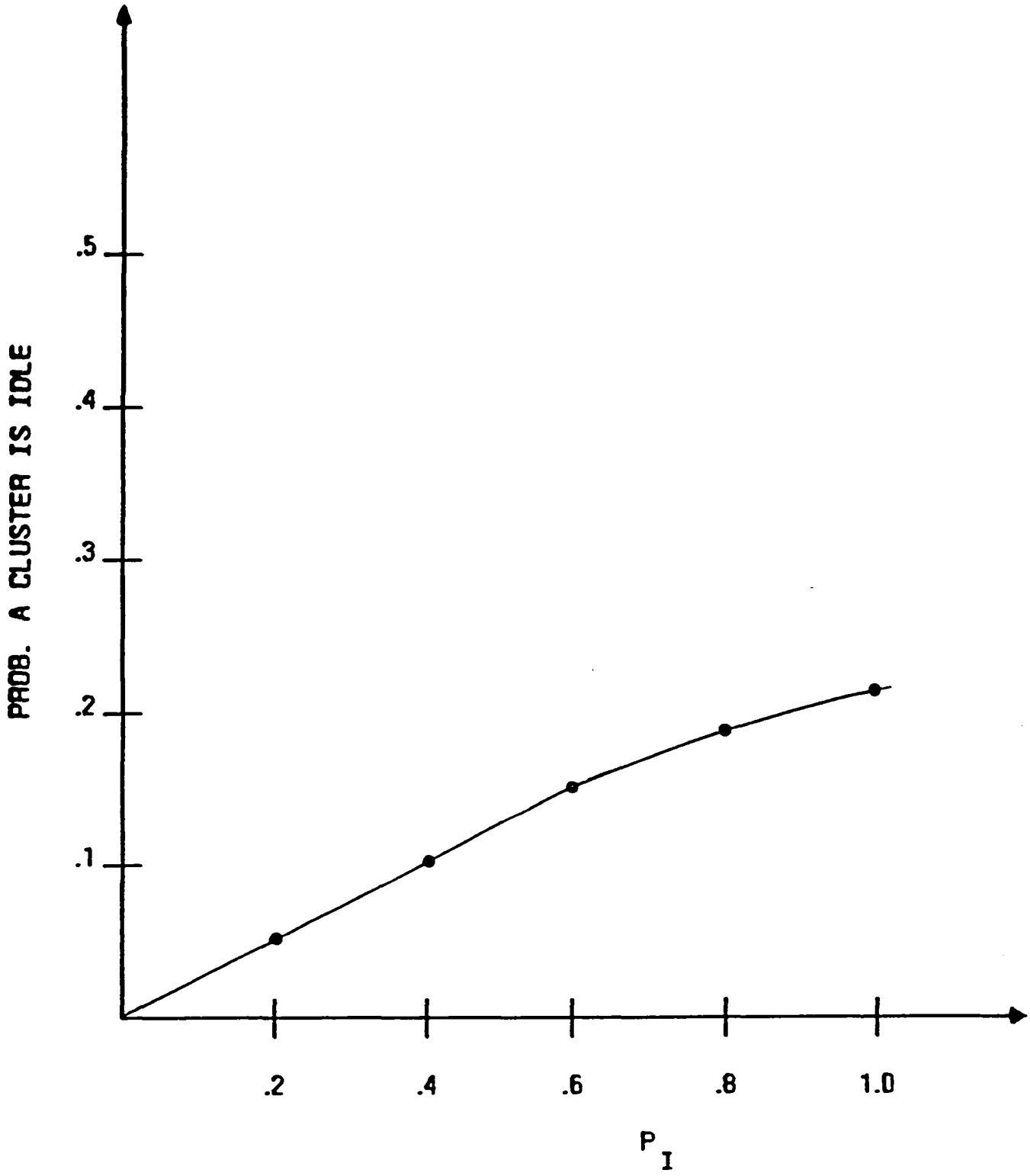

Figure 8. Probability of an Idle Cluster VS. P_I

fault latency is closely related to the physical property of a fault, whereas error latency represents the efficiency of the detection mechanisms used.

In a reliable computer system, the detection and isolation of faults and errors, and the subsequent reconfiguration are provided to tolerate faults and errors. These steps must be executed correctly by fault-free subsystems. In the face of multiple faults, the fault-tolerance capability is reduced and the coverage of failure is incomplete. It has been shown that an incomplete coverage is the major threat to a highly reliable system [22-24]. Thus, the accumulation of latent faults and the near-coincident occurrence of faults should be considered in the modeling and verification of a reliable system. However, the conventional modeling of a reliable system usually assumes that the system is recovered from an extant fault if no new fault occurs during the recovery period; otherwise, a coverage failure results. This is true only when there is no fault latency or a negligible fault latency during which no new fault occurs. That is, the conventional works have ignored the possibility of the accumulation of latent faults. Obviously, the conventional approach becomes invalid if fault latency has the same order of magnitude as the recovery period. Due to the reasons discussed above, it is essential to accurately evaluate both fault and error latencies.

In addition to the analysis of the coverage failure, the knowledge of fault latency is important to the study of transient faults. Clearly, a transient fault manifests itself only when its active duration is greater than fault latency. If fault latency is long, it is possible that most transient faults will disappear before they harm the system. In such a case, the transient faults captured by some detection mechanisms cannot represent the true characteristics of all transient faults.

In the past, several researchers conducted experiments and simulations to investigate faults' manifestations and subsequent error detections by injecting hardware faults [25-34]. Results were observed through the detection mechanisms following the fault injections. They measured the *probability of detection* and the *distribution of detection times* which are the sum of fault and error latencies. Since there does not exist a direct way to determine the moment of error generation, these experiments fail to indicate the moment of error generation which divides the detection time into fault latency and error latency. Instead, a combined effect of the inherent fault property and an associated detection operation can be observed via these experiments. Thus, these experiments neither help us understand the behavior of fault and error generation, nor give an accurate measure of the capabilities of detection mechanisms. In order to remove this inadequacy, we develop here a methodology to measure fault latency; with the measured fault latency and detection time, error latency can also be computed.

3.2. Methodology for Measurement of Fault Latency

Suppose there are some detection mechanisms which are able to detect the error generated by a fault f. Let t_f represent the fault latency of this specific fault, which is a random variable with the distribution function $F_f(t)$. We inject the fault f n_i times, and each injection is held active for the duration t_i. If t_f is greater than t_i, then no error will be generated. Otherwise, the fault manifests itself, inducing an error which will be captured later by the detection mechanisms. If there are d_i detections among these n_i injections, then the ratio $\frac{d_i}{n_i}$ indicates the probability that an error is generated during the fault active duration t_i. This is equivalent to the probability that the fault latency is smaller than t_i. Thus, we obtain the distribution function of fault

latency for the fault f as follows:

$$F_f\ (t_i) = Prob\ (t_f \leq t_i) = \frac{d_i}{n_i} \tag{1}$$

Notice that this measurement of fault latency is not affected by error latency. This also implies that the result of measurement is independent of the efficiency of detection mechanisms. Thus, as long as the error induced by the fault f can be detected, we can obtain the distribution of fault latency for the fault f .

We cannot overemphasize the fact that the moment of error generation is not directly observable. Although the occurrence of a logic failure caused by a fault can be identified by voting, the logic failure does not always induce an error at the function level. In other words, there may not exist a sensitized path such that the faulty signal can propagate to the output stage. Consequently, we have proposed a new methodology to indirectly measure the fault latency. Due to the "indirect" nature of our measurement, we obtain the distribution of fault latency instead of actual samples of fault latency. Clearly, this fact does not allow for any rigorous statistical analysis of our experimental data. However, to our best knowledge, the proposed indirect methodology is the first and the only attempt to measure fault latency.

3.3. Experimental Results and Analysis on FTMP

For our experiments, the original FIS (Fault Injection System) has been modified to enable us to inject transient faults.[6] Additional features are added to the command interpreter such that the active duration of a transient fault can be specified and passed to the injection handler. Injection ends if either the response of the FTMP indicates the accomplishment of detection, identification and reconfiguration, or the active duration

[6]The original FIS is designed for injecting permanent faults only.

becomes larger than the specified value. In the latter case, FIS is made to wait a few seconds for a possible response from the FTMP.

To measure fault latency and demonstrate the methodology proposed above, transient faults were injected to four circuit boards of the FTMP, i.e. CPU Data Path, CPU Control Path, Cache Controller, and System Bus Controller. The first three boards are in the CAP6 processor/cache region which is constructed with the AMD 2900 series bit-slice microprocessors. The System Bus Controller is responsible for transferring blocks of words between a local processor region and the shared memory. It also serves as a synchronizing mechanism such that the processors in a triad can be brought into full synchrony. On each board, several pins are selected for injecting transient faults. Selection of boards and pins is made arbitrarily. For each pin, stuck-at-0, stuck-at-1, and inverted signals are injected.

A *prime test* was applied to each selected pin to observe whether or not an error is generated after the injection of a permanent fault (which has an active duration of 3 seconds or more). In Wimmergren's experiments on the FTMP [33], undetected faults are reported to exist. Possible explanation for the existence of undetected faults are: (1) the circuits are not exercised, (2) there are "don't care" or redundant pins, and (3) the injected fault does not cause any logic failure. In our experiments, injection of transient faults is not made if there is no detection during the prime test. At certain pins, errors are detected when stuck-at-0 and inverted signal faults are injected, but not stuck-at-1 faults. In such a case, injection of stuck-at-1 faults is omitted.[7]

For each pin, transient faults with different active durations are injected 10 to 40 times repeatedly. At an early experiment, we found that d_i/n_i increases sharply when

[7]Obviously, there is no use of such an injection.

the transient durations are small. Thus, to have good resolution, the active duration of the transient faults injected, denoted by t_i, are not equally distanced. That is, we used a finer resolution for small t_i's and a coarser resolution for large t_i's. Moreover, since the fault latency at the System Bus Controller board is much larger than that at the other boards, t_i's used for testing this board are different from those used for the others.

Among more than 20,000 transient faults injected, only 15,111 results are used for the analysis. The other data are regarded unreliable because: (1) the fault identified by the FTMP was not in the LRU where the fault was actually injected, (2) the FTMP crashed during the fault injection, and (3) one of the detection, identification and reconfiguration times was negative. If the second case occurred, the injection was performed again. For every i and each type of fault at a pin, using the measured d_i/n_i, we obtained the averaged $d_i/n_i = F_f(t_i)$ for each board, which are listed in Table 7. In addition, we present $h_f(t_i)$ in the table which is defined as

$$h_f(t_i) = \frac{F_f(t_{i+1}) - F_f(t_i)}{(t_{i+1} - t_i)(1 - F_f(t_i))} \tag{2}$$

The function $h_f(t_i)$ becomes the hazard rate of fault latency as $t_{i+1} - t_i \to 0$. Despite the fact that negative numbers appeared twice in Table 7, the functions $h_f(t_i)$ in the table strongly suggest that the hazard rate of fault latency be *monotone decreasing*. Thus, two distributions with monotone decreasing hazard rates, i.e., Weibull and Gamma distributions, are used to fit the experimental results. Estimated parameters are given in Table 7 where the least-squares errors are also included. The experimental results and the estimated Weibull distribution are plotted in Figures 9 through 12.

The estimated parameter for exponential distributions is also presented in Table 8 for the purpose of comparison with Weibull and Gamma distributions. It can be seen

$t_i(ms)$	s-a-0		s-a-1		inverted	
	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$
0.0	0.0	21.0	0.0	9.0	0.0	35.0
0.01	0.21	1.27	0.09	0.36	0.35	3.59
0.10	0.30	0.071	0.12	0.20	0.56	0.40
0.50	0.32	0.23	0.19	0.074	0.63	0.11
1.00	0.40	0.079	0.22	0.054	0.65	0.078
5.00	0.59	0.073	0.39	0.049	0.76	0.025
10.00	0.74	0.058	0.54	0.028	0.79	0.043
20.00	0.89	-	0.67	-	0.88	-

(a). Experimental Results and $h(t_i)$ on Cache Controller.

$t_i(ms)$	s-a-0		s-a-1		inverted	
	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$
0.0	0.0	67.0	0.0	85.0	-	-
0.01	0.67	9.09	0.85	9.63	-	-
0.10	0.94	1.67	0.98	-3.75	-	-
0.50	0.98	2.00	0.95	1.20	-	-
1.00	1.00	-	0.98	0.11	-	-
10.00	0.98	0.01	1.00	-	-	-
20.00	1.00	-	1.00	-	-	-

(b). Experimental Results and $h(t_i)$ on CPU Control Path.

Table 7. Experimental Results and Estimated $h(t_i)$.

$t_i(ms)$	s-a-0		s-a-1		inverted	
	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$
0.0	0.0	25.0	0.0	34.0	0.0	49.0
0.01	0.25	5.67	0.34	2.65	0.49	7.35
0.05	0.42	1.72	0.41	1.69	0.64	11.7
0.10	0.47	0.566	0.46	0.42	0.85	0.83
0.50	0.59	0.097	0.55	0.44	0.90	0.40
1.00	0.61	0.038	0.65	0.086	0.91	0.28
5.00	0.67	0.030	0.77	0.052	0.92	0.0
10.00	0.72	0.025	0.83	0.041	0.92	0.0125
20.00	0.79	-	0.90	-	0.93	-

(c). Experimental Results and $h(t_i)$ on CPU Data Path.

$t_i(ms)$	s-a-0		s-a-1		inverted	
	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$	$F(t_i)$	$h(t_i)$
0.0	0.0	0.040	0.0	0.032	0.0	0.036
5.0	0.20	-.013	0.16	0.050	0.35	0.036
10.0	0.15	0.0106	0.37	0.0079	0.56	0.021
20.0	0.24	0.0026	0.42	0.0138	0.63	0.016
50.0	0.30	0.0037	0.66	0.0011	0.65	0.0074
100.0	0.43	0.0068	0.68	0.0056	0.76	0.0053
200.0	0.82	0.010	0.86	0.01	0.79	0.001
300.0	1.00	-	1.00	-	0.88	-

(d). Experimental Results and $h(t_i)$ on System Bus Controller.

Table 7. Experimental Results and Estimated $h(t_i)$ (cont'd).

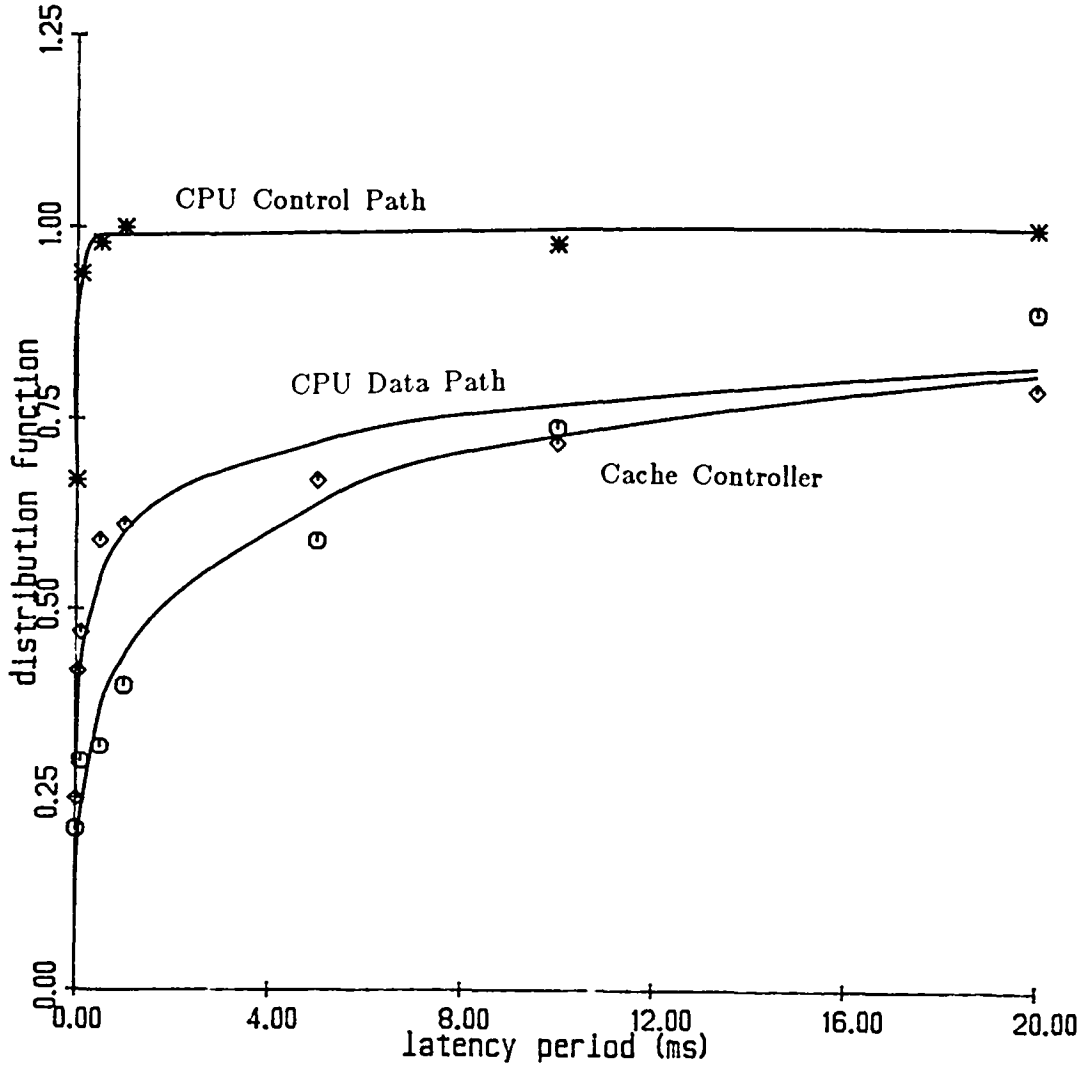

Figure 9. The Experimental Results and Estimated Distributions for Stuck-at-0 Faults.

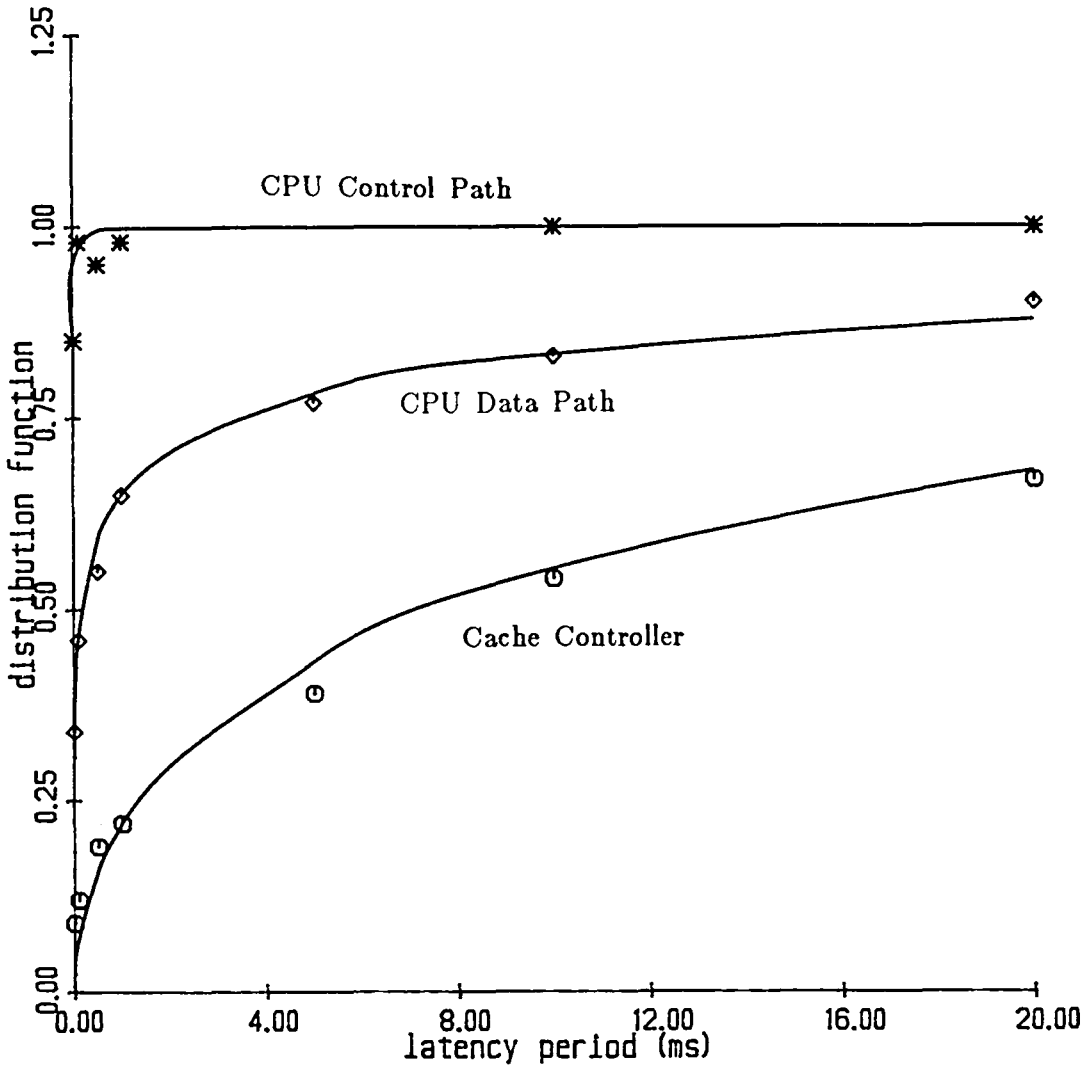

Figure 10. The Experimental Results and Estimated Distributions for Stuck-at-1 Faults.

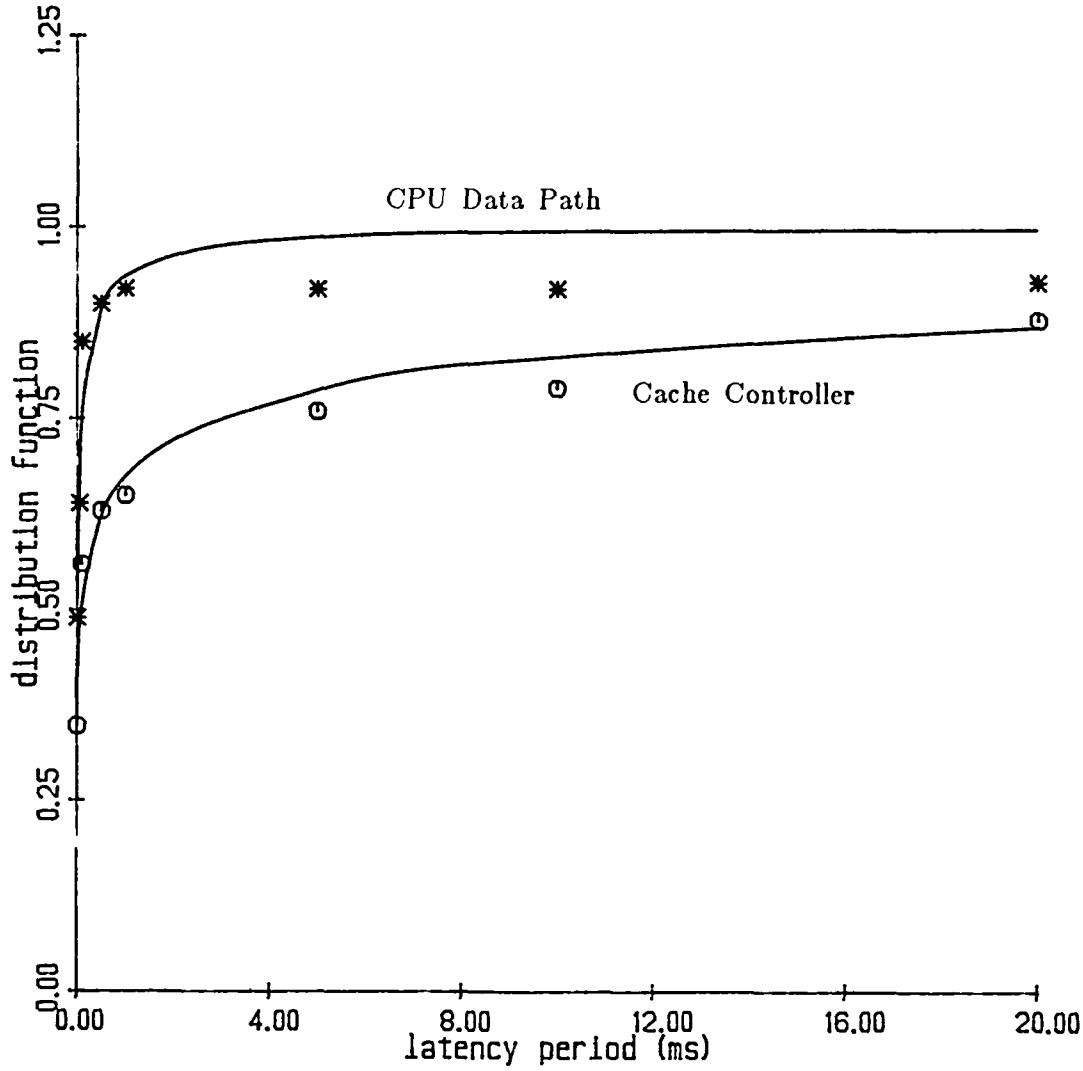

Figure 11. The Experimental Results and Estimated Distributions for Inverted Signal Faults.

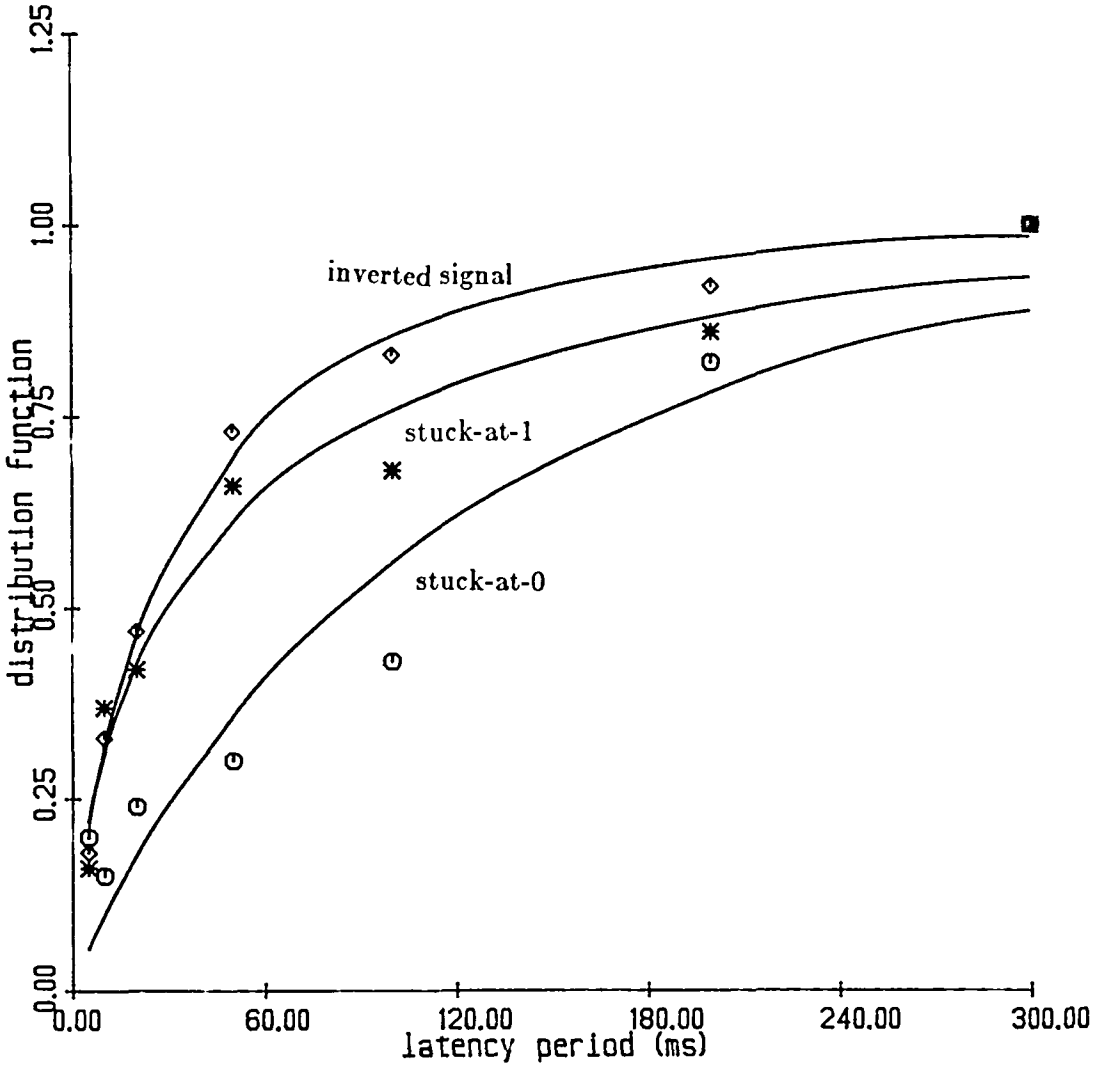

Figure 12. The Experimental Results and Estimated Distributions of Fault Latencies at System Bus Controller.

		Exponential		Weibull			Gamma		
		$1/\lambda$	error	$1/\lambda$	α	error	$1/\lambda$	α	error
CC	s-a-0	4.78	0.24	4.35	0.35	0.03	45.89	0.24	0.02
	s-a-1	13.07	0.08	15.24	0.51	0.015	61.61	0.38	0.006
	inverted	0.46	0.41	0.56	0.20	0.009	82.90	0.11	0.007
CPUC	s-a-0	0.009	0.004	0.0076	0.39	0.0006	0.117	0.19	0.0008
	s-a-1	0.005	0.003	0.001	0.27	0.0025	0.092	0.09	0.0029
CPUD	s-a-0	0.515	0.539	1.488	0.21	0.021	153.9	0.12	0.018
	s-a-1	0.628	0.31	0.799	0.23	0.006	56.79	0.13	0.0013
	inverted	0.036	0.115	0.030	0.29	0.0026	0.648	0.18	0.032
SBC	s-a-0	125.2	0.063	124.9	0.89	0.061	173.2	0.77	0.057
	s-a-1	46.9	0.097	54.85	0.58	0.020	176.18	0.44	0.021
	inverted	34.4	0.029	39.10	0.70	0.0045	80.44	0.58	0.0066

CC -- Cache Controller, CPUC -- CPU Control Path

CPUD -- CPU Data Path, SBC -- System Bus Controller

Table 8. Least-Squares Estimation of the Distributions of Fault Latencies.

that the constant error generation rate (i.e. exponential distribution) does not model the error generation well. The mean fault latencies -- which are $1/\lambda$ in the estimated parameter of the exponential distribution -- range from 0.0005ms of stuck-at-1 faults in the CPU Control Path to 125ms of stuck-at-0 faults in the System Bus Controller. This was due to the different exercise rates at each board. Since each injected stuck-at-0 or stuck-at-1 fault does not always represent a logic failure at the moment of injection, the fault with an inverted signal should have a shorter fault latency: this is confirmed by the experimental results.

As pointed out earlier, fault latency is not directly observable. This fact has led us to the development of a new methodology which allows for indirect measurement of fault latency. Note, however, that our experimental results give the distribution function of fault latency instead of data samples of fault latency. Hence, statistical analyses or hypotheses testing are not applicable to these experimental data. The least-squares estimation with commonly used distributions gives only approximate values of the parameters. They cannot test whether an underlying model is (statistically) good or bad. Indeed, from the least-squares errors in Table 8 it is unclear which distribution has the best fit. However, since the hazard rate converges to $1/\lambda$ and 0 for Gamma and Weibull distributions, respectively, it is possible to distinguish between them once additional injections with larger active durations are performed.

4. CONCLUSION AND DISCUSSION

In this report, we have presented first a model to be used to study the workload effects on performance for a highly reliable unibus multiprocessor used in critical real-time applications. Because of the strict performance criteria required for systems of this type, a detailed analysis is both desirable and necessary.

The operation of the computing system addressed has been illustrated using a modified Stochastic Petri Net(SPN). It was the purpose of this model to graphically describe the synchronous operation of multiple processing clusters. It was desired to show which aspects of the computer's operation have the most significant effect on the computer's performance. Most certainly, system bus contention, workload distribution, and idle processing periods have a marked effect on performance.

The modified SPN was useful for the purpose of describing computer activity. However, as a tool for performance evaluation, it was shown to be too complex for worthy analysis. A simpler model has been presented that still describes the critical performance related facets. This model is a closed queueing network consisting of multiserver nodes and a single non-preemptive priority queue.

The queueing model was shown to be easily solved for a set of given parameters. It was also observed that useful results pertaining to system performance could be directly obtained from the solution to the queueing model. The ease of obtaining these results and the overall importance of the results demonstrate the usefulness of the model for the purpose of performance evaluation.

The area that merits further research is in determining the distribution of the workload among different job classes. A systematic method has not been developed yet to construct the various job classes from the workload of a real-time control system. Characterization of real-time workloads is a more restricted problem than dealing with the workloads of a general purpose computer. This motivates continued research in solving the workload distribution problem. Once a characterization method is developed, one can then consider the possibility of obtaining an optimal workload distribution to provide optimal performance.

We have also developed a new methodology for indirectly measuring fault latency with the injection of faults. The methodology has been realized by experiments on the FTMP. The FTMP experimental results show a large variation in fault latencies for different circuits. It has also been observed that the hazard rate of fault latency is monotone decreasing. This implies that a fault tends to be latent if it did not generate an error at its early stage. The existence of a long fault latency should not be ignored in highly reliable systems. To reduce the accumulation of latent faults, additional on-line diagnostics must be incorporated into the area where a long fault latency exists.[8]

Although two possible distributions are used to fit the experimental results, no underlying model for fault latency can be concluded. It is mainly because of the unobservability of error generation. More experiments should be designed to investigate the behavior of a fault and its effect on system execution. An immediate extension of our experiments is to make the injections under different system workloads or the execution of different application tasks. We expect to see some variations of fault latency in certain circuits.

During the FTMP experiments, some interesting points were observed, especially when the faults were injected into the System Bus Control. At certain pins, identification results were different for various active durations of injections. For instance, with a long (in relative to fault latency) active duration, the SCC indicated that the whole LRU was faulty, but indicated that only a processor or memory was faulty when the active duration was short. This situation was sometimes reversed. In other words, the identification results by the SCC depend on both the location of injection and the active duration of the fault. For the injections in the other boards, e.g., Cache controller, CPU

[8]Such areas can be identified by the methodology proposed in this report.

Data and Control Path, a processor was identified as faulty. Note that the System Bus Controller is the interface between the processor region and system buses. These observations show that the errors do not propagate out of the processor boundary. They also suggest that an error easily propagates from interface circuits, but the identification of a faulty interface circuit is more difficult.

In addition, we encountered several problems that were inconsistent with the FTMP's specification. This forced us to abandon some experimental results. Specifically, fault injections to the System Bus Controller caused the FTMP to generate frequent system crashes or have wrong identifications. Certainly, the FTMP could not distinguish between the injection of a fault from the true occurrence of a fault. These abnormalities occurred too frequently to be treated as random failures. In addition, only 210 responses from the FTMP indicated that the detected faults were transient, even when faults with a 10 micro-second active duration were injected. In fact, all injections of transient faults in the Cache Controller, CPU Data and Control Path were regarded as permanent. A thorough verification is needed for the FTMP's detection and identification mechanisms. This is a matter for our future research.

ACKNOWLEDGMENT

The authors are grateful to Carlos Liceaga, Ricky W. Butler, Milt Holt, Brian Lupton, and Peter Padilla at the NASA AIRLAB for their assistance in the FTMP experiments.

REFERENCES

[1] K. S. Trivedi, "Modeling and Analysis of Fault Tolerant Systems," *CS-1984-9,* Dept. of Computer Science, Duke University, 1984.

[2] M. A. Marsan, G. Balbo, and G. Conte, "Comparative Performance Analysis of Single Bus Multiprocessor Architectures," *IEEE Trans. on Computers,* vol. C-31, pp. 1179-1191, Dec. 1982.

[3] M. A. Marsan and M. Gerla, "Markov Models for Multiple Bus Multiprocessor Systems," *IEEE Trans. on Computers,* vol. C-31, pp. 239-248, Mar. 1982.

[4] C. M. Krishna and K. G. Shin, "Performance Measures for Multiprocessor Controllers", *Performance '83,* edited by A. K. Agrawala and S. K. Tripathi, North Holland, pp. 229-250, 1983.

[5] L. J. Miller, "A Heterogeneous Multiprocessor Design and the Distributed Scheduling of its Task Group Workload," *Proc. 9th Symp. on Computer Architecture,* pp. 283-290, 1982.

[6] A. Singh and Z. Segall, "Synthetic Workload Generation for Experimentation with Multiprocessors," *Proc. 3rd Inter. Conf. on Distributed Computing Systems,* pp. 778-785, 1982.

[7] M. H. MacDougall, "Instruction-Level Program and Processor Modeling," *IEEE Computer,* vol. 17, no. 7, pp. 14-24, July 1984.

[8] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1983.

[9] D. Ferrari, "On the Foundations of Artificial Workload Design," *Proc. of 1984 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems,* pp. 8-14.

[10] M. K. Malloy, "On the Integration of Delay and Throughput Measures in Distributed Processing Models," Ph.D. dissertation, Univ. California, Los Angeles, 1981.

[11] M. A. Marsan, G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *Proc. of 1983 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems,* pp. 198-199.

[12] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

[13] L. Kleinrock, *Queueing Systems Volume 1: Theory.* New York: Wiley-Interscience, 1975.

[14] N. K. Jaiswal, *Priority Queues.* New York: Academic Press, 1968.

[15] D. R. Cox and W. L. Smith, *Queues.* New York: John Wiley & Sons, 1961.

[16] T. L. Saaty, *Elements of Queueing Theory With Applications.* New York: McGraw-Hill, 1961.

[17] T. B. Smith and J. H. Lala, "Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer Volume I: FTMP Principles of Operation," *NASA Contractor Report 166071,* May 1983.

[18] J. H. Lala and T. B. Smith, "Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer Volume II: FTMP Software," *NASA Contractor Report 166072,* May 1983.

[19] J. H. Lala and T. B. Smith, "Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer Volume III: FTMP Test and Evaluation," *NASA Contractor Report 166073,* May 1983.

[20] T. B. Smith and J. H. Lala, "Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer Volume IV: FTMP Executive Summary," *NASA Contractor Report 172286,* Feb. 1984.

[21] K. G. Shin and Y. H. Lee, "Error Detection Process - Model, Design, and Impact on Computer Performance," *IEEE Trans. on Computer,* Vol. C-33, No. 6, June 1984, pp.529-540.

[22] W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability Modeling Techniques for Self-Reparing Computer Systems," *Proc. 24th Ann. ACM Nat. Conf.,* 1969, pp. 295-309.

[23] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE,* Vol. 66, No. 10, Oct. 1978, pp. 1221-1240.

[24] K. Trivedi, R. Geist, and M. Dugan, "Modeling Imperfect Coverage in Fault-Tolerant Systems," *Proc. of the 14 Annual Int'l Symp. on Fault-Tolerant Computing,* 1984, pp. 77-82.

[25] S. J. Bavuso, et al., "Latent Fault Modeling and Measurement Methodology for Application to Digital Flight Control", *Advanced Flight Control Symposium,* USAF Academy, 1981.

[26] B. Courtois, "Some Results about the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunction", *Proc. of the 9th Annual Int'l Symp. on Fault-Tolerant Computing,* 1979, pp. 71-74.

[27] B. Courtois, "A Methodology for On-line Testing on Microprocessors", *Proc. of the 11th Annual Int'l Symp. on Fault-Tolerant Computing,* 1981, pp. 272-274.

[28] Y. K. Malaiya and S. Y. H. Su, "Reliability Measure of Hardware Redundancy Fault-Tolerant Digital Systems with Intermittent Faults", *IEEE Trans. on Computers,* Vol. C-30, No. 8, August 1981, pp. 600-604.

[29] P. Marchal and B. Courtois, "On Detecting The Hardware Failures Disrupting Programs in Microprocessors", *Proc. of 12-th Int'l Conf. on Fault-Tolerant Computing,* 1982, pp. 249-256.

[30] J. G. McGough and F. L. Swern, "Measurement of Fault Latency in a Digital Avionic Mini Processor," *NASA Contractor Report 3462,* Oct. 1981.

[31] J. G. McGough and F. L. Swern, "Measurement of Fault Latency in a Digital Avionic Mini Processor - Part II," *NASA Contractor Report 3651,* Jan. 1983.

[32] V. Tasar, "Analysis of Fault-Detection Coverage of a Self-Test Software Program", *Proc. of the 8th Annual Int'l Symp. on Fault-Tolerant Computing,* 1978, pp. 65-74.

[33] A. L. Wimmergren, "Verification of a Fault Tolerant Multi-Processor Architecture," *CSDL-T-782,* The Charles Stark Draper Lab., May 1982.

[34] J. H. Lala, "Fault Detection, Isolation and Reconfiguration in FTMP: Methods and Experimental Results," *Proc. 5th IEEE/AIAA Digital Avionics System Conf.,* Nov. 1983.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-3920		

4. Title and Subtitle	5. Report Date
Modeling and Measurement of Fault-Tolerant Multiprocessors	August 1985
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Kang G. Shin, Michael H. Woodbury, and Yann-Hang Lee	
	10. Work Unit No.
9. Performing Organization Name and Address	
University of Michigan Ann Arbor, Michigan 48109-1109	11. Contract or Grant No. NAG1-296, NAG1-492, and NGT 23-005-801
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address	
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 505-34-13-32

15. Supplementary Notes

Langley Technical Monitor: Ricky W. Butler

16. Abstract

The workload effects on computer performance are addressed first for a highly reliable unibus multiprocessor used in real-time control. As an approach to studying these effects, a modified Stochastic Petri Net (SPN) is used to describe the synchronous operation of the multiprocessor system. From this model the vital components affecting performance can be determined. However, because of the complexity in solving the modified SPN, a simpler model, i.e., a closed priority queuing network, is constructed that represents the same critical aspects. The use of this model for a specific application requires the partitioning of the workload into job classes. It is shown that the steady state solution of the queuing model directly produces useful results. The use of this model in evaluating an existing system, the Fault Tolerant Multiprocessor (FTMP) at the NASA AIRLAB, is outlined with some experimental results.

Also addressed is the technique of measuring fault latency, an important microscopic system parameter. Most related works have assumed no or a negligible fault latency and then performed approximate analyses. To eliminate this deficiency, we (i) present a new methodology for indirectly measuring fault latency, and (ii) derive the distribution of fault latency from the methodology. The proposed methodology has also been applied successfully to the measurement of fault latency for FTMP. The experimental results show wide variations in the mean fault latency of different function circuits within FTMP. Consequently, Gamma and Weibull distributions are selected for the least-squares fit as the distribution of fault latency. Based on experience from these and other experiments, we have made several remarks.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault-tolerant Real-time Performance modeling Fault latency Reliability analysis Workload	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	64	A04

N-305

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1985

National Aeronautics and
Space Administration

Washington, D.C.
20546

Official Business
Penalty for Private Use, $300

BULK RATE
POSTAGE & FEES PAID
NASA Washington, DC
Permit No. G-27

NASA

POSTMASTER: **If Undeliverable (Section 158 Postal Manual) Do Not Return**