Deterministic Time and Analytical Models of Parallel Architectures

by

Mark A. Holliday

Computer Sciences Technical Report #652

July 1986

DETERMINISTIC TIME AND ANALYTICAL MODELS OF PARALLEL ARCHITECTURES

by

MARK ARMIGER HOLLIDAY

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1986

Abstract

In parallel computer architectures many events are of constant duration. For example, a fetch of a cache block, ignoring resource contention, takes a fixed number of clock cycles. Analytical performance modeling techniques that include deterministic time have previously been proposed. However, serious restrictions on the set of allowed models are imposed in these previous techniques. In this dissertation we extend these previous modeling techniques by removing those restrictions. Those restrictions fall into two classes: those involving the construction of the state space and those involving the analysis of the state space. We propose interpretations for performance measures when those restrictions are removed. In this general case, the state space represents an arbitrary, finite state, discrete parameter Markov Chain. We also present algorithms that efficiently construct and analyze the state space in the general case.

Our technique is called Generalized Timed Petri Nets (GTPN). It has been implemented in a tool and has been used to develop models for several interesting architectures. The two most important studies examine bus-based multiprocessors. Performance degradation due to memory and bus interference in multiprocessors with a single-stage interconnection network has been frequently examined. Using the GTPN we are able to derive exact performance estimates in the important case when memory access times are constant and interrequest times are non-zero. Previously only approximate estimates and simulation results existed.

Caches are an important means for reducing memory contention in bus-based multiprocessors. Our second study is the first analytical performance comparison of the key features of protocols that maintain cache consistency when a single shared bus is assumed.

Acknowledgements

This research would not have been done without the advice and support of many people. My advisor, Mary Vernon, has been very supportive throughout the development of this thesis. Her keen insight, enthusiasm, and willingness to devote a substantial amount of her valuable time, have contributed greatly to my work.

I would like to thank the members of my committee, Mary Vernon, David Dewitt, Peter Glynn, Jim Goodman, and Udi Manber for taking the time to serve on the committee and for their comments and suggestions.

I would like to thank Bill Kalsow for his support of the document preparation software I have used in this thesis. His comments in numerous discussions have also aided my research considerably.

I would like to thank my parents, Sam and Joan, for their many years of support. I would like to thank my daughter, Julia, for bringing me so much joy. Most of all I would like to thank my wife, Mary. During many late nights and weekends she cheerfully kept our home running without me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Deterministic Times and Analytical Models

The goal of our research is to develop analytical performance models of parallel computer architectures. An important feature of computer architectures is deterministic time. Events—for example, a clock cycle or a fetch from memory—often take a constant amount of time. Little work has been done in developing analytical modeling techniques that allow representing deterministic times. Consequently, a substantial part of our work has involved developing a new modeling technique that supports deterministic times.

This new technique is called Generalized Timed Petri Nets (GTPN). As the name suggests, our technique is an extension of a group of previous techniques based on Timed Petri Nets (TPN). Timed Petri Nets are a graphical model description notation. The general approach used by these techniques is as follows. The system is described by a model formulated as a Timed Petri Net. An initial state is also specified. The modeling tool constructs the state space formed by all the states that can be reached from that initial state. The state space is then analyzed in order to determine various performance measures.

The previous TPN models had substantial limitations with respect to both constructing the state space and analyzing the state space. To simplify constructing the state space, restrictions are imposed on the connectivity of the model descriptions when viewed as a graph. These restriction prevent representing important aspects of system behavior. Restrictions are also imposed on the structure of the state space in order to simplify the analysis phase. These restrictions are especially limiting, because of the large semantic gap between the model description and the state space. A model description of 30 lines can easily have a state space of 45,000 with complex probabilities on edges and times-in-state.

A substantial portion of our work has involved removing these restrictions. The

only remaining requirement is that the model's state space be finite (in practice it is usually obvious when an error in the description causes the state space to be infinite). Removal of these restrictions involved two classes of issues: conceptual issues and computational issues. At the conceptual level we had to find reasonable meanings for the situations that are no longer prohibited. At the computational level we had to propose and implement efficient algorithms for constructing and analyzing the state space in the general case.

The state space analysis is based on Markov Chain theory. The use of Markov Chain theory was suggested in the earlier TPN models. However, even in the restricted case they allowed, a rigorous treatment of the relationship to Markov Chain theory, was lacking. We have supplied such a treatment.

Of course, the extensions to previous TPN models we have made in our GTPN are a significant research contribution only to the extent that the GTPN adds to our ability to model interesting systems. We contend that the GTPN does significantly add to our modeling abilities. That contention is based on several modeling studies that have been conducted using the GTPN. By providing a flexible, efficient modeling tool that supports deterministic time, we have been to reach some interesting new results. For example, in a general model of multiprocessor memory and bus interference in single-stage interconnection networks, we are the first to derive exact performance estimates. Previously, only approximate derivations and simulations existed. In another study we provide the first analytical comparison of a wide range of cache consistency protocols that assume a shared bus.

An important result of our studies was demonstrating that interesting systems can have "unusual" state space behavior. By "unusual", we mean two things, both of which will be defined more completely later. First, a system may have multiple long run behaviors. Second, a system may be "periodic" which loosely means that the system may indefinitely oscillate among several probability distributions across the states.

1.2 Organization of this Thesis

In this section we describe the organization of the remainder of the thesis. The remainder of this chapter reviews related work in modeling techniques. In particular, it looks at untimed Petri Nets, the general class of Petri Nets with time, and lastly, the most relevant previous TPN models. The next three chapters present and discuss the GTPN. Chapter 2 discusses the GTPN model description semantics and the algorithms used in constructing the state space. Chapter 3 discusses the analysis of the state space. The first part of this chapter discusses how Markov Chain theory can be used to analyze an arbitrary finite state space. The remainder of the chapter covers the numerical issues involved. Stochastic Petri Nets (SPNs) are an important alternative method of including time and probabilities into Petri Nets. Chapter 4 compares and contrasts the GTPN with the SPN models.

Chapters 5 through 7 present applications of the GTPN. Chapter 5 looks at two model studies. The first is of the Dining Philosophers problem. The second is modeling typical workloads on the scalar mode portion of the CRAY-1 supercomputer. Chapter 6 examines memory and bus interference in a multiprocessor with a single-stage interconnection network. Chapter 7 addresses the performance evaluation of alternative shared bus cache consistency protocols.

The last chapter, Chapter 8, concludes the thesis by summarizing our work and then discussing possible future research directions.

1.3 Related Work

In this section we describe untimed Petri Nets and previous Petri Net models with time. This provides a foundation for the presentation of the GTPN which starts in Chapter 2. Subsection 1.3.1 describes untimed Petri Nets. A more thorough introduction to untimed Petri Nets can be found in Peterson [PET81]. Subsection 1.3.2 surveys performance-oriented Petri Nets. Subsection 1.3.3 reviews the most relevant previous: the work of Zuberek, and Razouk and Phelps.

1.3.1 Untimed Petri Nets

Untimed Petri Nets (PNs) contain *places* P, *transitions* T, and *arcs* A. The arcs are directed and can only connect transitions to places and places to transitions. If an arc exists from a place to a transition, then the place is an *input place* for that transition. If an arc exists from a transition to a place, then the place is an *output place* for that transition. Places may contain *tokens*. The *state* of a PN is defined by the number of tokens in each place and is represented by a vector M called the *marking* vector. $M[i]$ is the number of tokens in the ith place.

Petri Nets are often illustrated graphically, as shown in Figure 1.1. Circles represent the places. Black dots in the circles represent the tokens. Bars represent the transitions.

The number of arcs connecting a place to a transition is that input place's *multiplicity*. A transition is *enabled* if each of its input places contains at least as many tokens as there are arcs from the place to the transition. The tokens on all input places which exactly equal the input's multiplicity are the transition's *enabling* tokens. An enabled transition can *fire*. A transition *fires* by: 1) removing all of its enabling tokens from its input places, and 2) placing on each of its output places one token for each arc from the transition to that output place. Each firing of a transition changes the assignment of tokens to places and thus creates a new state. The *reachability set* of a PN and a given initial state is the set of all states that can be reached from that initial state via a sequence of transition firings. The *reachability graph* associated with a reachability set can be constructed as follows. Represent

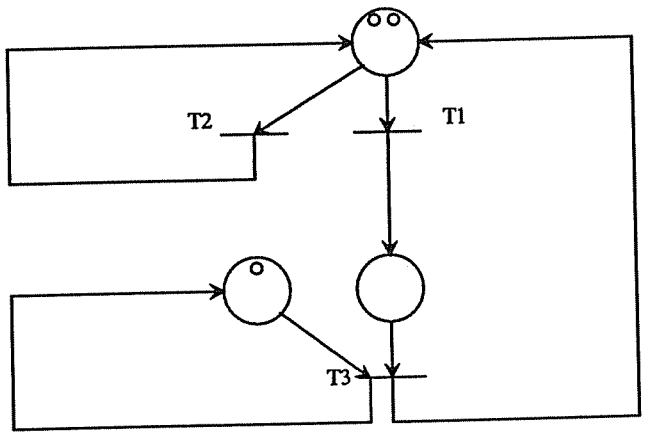

Figure 1.1: Example of an untimed Petri net

each state by a vertex and place a directed edge from vertex v_1 to vertex v_2 if the state v_2 can result from firing some transition enabled in state v_1.

1.3.2 Performance-oriented Petri Net Models

Ramachandani [RAM74] was the first Petri Net model to include time. His approach introduced a fixed firing time with each transition in a Petri net. Since 1974 many other models have been proposed. These models may be distinguished by two simple criteria. One, is by whether firing times are associated with transitions or with places. The second is by whether firing times are constants or random variables (models with random variable firing times, also sometimes allow constant firing times as a special case).

The second criteria is much more significant. The models assuming constant firing times we will refer to as Timed Petri Nets (TPNs). We will refer to the models assuming firing times are random variables as Stochastic Petri Nets (SPNs). Several of the TPNs use an algebraic approach to the analysis. Ramamoorthy and Ho [RAM80] consider the structure of the net as a graph. For a very restricted class of nets, they find the longest circuit in the graph using a matrix algorithm. Coolahan and Roussopoulos [COO83] derive lower and upper bound timing equations from the net, again by viewing the net as a graph and determining the possible paths through the graph. In contrast to the approaches by Ramamoorthy and Ho and by Coolahan and Roussopoulos, Stotts and Pratt [STO85] build the net's state space. However, they do not introduce a stochastic process interpretation.

The SPNs, models that assume firing times are random variables, on the other hand, almost invariably construct the net's state space, to which stochastic process techniques are applied. The first models assumed that all firing times are exponential random variables. These models were independently proposed by Natkin [NAT80], Symons [SYM80], and Molloy [MOL81,MOL82]. The restriction to exponential random variables allows a simple analysis because the underlying stochastic process is a

continuous-time Markov Chain. Loosening this restriction supports a more flexible semantics, but implies a more complex underlying stochastic process. The more recent models have attempted to deal with this problem. An important such model is Generalized Stochastic Petri Nets (GSPNs) by Marsan, Balbo, and Conte [AJM84]. In the GSPN instantaneous firing times as well as exponential firing times are allowed. The ESP of Cumani [CUM85] is an extension so that firing times have an arbitrary phase-type distribution. The Extended Stochastic Petri Nets of Dugan, Trivedi, Geist, and Nicola [DUG84] is a very general model. The firing times have arbitrary distributions. If the underlying stochastic process is analytically solvable, then it is solved; otherwise, the net is simulated.

Some work has been done on attempting to merge the TPN and SPN approaches. Two noteworthy attempts are Molloy's Discrete-Time Stochastic Petri Nets [MOL85] and the Deterministic Stochastic Petri Nets of Marsan and Chiola [AJM85b]. Both approaches allow deterministic firing times, but only with fairly strong restrictions.

An alternative perspective is reflected in the models of Zuberek [ZUB80], and Razouk and Phelps [RAZ84]. Their models are TPNs but they build the state space and analyze it as a stochastic process. For many systems to be modeled, deterministic time is the right semantics. In addition, interpreting the evolution of a system as a stochastic process allows the powerful results of stochastic process theory to be used. Moreover, often when deterministic time is not the right semantics, the desired random variable can be constructed from deterministic firing times. For these reasons, our approach is built on their work.

Despite the promise of their approach, serious problems existed. They assumed fairly strong restrictions on the structure of the net. In addition, the stochastic process foundation of their analysis was unclear. A more detailed description of their models and restrictions is in the following subsection. In Chapter 4 we compare and contrast our extension of their work with the SPN models.

1.3.3 Most Relevant Previous TPN Models

In this subsection we describe the TPN models of Zuberek, and Razouk and Phelps. The TPN model [ZUB80,RAZ84] is a Petri net which has been augmented to include a set of firing durations (D), a set of firing frequencies (F), and a set of named resources (R). Each set is associated with the transitions in the net. Letting S denote the set of reachable states, $\Re^+$ denote the positive reals, and $\mathcal{P}$ denote the power set, the model is formally defined as follows:

$$TPN = (P, T, A, M_0, D, F, R)$$

where

$$
\begin{array}{rl}
P = \{p_1, p_2, \ldots, p_n\} & \text{(places)} \\
T = \{t_1, t_2, \ldots, t_m\} & \text{(transitions)} \\
A: \{P \times T\} \cup \{T \times P\} \rightarrow \{0, 1, 2, ...\} & \text{(directed arcs)} \\
M_0: P \rightarrow \{0, 1, \ldots\} & \text{(initial marking)} \\
D: T \times S \rightarrow \Re^+ \cup \{0\} & \text{(firing durations)} \\
F: T \times S \rightarrow \Re^+ \cup \{0\} & \text{(firing frequencies)} \\
R: P \cup T \rightarrow \mathcal{P}(\{r_1, r_2, \ldots, r_k\}) & \text{(resources)}
\end{array}
$$

The state of a TPN is defined differently than in untimed Petri Nets because firing a transition is not an atomic operation. A transition has an associated deterministic firing duration. There is a *start firing*, and an *end firing* event. In between the firing is *in progress*. The removal of tokens from a transition's input places occurs at start firing. The placement of tokens on a transition's output places occurs at end firing. While the firing of a transition is in progress, the time to end firing, called the *remaining firing time* (RFT), decreases from the firing duration to zero (without causing a change in the marking of the net). Because firings can be in progress when a marking change occurs, a state is only partially defined by the distribution of tokens. A state must also include the RFT of each firing in progress. A state is thus a marking vector and a set of RFT's.

Also unlike an untimed Petri net, the next state is not generated by a single start firing or end firing event. Instead it is generated by a *set* of start firings or a *set* of end firings which occur simultaneously. Given a particular state, the basic rule for finding the possible next states is straightforward. Find how many *enablings* of each transition exist. (Instead of a transition being either enabled or not, it has a nonnegative number of *enablings*. N enablings of a transition exist if each of its input places contains a number of tokens equal to at least N times its multiplicity.). Find the *maximal* sets[1] that can start firing simultaneously. Each maximal defines a next state. The time spent in the original state is zero. The RFT vectors of the transitions which just started firing are set to their transition's duration. The frequencies are used in assigning probabilities to next states formed by the start firings of maximal sets. If there are no enablings, but there are some firings in progress, then the next state is generated by the end firing of all transitions with the smallest RFT (Tmin). The time-in-state value in this case is Tmin. If there are no enablings and no firings in progress, then the net remains in the current state forever.

The rule that next states are generated by sets of events that occur simultaneously, is not strictly necessary. The advantage of having it is that the state spaces generated are dramatically smaller. The disadvantage is that the algorithms for building the state space are more complicated.

Zuberek suggested that the reachability graph of the Timed Petri Net be viewed as a Markov Chain and that performance measures be computed using standard

[1] A set with property α is a *maximal* set with property α if it is not a proper subset of any other set with property α.

techniques for analyzing the Markov Chain's long run behavior. Extensions of his work, however, are desirable in two areas. One, he only proposed a method for constructing a net's reachability graph for a restricted class of nets: nets that are *safe* and *free choice*. A net with a given initial state is said to be *safe* if, for every state in the reachability set, no place has more than one token. A net is *free choice* if each place that is an input to more than one transition is the only input to those transitions. Two, even for safe and free choice nets, the structure of the reachability graph (i.e. the Markov Chain) may be such that Zuberek's approach gives incorrect values for the performance measures. The states in a discrete time Markov Chain can be divided into classes. A set of classes, called *recurrent* classes, is important because in the long run the model will reach and stay in one of these classes. Zuberek's approach gives correct values only when there is exactly one recurrent class.

Razouk and Phelps [RAZ84] extend Zuberek's work in the first of the two areas above. They allow a superset of the class of safe, free choice nets. Two or more transitions are said to be in the same *conflict set* if their sets of input places intersect. Two conflict sets *overlap* if at least one transition is in both. Razouk and Phelps make the restrictions that conflict sets do not overlap and that all transitions in a conflict set are *mutually disabling*, i.e. firing of one, disables all the others. They maintain Zuberek's restriction in the second area (they call this, requiring a *cyclic* net).

Razouk and Phelps also introduce the concept *resources*, originally proposed in E-Nets [NUT72]. A resource in their model can be associated with one or more transitions. Whenever one of those transitions is firing, the resource is in *use*. If more than one of these transitions is firing simultaneously, the resource has several *usages* occurring. By building and analyzing the net's reachability graph we can find the average number of uses of a resource over time. This average, if properly implemented and interpreted, can be used to obtain a variety of meaningful performance estimates.

Chapter 2

The GTPN Model

The GTPN model extends the models of Zuberek and Razouk and Phelps by: 1) removing all restrictions on the net except the obvious one that the state space be finite, and 2) computing correct performance estimates for any reachability graph (i.e. an arbitrary embedded discrete parameter, finite state Markov Chain). We also allow the firing duration to be an arbitrary real number (the non-integer case is not discussed by Zuberek or Razouk and Phelps) and we allow resources to be associated with places as well as transitions.

A third extension we have found useful involves firing durations and frequencies. In the models of Zuberek and Razouk and Phelps, the duration and frequency are state-independent constants. In the GTPN model a transition's firing duration and frequency can be expressions containing immediate values (real and integer), names of places, names of transitions, and arithmetic, relational, and logical operators. A place name stands for the number of tokens in that place in the current state. A transition name represents the value one if at least one firing of that transition is in progress in the current state, and is otherwise zero. The state-dependent durations and frequencies become deterministic values when used to determine time-in-state and next state probabilities for a state in the reachability graph. (Note that Petri net *inhibitor arcs* can be modeled using the state-dependent frequency expressions.)

Besides a firing duration, frequency, and set of resources, a GTPN transition has a flag associated with it that is used in computing the next state probabilities as described in section 2.2.

Figure 2.1 shows an example GTPN net, including the initial state distribution of tokens. Each place and transition is labelled. Each transition has, from left to right, its firing duration expression, its frequency expression, its flag, and its list of resources. This example models users at terminals, who with a geometric think time generate requests for a server. There is one token on place P1 for each user. Transitions T1 and T2 implement the think time. Transition T3 implements a load-dependent server with a firing duration that depends on the number of tokens on P2.

In Figure 2.2 and Table 2.1 we show the reachability graph for the simple GTPN

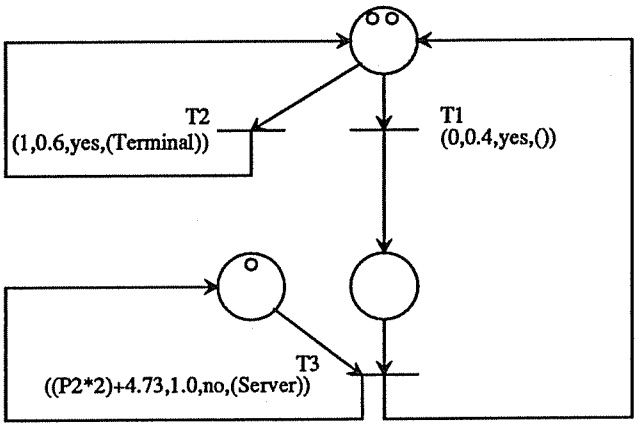

Figure 2.1: Example of a GTPN net

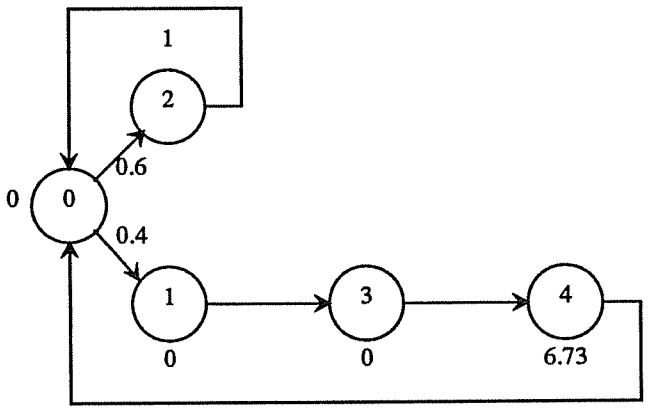

Figure 2.2: Reachability Graph for example

Table 2.1: Reachable States for example

States	Marking			RFT Set	Resources
	P1	P2	P3		
0	1	0	1	{}	{}
1	0	0	1	{T1,0.0}	{}
2	0	0	1	{T2,1.0}	{Terminal(1)}
3	0	1	1	{}	{}
4	0	0	0	{T3,6.73}	{Server(1)}

in Figure 2.1 assuming only one user. The labels on the edges of the graph are the next state probabilities. The labels on the vertices of the graph are the values for time-in-state. The marking vectors are shown in the table. The RFT sets are shown as a list of pairs, with one pair per in-progress firing of a transition. The first component of each pair is the name of the transition. The second component is the remaining firing time. The resources used and their number of uses are also shown in the table.

The Razouk and Phelps' TPN model does not allow multiple tokens on place P1. Allowing such nets complicates constructing the reachability graph. An overview of our reachability algorithm, which handles these complications, is in Figure 2.3. The TimeInState and ResUsages (number of usages of a resource) functions are used in the performance analysis as described in Section 3. The algorithm has two complex parts: 1) finding the next states when the next states are due to maximal sets of transition enablings which start firing together, and 2) assigning probabilities to next states. These two parts are discussed in the Sections 2.1 and 2.2.

2.1 Finding Maximals

Our first point is somewhat discouraging. In an arbitrary state in a net the number of maximals in the worst case is exponential in the number of enablings.

Theorem 2.1 *Consider a state S in a GTPN with n enablings. The number of maximals is* $\Omega(2 \cdot \frac{2^n}{\sqrt{2\pi n}})$.

pf. First we will construct a state in a net such that the number of maximals is $\binom{n}{n/2}$. Consider the case where n is even (The n is odd case is similar.). Consider the net with n transitions and one place P which is an input place for all the transitions. Assume each of the n transitions also has another input place, with one initial token, which is also an output place of the transition. Let the place P have $n/2$ tokens. In this state, there are n enablings (i.e. each transition is enabled once), and $\binom{n}{n/2}$ maximals. Thus, the number of maximals is $\Omega\binom{n}{n/2}$. The observation that by Stirling's approximation, $n! = \sqrt{2\pi n}(n/e)^n(1 + O(1/n))$, completes the proof. ∎

This result should not be given too much weight. In practice, we find that the number of maximals is far less than exponential in the number of enablings. Theorem 2.1 does, however, point out that the space of potential maximals is large. Consequently, an algorithm for finding the maximals must be carefully thought out in order to prevent poor performance when the actual number of maximals is small. The algorithm described below meets this criterion. When we profiled our GTPN tool, the percentage of total program time taken by this algorithm was less than 5% for the analysis of large nets [1].

[1] we profiled our program using *gprof* under 4.2 bsd UNIX^{TM} running on a VAX-11/780^{TM}

```
X.State ← Initial State; X.Class ← Frontier
while at least one Frontier state, Y do begin
    if Y is a duplicate of an Interior state Z then
        Y.Class ← Duplicate
    else begin
        Find the set of enablings in Y
        If no enablings and the RFT set is empty then
            Y.Class ← Terminal
        else if any enablings then begin
            Find the set of maximals of enablings
            Compute the probability of each maximal
            For each maximal M create a new state Z from Y
                Remove tokens from the input places of
                    transitions that have enablings in M
                Add a firing, f_t, to the RFT set for each
                    enabling in M
                Set the RFT of each added firing, f_t, to the
                    firing duration of transition t
                Z becomes a child of Y; Z. Class ← Frontier
            for all resources ResUsages[Y] ← count uses
            TimeInState[Y] ← 0; Y.Class ← Interior end
        else begin
            Let Tmin be the smallest RFT in Y
            Create state Z from Y by subtracting Tmin from
                each RFT in Y
            For each firing f_t whose RFT = 0 in Z do
                Add tokens to the output places of transition t
                Remove f_t from the RFT set
            Z becomes a child of Y; TimeInState[Y] ← Tmin
            for all resources ResUsages[Y] ← count uses
            Z.Class ← Frontier; Y.Class ← Interior end
    end
```

Figure 2.3: Overall State Space Algorithm

Our algorithm consists of two independent subalgorithms *Partition* and *FindMax*. The Partition algorithm partitions the set of enablings into *Generalized Conflict Sets*, such that maximals of the partitioned sets can be efficiently combined to generate all the maximals for the original set of enablings. The Partition algorithm does not specify how the maximals for each partition member are found. Maximals are found for each member of the partition by FindMax.

2.1.1 The Partition Algorithm

The Partition algorithm has two parts. The first part constructs a static partition of the set of transitions, T. Define the *directly-conflicts-with* relation on T as follows. For all t_1 and t_2 in T, t_1 directly-conflicts-with t_2 if the set of input places for t_1 intersects the set of input places for t_2. Define the *conflicts-with* relation on T as the transitive closure of the directly-conflicts-with relation. The conflicts-with relation is clearly reflexive, symmetric, and transitive, so it is an equivalence relation on T. The partition of T induced by conflicts-with is denoted by $\{GCS[i] | 1 \leq i \leq N\}$, where $GCS[i]$ is the ith member of the partition and N is the size of the partition (GCS stands for *generalized conflict set*). Note that a transition which does not share any input places with any other transitions forms a GCS of size one.

Part two of the Partition algorithm uses the conflicts-with partitioning of T to partition the set of enablings, $Enablings(S)$, for each state S. The desired partition of $Enablings(S)$ is

$$\{EGCS[S,i] = Enablings(S) \cap GCS[i] | \text{partition i}\}.$$

At this point, FindMax is applied to each partition member to find its *local maximals*. Let $Maximals[Enablings(S)]$ be the set of maximals over all the enablings. We have constructed our partition such that $Maximals[Enablings(S)]$ is simply the Cartesian product of the local maximals.

$$Maximals[Enabings(S)] = FindMax(S, EGCS[1]) \times \cdots \times FindMax(S, EGCS[N])$$

2.1.2 The FindMax Algorithm

We want to find the local maximals for a generalized conflict set, G, in a given state S. Any subset of $EGCS[S,G]$ is a potential local maximal (for brevity's sake we will call a local maximal, a maximal, in this section) The power set, $\mathcal{P}(EGCS[S,G])$, unfortunately, can be a large search space. Our goal is to minimize the number of members of this power set that we examine. Note that set inclusion defines a partial order on $\mathcal{P}(EGCS[S,G])$ which, in turn, induces a directed acyclic graph (see Figure 2.4). This graph has one root which represents the set $EGCS[S,G]$ itself. The FindMax algorithm does a breadth-first search of this graph searching for vertices that are maximals. The traversal is implemented in the standard way using a queue. Initially, the root vertex is the only entry on the queue.

Searching the graph breadth-first causes the order in which vertices are examined to have an important property: if 1) the set of enablings, E_1, represented by a vertex can fire together, and 2) E_1 is not a subset of any maximal already found, then E_1 is a maximal. In other words, it is impossible that some vertex examined later will have a set of enablings, E_2, that can all fire together and for which E_1 is a subset. Thus, to find the maximals we just do the breadth-first search, checking each vertex to see if it satisfies properties 1) and 2).

Three methods are used to avoid searching the entire graph. One, if a vertex's set of enablings, E_1, can all fire together, then none of its descendents needs to be examined (so its children are not added to the queue). Two, a pointer is used to ensure that vertices are never examined more than once. Note that the graph is not a tree. Thus, a naive breadth-first search would cause vertices to be examined multiple times. Three, the pointer used in method two is used to implement a heuristic for pruning subtrees.

Methods two and three are based on a pointer, V_p, associated with each vertex, V, in the graph. Consider the set of enablings, E_1, represented by V to be described by a vector of nonnegative integers. The ith component of the vector gives the number of enablings of the ith transition. V_p points at one of the components in this vector. The set of transitions to the left of V_p is V_L. The set of transitions to the right of V_p and including the transition pointed to by V_p is V_R.

The pointer V_p can be used to ensure that a vertex is only examined once. The method is as follows. Set the V_p of the root vertex to point to the leftmost transition. For each parent vertex, subtracting one enabling from the transitions in V_R defines a child vertex. For each of these child vertices, set its V_p to point to the transition that was decremented. It can be shown inductively that at each level in the graph, the sets V_L are distinct for each vertex. Consequently, any child resulting from decrementing V_R of one parent can never be the same as a child resulting from decrementing V_R of another parent. Note, also, that the use of V_p does not cause any state to not be examined that should be examined. For each state, c, that should be examined, there is a vertex, p, whose set of enablings over the transitions in V_L matches c's set of enablings over the same transitions. Consequently, c will become a child of p.

Method three is a simple optimization made possible by the existence of V_p. If the enablings in V_L cannot fire together, then it is irrelevant what enablings are subtracted from V_R. Consequently, none of the child vertices resulting from decrementing V_R need be added to the queue.

Figures 2.4 and 2.5 illustrates the FindMax algorithm. Figure 2.4 is the complete directed acyclic graph defined on the power set of $\{1, 2, 3\}$ by set inclusion. Figure 2.5 shows only the edges that will be used in the breadth first search. The vertical arrows in the middle row are the pointers used in the optimizations.

Partition and FindMax are used to calculate all maximal sets of enabled transitions which can start firing together. Next state transition probabilities must be assigned to these maximal sets. This is discussed in the next section.

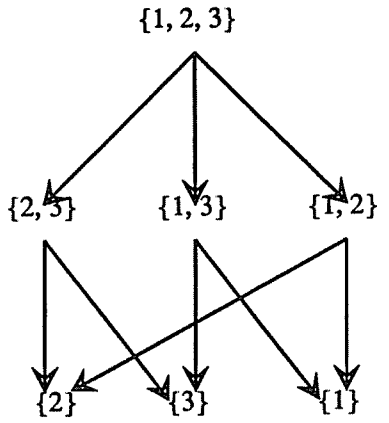

Figure 2.4: FindMax Example: directed acyclic graph

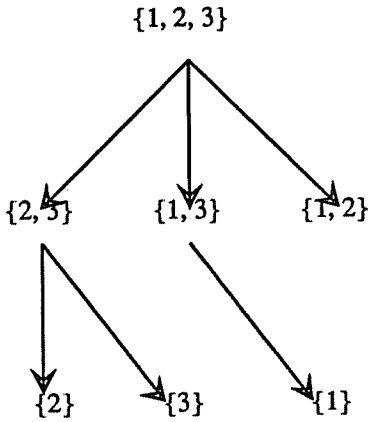

Figure 2.5: FindMax Example: optimized search

2.2 Computing Probabilities

To interpret the reachability graph as a Markov Chain we need to assign probabilities to next states. In the nontrivial case, we need to assign a probability to each maximal set of transitions that can start firing together. From the previous section we know that a maximal is the union of a set of *independent* local maximals, one from each GCS. Thus a reasonable probability for the maximal is the product of the probabilities for the local maximals. Suppose that $LocalMax[k,i]$ is the kth local maximal of the ith generalized conflict set. Suppose that there are N generalized conflict sets and the jth global maximal is the union over all GCS's of the j_ith local maximal of the ith generalized conflict set. Then

$$Pr\{Maximal[j]\} = \prod_{\{i:i=1,\ldots,N\}} Pr\{LocalMax[j_i,i]\}$$

In order to compute the probability of a local maximal, we take the product of the frequencies of all the enablings in the local maximal. In some cases, this is multiplied by a number $NumComb$ discussed below. Then for each local maximal, we normalize this product by dividing it by the sum of the products over all local maximals. More formally, suppose the ith GCS has M local maximals and the frequency expression for the kth enabling in the ith GCS is f_k. Then,if NumComb is used, our formula for $Pr\{LocalMax[j_i,i]\}$ is

$$Pr\{LocalMax[j_i,i]\} = \frac{NumComb[LocalMax[j_i,i]] \times \prod_{\{k:k\in LocalMax[j_i,i]\}} f_k}{\sum_{\{m:m=1,\ldots,M\}} NumComb[LocalMax[m,i]] * \prod_{\{k:k\in LocalMax[m,i]\}} f_k}$$

If NumComb is not used, our formula for $Pr\{LocalMax[j_i,i]\}$ is

$$Pr\{LocalMax[j_i,i]\} = \frac{\prod_{\{k:k\in LocalMax[j_i,i]\}} f_k}{\sum_{\{m:m=1,\ldots,M\}} \prod_{\{k:k\in LocalMax[m,i]\}} f_k}$$

$NumComb$ means *number of combinations* and is a combinatoric value associated with each local maximal. This value is defined as the number of ways tokens can be removed from input places in order to implement that local maximal. As a simple example, suppose the local maximal consists of one enabling of one transition with one input place, two arcs connect the place to the transition, and the input place has three tokens. In this case, the combinatoric value $NumComb$ is $\binom{3}{2}$.

Computing $NumComb[LocalMax]$ is done by decomposing it first on the transitions in the local maximal and second on the input places for the given transition. The number of ways that the tokens can be removed from the given input place by the given transition is a binomial coefficient. As we consider each transition,

the value in the upper position of this binomial coefficient is changed to reflect any removals made by transitions considered earlier.

More formally, consider the local maximal, $LMax$, of the ith generalized conflict set. Let $InputPlace[t]$ be the set of input places for transition t. Let $Enab[t]$ be the number of enablings of transition t in this maximal. Let $TokNeeded[t,p]$ be the number of tokens needed from input place p by one enabling of transition t. Note that $Enab[t] * TokNeeded[t,p]$ is the number of tokens needed from place p by transition t in this maximal. Let $TokLeft[p]$ initially be the number of tokens on input place p in the parent state. After looking at an input place, p, $TokLeft[p]$ is updated to reflect the start firing of all the enablings of transition t. With this notation we have:

$$NumComb[LMax] = \prod_{\{t:t \in LMax\}} \prod_{\{p:p \in InputPlaces[t]\}} \binom{TokLeft[t,p]}{Enab[t] * TokNeeded[t,p]}$$

From practical experience it appears that in some cases it is reasonable to use this combinatoric value when assigning probabilities to local maximals. A boolean flag associated with each transition specifies whether this should be done. Only if the flag is *yes* for all transitions in the maximal, is $NumComb$ used. Note that if NumComb is used, the next state probabilities are the same as if we constructed the reachability graph by allowing one start firing event at a time with all CntComb flags set to zero, and then summed the probabilities over all paths leading to the state which represents the maximal set.

The motivation for our method of assigning probabilities to local maximals is that it assigns the right probabilities in the important case where all the enablings in a local maximal are independent events. In the case where there are dependencies between the enablings it is difficult to envision a single formula that will always generate the "right" probabilities. This case motivated our introduction of state-dependent frequency expressions. Such frequency expressions can specify what probabilities maximals should have in different markings. Allowing state-dependent frequency expressions also allows the possibility that in some states a transition's frequency expression may evaluate to zero even though it has one or more enablings. We remove these enablings from the set of considered enablings before finding the local maximals.

Chapter 3

GTPN Performance Analysis

For the purpose of performance analysis, we view the GTPN as a stochastic process. The time-in-state is a deterministic function, TimeInState, of the state. Nevertheless, the process is stochastic because of the probability distribution over the possible next states. Since the time-in-state can be an arbitrary real number, the process is a continuous time stochastic process. The parameter set is described in Section 3.5. The states of the stochastic process are divided into classes. In the long run, with probability one, the process will reach and stay in one of the set of classes called *recurrent* classes [1]. Consequently, the *long run fraction of time spent in each state* depends on which recurrent class the process reaches in the long run. For each recurrent class the long run fraction of time spent in each state forms a probability distribution over the states. Thus, there is a vector of long run probability distributions with one component for each recurrent class. In addition, we can compute the probability (the *absorption* probability) of reaching each recurrent class in the long run. These absorption probabilities allow us to assign relative weights to the components of the vector of long run probability distributions.

The number of usages of a given resource is also deterministic for a given state; it is a function ResUsages of the state. Consequently, ResUsages, being a function of a random variable, is a random variable. A performance estimate for a resource is a vector with one component for each recurrent class. The value for recurrent class R's vector component is the long run expectation and distribution of that resource's ResUsages random variable, with respect to R's long run probability distribution. In other words, the expectation is the weighted sum of the long run fractions of time spent in each state, given that in the long run the process is in class R. The weight of a state is the number of resource usages in that state. The distribution of a ResUsages random variable is obtained by summing the probabilities of the

[1] Note that a terminal state in the reachability graph is a recurrent class due to the self-loop we added (see Figure 2.3).

states that use the resource the same number of times, given that in the long run the process is in class R.

Our approach to computing performance estimates uses the key observation [2] that the times at which state changes occur form an embedded, discrete time, finite state Markov Chain. Consequently, our approach has four parts: 1) building the Markov Chain, 2) aggregating the states in order to reduce the size of the state space, 3) analyzing the Markov Chain, and 4) computing resource usage distributions and expectations in the original stochastic process.

Building the Markov Chain involves building the reachability graph and assigning next state probabilities, as described in Chapter 2. Our aggregation rule is: Any state S_2 can be aggregated with its parent state, S_1, if and only if S_1 is S_2's only parent and S_2 is S_1's only child. The number of usages of a resource in an aggregated state equals the sum of its usages in the internal states weighted by the relative fraction of time spent in each internal state. Part 3, the Markov Chain analysis, has three steps: a) finding the chain's recurrent classes, b) finding the absorption probability for each recurrent class, and c) finding, for each recurrent class, the long run fraction of visits to each state. These steps are discussed, respectively, in sections 3.1, 3.2, and 3.3.

Part 4 has two steps: a) for each recurrent class R, computing the long run fraction of time spent in each state (the TimeInState function and the long run fraction of visits to each state are used to do this), b) for each recurrent class R, use the ResUsages functions and the long run fraction of time spent in each state to find the long run distribution and expected number of usages of each resource. These two steps are discussed in section 3.4.

As mentioned above, the long run resource usage distributions and expectations are the performance estimates (given that the process is in class R in the long run). For each resource we thus have a vector of performance estimates. If desired, the expectations could be weighted by the absorption probabilities to give a single performance estimate.

In section 3.5 we give a more precise definition of the parameter set of the GTPN stochastic process. In sections 3.6 and 3.7 we discuss the important numerical issues involved in analyzing the Markov Chain.

3.1 Finding Recurrent Classes

In order to find the recurrent classes we need to first define a *recurrent* class. Recall that $\forall n\ P_{ij} = Pr\{X_{n+1} = j | X_n = i\}$. $P = [P_{ij}]$ is the *one-step transition probability matrix*. $P^{(n)}$, the *n-step transition probability matrix* is defined similarly. $f_{ii}^{(n)}$ is the probability that, starting from state i, the first return to state i occurs at the nth transition. A state is *recurrent* if $\sum_{n=0}^{\infty} f_{ii}^{(n)} = 1$. In other words, a state is recurrent

[2] The GSPN model uses a similar observation.

if and only if, after the process starts from state i, with probability one the process returns to state i in a finite length of time. A state is *transient* if it is not recurrent. State j is said to be *accessible* from state i if $P_{ij}^{(n)} > 0$ for some integer $n \geq 0$. Two states i and j that are each accessible to the other, are said to *communicate.*

Note the following facts. *Accessibility* defines a partial order on the states. This partial order implies a directed graph with the vertices being the states. *Communication* is an equivalence relation on the states. Thus it partitions the states into subsets called *classes.* The *communication* classes are the strongly connected components of the accessibility graph. These strongly connected components form a directed acyclic graph, DAG.

All the states in a class are recurrent or none are, so we can speak of *recurrent* classes and *transient* classes. In the case of a finite state space [3], the *recurrent* classes are the leaves of the DAG [KEM76] of strongly connected components. The interior nodes of the DAG are the *transient* classes.

Given that the recurrent classes are the leaves of this DAG, the algorithm to find the recurrent classes is immediate. The reachability graph of the GTPN is the accessibility graph. Create the DAG by finding the strongly connected components of the accessibility graph. We do this using Tarjan's $O(n)$ algorithm [SED81]. Then find the leaves of the DAG by a depth first search starting at the initial state.

3.2 Absorption Probabilities

In a finite state Markov Chain, if we start in a state in a transient class we will eventually reach and stay forever in one of the recurrent classes. We are said to be *absorbed* by a particular recurrent class. Computing the probability of absorption in a particular recurrent class R given a particular initial state, i, can be done using a standard technique called *first-step analysis* [TAY84]. On the first state change, the process will move from state i to a state j that is in a transient class or in a recurrent class. If j is in class R, the future probability of being absorbed by class R is one. If j is in another recurrent class, the future probability of being absorbed by class R is zero. If j is in a transient class, then, by the memoryless property, the probability of being absorbed by class R is the same as if j were the initial state.

More formally, suppose that the states in all the transient classes are numbered $0, ..., r-1$ and consider a fixed recurrent class R and fixed initial state i. Let $U_i = Pr\{$ Absorption in class $R | X_0 = i\}$ for $0 \leq i < r$.

$$U_i = \sum_{\{j \in R\}} P_{ij} + \sum_{j=0}^{r-1} P_{ij} U_{jR}, \qquad i = 0, 1, ..., r-1$$

This equation cannot be be solved in isolation. However, if we consider all possible initial states, then we have a system of linear equations that can be solved for the

[3] This is not true if the the state space is countably infinite. A simple counterexample is a one-dimensional, asymmetric random walk.

U_i's. The *mean time to absorption* can be computed using a similar system of linear equations. Section 3.6 discusses the numerical issues involved.

3.3 Long Run Expected Fraction of Visits

If R is a recurrent class in a finite state space, then $\forall j \in R$ a number π_j exists such that $\forall i \in R$

$$\lim_{m\to\infty} E[\frac{1}{m}\sum_{k=0}^{m-1} 1_{\{X_k=j\}}|X_0 = i] = \lim_{m\to\infty}\frac{1}{m}\sum_{k=0}^{m-1} P_{ij}^{(k)} = \pi_j$$

The leftmost expression above is *the long run expected fraction of visits spent in state j*. $1_{\{X_k=j\}}$ is the indicator random variable that equals one when the outcome chosen is in the event $\{X_k = j\}$ and 0 otherwise.

We want to find these π_j's for each class R. We do this by noting that each recurrent class R in a finite state space has one and only one *stationary probability distribution* and the vector π_R of its π_j's is this stationary probability distribution. This stationary probability distribution is easy to find since it is the unique solution to the set of equations

$$\pi_R = \pi_R P_R \qquad \text{and} \qquad \sum_{j\in R}\pi_j = 1$$

The matrix P_R is $P_R = \{P_{ij}|i \in R\}$. We solve this system of equations numerically using an iterative matrix algorithm, the Power Method [SAU81]. The numerical issues involved in computing this stationary probability distribution are discussed in section 3.7.

It is also true that an arbitrary Markov Chain with a finite state space has at least one stationary probability distribution over the entire state space. However, if the Markov Chain has more than one recurrent class, then any linear combination that sums to one of the stationary probability distributions of the individual recurrent classes is a stationary probability distribution of the chain as a whole.

Our approach is correct regardless of whether the recurrent classes are periodic or aperiodic. Recall that the *period* of a state is the greatest common divisor of all integers $n \geq 1$ for which $P_{ii}^{(n)} > 0$. A state is *aperiodic* if its period is 1, else is *periodic*. All the states in a class have the same period so we can refer to a class as periodic or aperiodic. Only for the states i in an aperiodic recurrent class does the limit $\lim_{n\to\infty} P_{ii}^{(n)}$ exist. However, the long run expected fractions of visits and the stationary probability distribution exist in both cases. Again we assume a finite state space.

3.4 Resource Usage Estimates

We find, for each recurrent class R, the long run expected fraction of time spent in each state. Then, we find the long run distribution and expectation of each resource with respect to each recurrent class.

Let S be the set of states. Let $RelTime(S_1)$ be the long run expected fraction of time spent in state S_1, given that the process is absorbed in class R. From the Markov Chain we know the long run expected fraction of visits to each state k, π_k, given absorption in class R. $RelTime(S_1)$ can be computed, using the TimeInState function, as follows:

$$RelTime(S_1) = \frac{\lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} E[1_{\{X(t)=S_1\}} TimeInState(X(t))]}{\lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} E[TimeInState(X(t))]}$$

$$= \frac{TimeInState(S_1)\pi_{S_1}}{\sum_{k\in S} TimeInState(k)\pi_k}$$

Recall that 1_A is the indicator random variable for the event A. To show why the last equality holds we derive its denominator. A similar derivation holds for the numerator. Let S be the set of states.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} E[TimeInState(X(t))]$$

$$= \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} [\sum_{k\in S} TimeInState(k) Pr\{X(t) = k\}]$$

$$= \sum_{k\in S} TimeInState(k) \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} Pr\{X(t) = k\}$$

$$= \sum_{k\in S} TimeInState(k)\pi_k$$

To find the long run distribution of the number of usages of a resource for each absorbing recurrent class, we simply sum over all of the states in the class that use the resource the same number of times, the long run fraction of time spent in that state. To find the long run expected number of usages of each resource for each absorbing recurrent class, we simply take the expectation of the random variable ResUsages:

$$E[ResUsages] = \sum_{k\in S} ResUsages(k) Pr\{statek\} = \sum_{k\in S} ResUsages(k) RelTime(k)$$

3.5 Parameter Set of the Stochastic Process

Since firing durations can be zero in the GTPN model, a GTPN can be in two or more states at the same "time". This slightly complicates viewing a GTPN as a stochastic process. In this subsection we discuss that complication and how it can be resolved. The complication is that the right halfline cannot be the parameter set of the GTPN stochastic process. To see this, recall that a stochastic process is a family of random variables indexed by the parameter set and a random variable is a function from the sample space into the reals. If the right halfline were the parameter set, then on some sample path at some parameter t, the random variable $X(t)$ could simultaneously hold more than one value in its range. This contradicts $X(t)$'s being a function.

This complication is resolved by using a different parameter set. The parameter set used is the lexicographically ordered Cartesian product of the nonnegative reals and the natural numbers. The parameters are assigned in the following way. Consider an arbitrary sample path. At any time t in the nonnegative reals, if there are $n(n \geq 0)$ instantaneous state changes, then $X(t,0)$ is the state before the first (if any) state change, $X(t,1)$ is the state after the first state change, ... $X(t,n-1)$ is the state after the $n-1$th state change, $X(t,m), m \geq n$ is the state after the nth instantaneous state change. Since at most a countably infinite number of instantaneous state changes can occur, the process is never in two or more states at the same "time". Note that the parameter set of the embedded Markov Chain need only be the nonnegative integers with the nth parameter meaning the nth state change.

3.6 Computational Issues: Transient States

As discussed in section 3.2, two of the characteristics of the system's performance that we need to determine are the absorption probabilities and mean time to absorption. First Step Analysis was proposed as the method for computing these characteristics. In this section we discuss the computational issues involved in determining those characteristics. Section 3.6.1 discusses some of the efficiency issues involved in implementing First Step Analysis for computing the absorption probabilities. Section 3.6.2 describes an optimization which can significantly accelerate solution of the First Step Analysis equations. Section 3.6.3 covers the analogous material for computing the mean time to absorption. In an important special case that arises frequently in GTPN models, an alternative to First Step Analysis can be used to compute absorption probabilities. Section 3.6.4 describes this still more efficient method.

3.6.1 First Step Analysis

Using First Step Analysis implies that r systems of n linear equations are solved where r is the number of recurrent classes, and n is the number of transient states in the Markov Chain. Solving one system of equations determines the absorption probability of interest for one recurrent class. Since the GTPN is intended to be a practical tool, an efficient solution method is important.

We write the systems of linear equations in matrix form as follows:

$$U_R = P_{TR} + P_T U_R, \tag{3.1}$$

where U_R is the $n \times 1$ vector of absorption probabilities for recurrent class R from transient states $0, 1, ..., n-1$, P_{TR} is the $n \times 1$ vector of one-step transition probabilities from transient state i to any state in R, and P_T is the $n \times n$ one-step transition probability matrix for the transient states.

The standard form for a system of linear equations is $Ax = b$ where A is $n \times n$, x is $n \times 1$, and b is $n \times 1$. Our form maps into the standard form by letting $A = (P - I)$, $x = U_R$, and $b = -P_{TR}$:

$$(P_T - I)U_R = -P_{TR}. \tag{3.2}$$

There are many methods of solving systems of linear equations. Gaussian elimination is the primary direct method. Iterative methods, such as the Gauss-Seidel method, are much more efficient for large matrices. Before selecting a solution method, however, we will discuss an important optimization that can be applied to First Step Analysis.

3.6.2 Optimization of First Step Analysis

An optimization exists that can significantly accelerate solution of the linear systems of equations for First Step Analysis. This optimization is based on a key observation: by permuting the rows and columns of the matrix P_T, P_T can be put into block upper triangular form, where each diagonal block represents the transitions within one transient class of the Markov Chain. To see this, recall that the Markov Chain classes are the strongly connected components in the reachability graph. They thus form a directed acyclic graph (DAG), the condensed graph. As with any DAG, the vertices of the condensed graph can be numbered via a topological sort so that the number assigned to a vertex is always less than the numbers assigned to its children. This numbering defines the permutation that generates the block upper triangular form. We find the strongly connected components using Tarjan's $O(n)$ algorithm, and perform the topological sort on the DAG using another linear-time depth-first search [SED81].

After the permutation, equation 3.2 is of the following form:

$$\begin{pmatrix} A_{11} & A_{21} & \cdots & A_{1N} \\ & A_{22} & & A_{2N} \\ & & \ddots & \vdots \\ & & & A_{NN} \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_n \end{pmatrix} = \begin{pmatrix} -P_{1R} \\ -P_{2R} \\ \vdots \\ -P_{nR} \end{pmatrix} \tag{3.3}$$

where the elements are matrices, each diagonal block A_{ii} is square of order n_i and

$$\sum_{i=1}^{N} n_i = n.$$

The system 3.3 can be solved as a sequence of N smaller problems. Problem i is of order n_i and the matrix of coefficients is $A_{ii}, i = 1, 2, ..., N$. The procedure is as follows [PIS84] (see also [TAR76]):

1. Solve the last subsystem, with A_{NN} as the matrix of coefficients, for the last n_N unknowns. Compute the vector x_N of order n_N.
2. Subtract the products $A_{jN}x_N$ from the right-hand side for $j = 1, ..., N-1$. A block upper triangular matrix of order $N-1$ is obtained, and the procedure is repeated until the complete solution is obtained.

Note that the assumption must be made that the diagonal blocks in Equation 3.2 are nonsingular. The solution of each subsystem can be done by any method for solving linear equations, such as Gaussian elimination or the Gauss-Seidel iteration.

3.6.3 Computing Mean Time to Absorption

The mean time to absorption can also be computed using a first-step analysis. One system of n linear equations is solved where n is the number of transient states. Each transient state i has one equation. Starting in state i, the mean time to absorption is the time-in-state for state i plus zero if the next state is in a recurrent class and plus P_{ij} times the mean time to absorption for state j if state j is transient. Note that the Markov property is again being used. Formally, each equation is of the form:

$$U_i = TimeInState(i) + \sum_{j=0}^{n-1} P_{ij}U_j, \qquad i = 0, 1, ..., n-1$$

If the integer 1 replaces the time-in-state, then the mean number of visits to transient states before absorption can be computed. The mean number of visits to a given set of transient states before absorption can be computed if the time-in-state is replaced by the integer 1 when visiting a state in the given set and is replaced by the integer 0, otherwise.

The discussion in Section 3.6.2 applies here also. In particular, the system of equations can be changed into the form $Ax = b$, permuted into block lower triangular form, and then solved as outlined.

3.6.4 More Efficient Solution of an Important Special Case

In a special case that occurs frequently in GTPN models, an efficient alternative approach based on the condensed reachability graph (i.e. one vertex per strongly connected component) can be used to compute absorption probabilities. In this alternative, probabilities are assigned to the edges leaving each vertex in the condensed graph A depth first search is then done. During the depth first search the probability along each path to each leaf is determined by taking the product of the edge probabilities along the path. The absorption probability for a given leaf is then simply the sum of the path probabilities terminating at that leaf.

The difficulty with this approach is with determining the edge probabilities leaving a vertex in the condensed graph, If all the exit edges originate at the same vertex V_1 within the strongly connected component, then there is not a problem. Find all the edges of which V_1 is the parent that are exit edges. Sum their probabilities as edges in the original graph. The probability of each edge in the condensed graph is its probability in the original graph normalized by this sum.

The problem is when more than one vertex, say V_1 and V_2, in the original graph are parents of exit edges. Determining the probability that each of these parent vertices is the vertex from which exiting occurs is dependent on the detailed structure of the strongly connected component. Our current implementation uses this condensed graph approach when the special case of one parent vertex in the strongly connected component is met. When the special case is not met, first-step analysis with Gauss-Seidel iteration is used. We plan to implement the optimization described in section 3.6.2 with subsystem solution by Gauss-Seidel iteration.

3.7 Computational Issues: Stationary Probability Distributions

In this section we discuss the method we use to calculate the stationary probability distribution for each recurrent class in the Markov Chain. Let R denote a recurrent class with states $j = 0, 1, ..., n$, and let π_j represent the long run expected fraction of visits to state j, given that the modeled system is absorbed in class R.

Recall that the vector $\pi_R = (\pi_0, \pi_1, ..., \pi_n)$ is uniquely determined by the following equations:

$$\pi_R = \pi_R P_R \qquad \text{and} \qquad \sum_{j=1}^{n} \pi_j = 1, \tag{3.4}$$

where P_R is the $n \times n$ state transition probability matrix for R.

Because the matrix P_R is sparse, but potentially very large, iterative methods are more practical than direct methods. One of the most widely used iterative methods, the Power Method [JOH82,WIL65], views equation 3.4 as an eigenvalue

problem. In particular, π_R is the eigenvector associated with the unit eigenvalue of P_R. Applying the Power Method to iteratively solve for π_R, is done as follows:

$$\pi_R^{k+1} = \pi_R^k P_R.$$

Since our current implementation uses the Power Method, the remainder of this section focuses on issues related to it and, in particular, with ensuring convergence.

3.7.1 Spectral Distribution

An eigenvalue, λ, of a real $n \times n$ matrix A, is *strictly dominant*, if its modulus is strictly greater than those of all the other eigenvalues of A. Direct application of the Power Method converges to an eigenvector corresponding to the dominant eigenvalue, if and only if the matrix has a simple, strictly dominant eigenvalue.

This constraint on the direct application of the Power Method can be expressed in terms of the periodicity of the recurrent class in the Markov Chain. In order to make that connection between the spectral distribution and the periodicity of the Markov Chain class, we need the theorems of Perron and Frobenius [CIN75,SEN81,BER79]. These theorems state the following:

> An irreducible non-negative matrix, A, has a real, positive, and simple eigenvalue, α, which is *greater than or equal to* all other eigenvalues of A in modulus. The eigenvector corresponding to the eigenvalue α is strictly positive. If A is *aperiodic*, (i.e. $A^k > 0$ for some k), then α is *strictly greater than* all other eigenvalues of A. A periodic matrix with period δ, has exactly δ eigenvalues with absolute values equal to α. These eigenvalues are all distinct and are given by:
>
> $$\lambda_k = \alpha[e^{2\pi i/\delta}]^{k-1}, k = 1, 2, ..., \delta, i = \sqrt{-1}$$

If A is a stochastic matrix (i.e. all rows sum to one), then $\alpha = 1$. An aperiodic stochastic matrix thus has a simple unit eigenvalue and all other eigenvalues are of strictly smaller modulus. A periodic stochastic matrix with period δ has exactly δ eigenvalues of unit modulus all of which are simple. These eigenvalues can be regarded as a set of points around the unit circle in the complex plane, which goes over into itself under a rotation of the plane by the angle $2\pi/\delta$.

3.7.2 Ensuring Convergence

According to the above discussion, the Power Method can only be applied directly to find the stationary probability distribution when the recurrent class is aperiodic. To handle the case of a periodic recurrent class, the following theorem [STE73] concerning shifting and scaling the matrix becomes important:

Theorem 3.1 *Let A be a complex $n \times n$ matrix, and let λ be an eigenvalue of A with eigenvector x. Then:*

1. *$a\lambda$ is an eigenvalue of aA with eigenvector x.*
2. *$\lambda - b$ is an eigenvalue of $A - bI$ with eigenvector x.*

This theorem allows us to transform the matrix P_R to ensure that the unit eigenvalue is strictly greater than all other eigenvalues in modulus. In particular, we make the following transformations on P_R:

$$P'_R = \varepsilon(P_R - I) + I = \varepsilon P_R + (1 - \varepsilon)I \qquad 0 < \varepsilon < 1.$$

The first transformation, subtracting I, shifts all eigenvalues of P_R to the left by one in the complex plane. The eigenvector π_R now corresponds to the eigenvalue zero. The second transformation, multiplication by ε, shrinks all of the eigenvalues except the zero eigenvalue corresponding to π_R (the circle is now centered at $-\varepsilon$ and is of ε radius). The third transformation, adding I, shifts all eigenvalues to the right by one, creating a unit eigenvalue corresponding to π_R whose modulus is strictly greater than that of all other eigenvalues. Thus, application of the Power Method using P'_R always converges.

3.7.3 Convergence Rate

The convergence rate of this method is essentially the rate at which ${\lambda'_2}^k$ converges to zero, where λ'_2 is the second largest eigenvalue of the matrix P'_R [JOH82]. The value of ε indirectly influences the rate of convergence by scaling all of the eigenvalues. Wallace and Rosenberg [WAL66] define a suitable value for ε to be $0.99 \times [max(|P_{ii} - 1|)]^{-1}$. We have used this value, which equals 0.99 for most GTPN models, in our implementation.

In general, the value of λ'_2 depends on the size and structure of the particular GTPN model constructed, and convergence rates vary considerably. In particular, the convergence rate may be extremely slow. A recent paper by Stewart and Goyal [STE85], suggests that successive overrelaxation is a better method for solving steady-state equations for continuous time Markov chains. We plan to investigate whether this approach would also be more efficient for the GTPN.

Chapter 4

Comparison with SPN Models

As discussed in subsection 1.3.2 the SPN (Stochastic Petri Net) models form an important alternative class of performance-oriented Petri Nets. Recall that the original SPN models were independently proposed by Natkin [NAT80], Symons [SYM80], and Molloy [MOL81,MOL82]. In these models firing times are exponential random variables. Marsan, Balbo, and Conte [AJM84] generalized the continuous-time SPN model, GSPN, by allowing transitions which fire in zero time. Molloy [MOL81,MOL85] also proposed a discrete time SPN model with transition firing times that are geometric random variables. The SPN models are interesting because the reachability graph for these models are (continuous-time or discrete-time) Markov Chains. In this chapter, we compare the conflict resolution and probability assignment methods of the GTPN model and the SPN models. We then compare the modeling features of the GTPN and SPN models in four respects. Finally, we comment on the complexity issues concerning deterministic firing durations.

4.1 Conflict Resolution

The addition of timing information to the Petri net model provides several options for conflict resolution. The method of resolving conflicts in the GTPN is different than the method defined for the GSPN model.

The GTPN conflict resolution submodel uses (possibly state-dependent) *transition firing frequencies* to resolve conflicts. Our underlying assumption is that the conflict is resolved before one of the conflicting transitions starts firing. We also assume that once a transition is in progress, it cannot be preempted by a new conflict. This submodel is useful, for example, if the conflict is due to contention for a shared resource (e.g. two processors requesting use of a shared bus in a multiprocessor).

In contrast, the conflict resolution submodel for timed transitions in the GSPN is based on *competing transition delays*. The transition which fires first wins the

conflict. This mechanism is based on the view that the physical events modeled by the conflicting transition are in progress simultaneously and that the completion of one event disables the other. For example, this view holds if one transition models the successful acknowledgement of a message in a computer network, and the other transition models a timeout process. (Note that the timeout process is exponentially distributed in these models.) The removal of tokens at the time a transition is enabled is not useful for this approach.

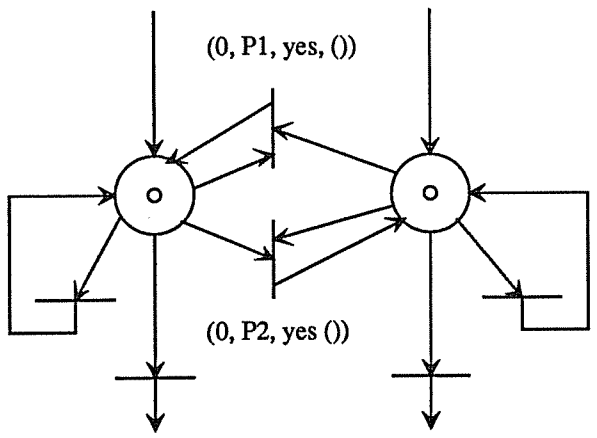

Figure 4.1: Competing Geometric Delays in the GTPN.

Each of the conflict resolution submodels is valid for the corresponding physical systems. Both can be employed in the GSPN and in the GTPN model. For example, the GSPN model uses firing frequencies to resolve conflicts for instantaneous transitions. Conversely, the GTPN can have conflicting geometric holding times such that assigned next-state probabilities are equal to probabilities based on competing delays. To do this, we assign the appropriate competing rate frequencies to instantaneous transitions at the start of a timestep and then model the geometric delays in the usual way. Figure 4.1 illustrates this technique. In this figure, $P1 = \frac{\lambda_1}{\lambda_1+\lambda_2}$ and $P2 = \frac{\lambda_2}{\lambda_1+\lambda_2}$ where λ_1 and λ_2 are the means of the competing geometric holding times.

Razouk and Phelps have defined *enabling times* in their TPN. An enabling time specifies a deterministic time that a transition must be enabled before it will start firing. The competing delays in this case are the deterministic enabling time (e.g. a timeout), and a random delay that leads to enabling of a conflicting transition. We plan to extend the GTPN reachability graph construction methods to incorporate enabling times.

Alternative conflict resolution submodels are also discussed in [HOL85a,AJM85c].

4.2 Probability Assignment

The continuous time GSPN model and the GTPN have an embedded discrete time Markov Chain, while the discrete time SPN model itself is a discrete time Markov Chain. In either case, probabilities need to be assigned to next states. Each model assigns probabilities that are consistent with its conflict resolution semantics.

When no instantaneous transitions are enabled, the SPN models assign probabilities according to competing transition delays. In the continuous time SPN's, with probability one, no two transitions in progress will finish firing simultaneously. This simplifies probability assignments. In particular, the probability of the next state associated with transition t_i's finishing first has probability equal to t_i's firing rate divided by the sum of the firing rates of the transitions that are in progress.

In the discrete time SPN, with probability greater than zero, two or more transitions in progress can finish firing simultaneously. Though more complicated, it is still possible to assign probabilities to next states using competing transition delays [MOL85]. Unfortunately, using competing delays to assign next state probabilities restricts the allowed probability assignments in the important special case of deterministic firing delays. A deterministic firing delay is represented as a geometric firing delay that has probability one of firing in the next time step. Consequently, using competing delays implies that all the next states have equal probability.

The GTPN uses *transition frequencies* to assign next-state probabilities, independent of transition delays. The transition frequency method, unlike the discrete time SPN, can assign non-uniform probabilities to next states in the case of conflicting transitions with deterministic delays. In general, although the frequency method is powerful, the assignment of static (state-dependent) frequencies which will be used for the dynamic calculation of probabilities, requires careful thought during model construction. The *random switches* used in the GSPN when there are instantaneous transitions firing is a similar method which uses firing probabilities to determine probability assignments. In the random switches method, however, instead of one frequency expression per transition, a probability distribution is explicitly given for each possible set of enabled transitions.

4.3 Modeling Features

The GTPN has capabilities for modeling and analysis lacking in the existing SPN models, in the probability assignments discussed above and in two additional respects: 1) firing durations, and 2) analysis of multiple recurrent classes.

The first respect is firing durations. The GTPN can represent deterministic firing durations which are arbitrary nonnegative real values, including zero. The GTPN can also model geometric holding times as can the discrete time SPN. Since a holding time in the discrete time SPN must be a multiple of some unit step, the GTPN can represent a larger class of firing durations, than the discrete time SPN.

Also, since the geometric distribution is the discrete-time analog of the exponential distribution, and since the GSPN cannot represent deterministic delays except with certain strong restrictions [CHI85], the GTPN can also represent a larger class of firing durations than the GSPN model.

The second respect is the GTPN's analysis of multiple recurrent classes. The GTPN allows the performance evaluation of systems that have several possible long run behaviors. In contrast, the GSPN and the discrete time SPN analyses assume that their Markov Chains are irreducible (i.e. have only one recurrent class). We believe that the SPN analysis could be developed to support multiple recurrent classes.

We note that the Extended Stochastic Petri Net (ESPN) model of Dugan, Trivedi, Geist, and Nicola [DUG84] is a SPN model that, in at least one respect, is more powerful than the GTPN. The ESPN allows arbitrary holding times, and has been shown to be useful for analyzing system response to failure. However, the ESPN is analytically tractable only for models with (simple) acyclic reachability graphs, or models where the firing times for all concurrent transitions are exponentially distributed.

4.4 Complexity Issues

Representing deterministic holding times inherently leads to greater complexity than when holding times are geometrically or exponentially distributed. This is because the memoryless property of the geometric and exponential random variables does not apply. Thus, both the GTPN and the discrete-time SPN model have the potential for large state spaces when deterministic holding times are represented.

The GTPN contains new states for start firing as well as end firing events. If we loosely identify these two state changes as one, there is an equivalence between the states in the GTPN and the discrete-time SPN when geometric holding times are in progress (i.e. one state change per time step, including a cycle back to a given state with the probability that none of the geometric delays completes in the step). When only deterministic holding times are in progress, however, the GTPN may not contain state changes (or new states) for every time step, whereas the discrete-time SPN must. Thus, it appears that the GTPN has at most twice as large a state space as the discrete-time SPN and that for some deterministic models, the GTPN has a smaller state space. We note that the RFT vector, which allows a potential reduction in the size of the state space (in comparison with the discrete-time SPN), adds minimal complexity. It is easy to assign new values to the RFT vector and to find the smallest value in it.

Due to the inherent complexity of deterministic delays, two important goals during our development of the GTPN were to minimize the size of the state space and to minimize the cost of constructing and analyzing it. These goals are reflected in the algorithms and performance analysis techniques presented in chapters 2 and 3.

The GTPN state space is reduced by generating next states for *maximal* set of events that occur simultaneously (including multiple start-firings of a single transition). Another example of reducing the cost of building the state space is our definition of Generalized Conflict Sets (GCS) and the Partition algorithm. Generalized conflicts sets are state independent and thus need only be calculated once. We thus are able to avoid a large fraction of the conflict determination cost when we are calculating the next states of the reachability graph. In contrast, the definition used in the discrete time SPN is based on partitioning enabled transitions into sets that are *in conflict*. This partition is state dependent and thus must be computed for each state.

An even more important goal than minimizing complexity is to demonstrate the usefullness of the GTPN modeling technique by applying it to interesting problems. Our contention is that this goal has also been met. The applications supporting this opinion start with the next chapter.

Chapter 5

Two Examples

This chapter begins the presentation of the applications that have been made using the GTPN modeling technique. This chapter discusses two simple applications. The first is the dining philosophers problem. The second is the scalar mode of the Cray-1 supercomputer. Chapter 6 derives exact performance estimates for multiprocessor memory and bus interference in an important general case. Previously, such a derivation had been considered computationally intractable. Chapter 7 is the first analytical comparison of cache consistency protocols that require a single shared bus.

5.1 The Dining Philosophers

The Dining Philosopher model is a well-known example which violates net restrictions in previous TPN models. Although performance of this system is largely hypothetical in nature, it serves to illustrate the capabilities of the GTPN analyzer, and it yields some insight into the timing behavior of the dining philosopher protocol.

A GTPN model of the 5 Dining Philosophers [PET81] is shown in Figure 5.1. The initial marking of the net shows all 5 philosophers thinking. We have analyzed the model with deterministic think times, ThinkTime(i), as shown, for each philosopher i. We have also used a slightly modified model (see Figure 2.1), to represent think times that are geometrically distributed with mean ThinkTime(i). After thinking, the philosopher competes for two forks which are shared with neighboring philosophers on the left and right, respectively. After acquiring the forks, the philosopher spends a deterministic amount of time eating, DineTime(i). This cycle is repeated as many times as necessary to finish the meal. The firing frequencies, f_i, associated with transitions that model fork acquisition are used to compute the probabilities that various maximal sets of competing philosophers get the forks they

require. These probabilities are calculated as described in Section 2.2. Note that all of these transitions are in the same Generalized Conflict Set.

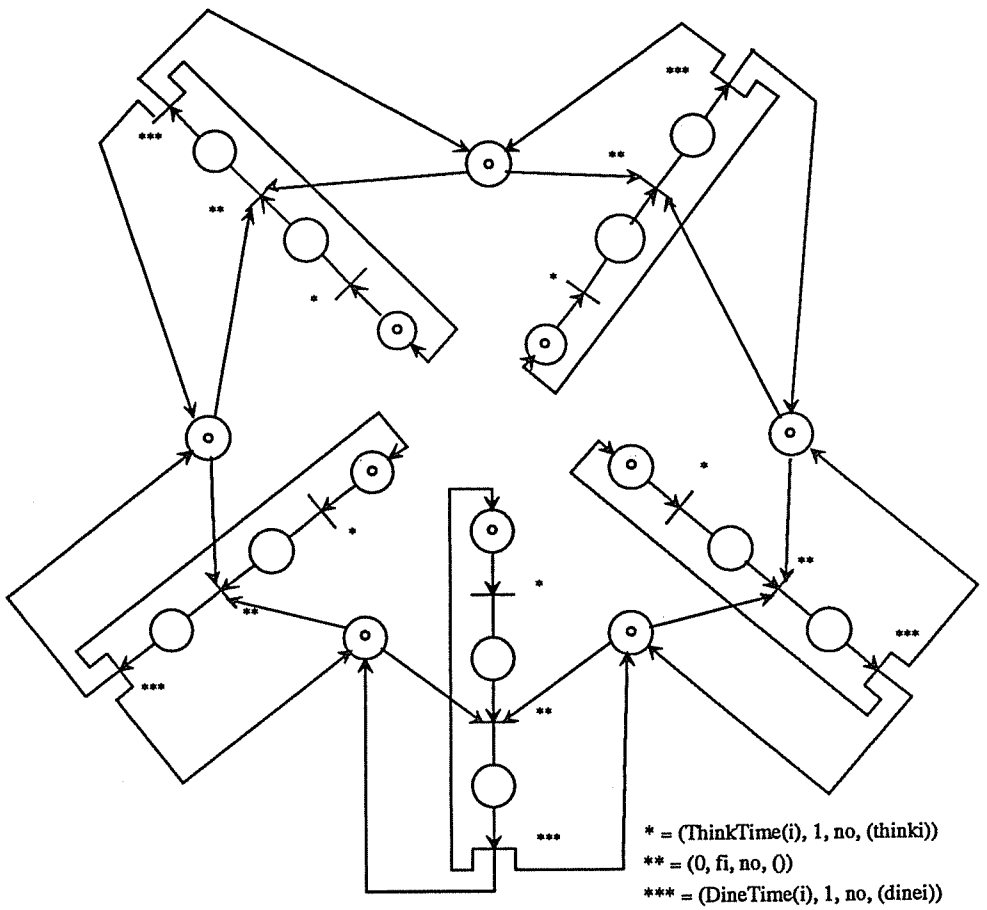

Figure 5.1: Dining Philosophers GTPN Model

A unique resource is associated with each thinking and dining transition. The GTPN analyzer will compute the long run expected usages of these resources, (ThinkFraction(i) and DineFraction(i)), which correspond to the long run fractions of time that philosopher i spends thinking and dining. From these measures, we can calculate the long run fraction of time philosopher i is idle, waiting for forks (IdleFraction(i) = 1 - ThinkFraction(i) - DineFraction(i)). Performance of the dining philosophers is maximized when time spent waiting for forks and expected time to complete the dinner are minimized.

The Dining Philosopher model can be analyzed quickly for various think times, dining times, and firing frequencies (i.e. relative aggressiveness in grabbing forks). We first consider the case of two classes of philosophers and deterministic think times. The first class of philosophers, formed by any two non-neighbors, thinks for N units of time and dines for N units of time. The second class of philosophers also dines for N units of time. In one experiment, we let N=3, and vary the duration of

the think time of the second class of philosophers between 1 and 12 units of time. The firing frequencies, f_i for both classes of philosophers in this experiment are set equal to one. Figure 5.2 and figure 5.3 show ThinkFraction(i), IdleFraction(i), and DineFraction(i) for the model. Each of the four data points, corresponding to ThinkTime(2) equal to 4, 7, 8, and 9, have two recurrent classes. In each case, both recurrent classes have absorption probabilities equal to 0.5. In each case, the resource usage estimates are the same for both recurrent classes. Each of three data points, corresponding to ThinkTime(2) equal to 5, 10, and 11, have four recurrent classes in the Markov Chain. In each case, all four recurrent classes have absorption probabilities equal to 0.25. For ThinkTime(2) equals 5 or 11, the resource usage estimates are also the same for all recurrent classes. For ThinkTime equal to 10, two recurrent classes have identical resource usage estimates which are distinct from the identical resource usage estimates of the other two recurrent classes. Both values are shown in figure 5.2 and figure 5.3.

Our first observation is that the fractions of time the philosophers spend thinking, waiting, and dining, vary in a complex way with the input parameters in this experiment. Reasoning about the behavior of the system for one parameter setting (i.e. when ThinkTime(2)=3), shows that after 9 units of time, the system reaches "steady state", in which two philosophers are thinking, two are dining, and one is waiting, (interchangeably), forever after. We note that this behavior is highly dependent on the relative delays in the model.

We investigated the complex behavior of system performance as a function of varying think times for the two classes of philosophers further. Assume that each philosopher requires R=60 units of time dining to complete the meal. Let MaxDineFraction be the maximum value of DineFraction(i) over all i. Then the expected time that the first philosopher(s) complete their meal is $Dmin = R/MaxDineFraction$, which we define to be the "end of the dinner". Figure 5.4 shows the total time spent thinking, idle, and dining, and the amount of time needed to complete any unfinished portions of the meal, for a few interesting parameter settings. Starting with a "baseline" model (ThinkTime(i) = DineTime(i) = N), in (a) of figure 5.4, we see that Dmin=2.5 hours, of which each class of philosopher spends 60 minutes (40%) eating, 60 minutes thinking, and 30 minutes waiting for forks. This corresponds to $ThinkTime(2) = 3$ in figure 5.2 and figure 5.3. All philosophers finish the meal at the same time. Note that two philosophers is the maximum number that can be dining at the same time, so the baseline model is optimum with respect to $DineFraction(i) = 0.4$. The question is whether the idle time for the philosophers can be reduced while still dining at full capacity. In (b) of figure 5.4, the second class of philosophers reduce their idle time by slightly increasing their think time by some amount $x, x < \frac{N}{2}$. In (c) of figure 5.4 the second class of three philosophers increase their think times to 3N, which reduces idle time to zero for all philosophers, but causes the three to miss half their meal. This corresponds to $ThinkTime(2) = 9$ in figure 5.2 and figure 5.3. In (d) of figure 5.4, both classes of philosophers have think times set to $ThinkTime(i) = 1.5DineTime(i) = 1.5N$, which represents the

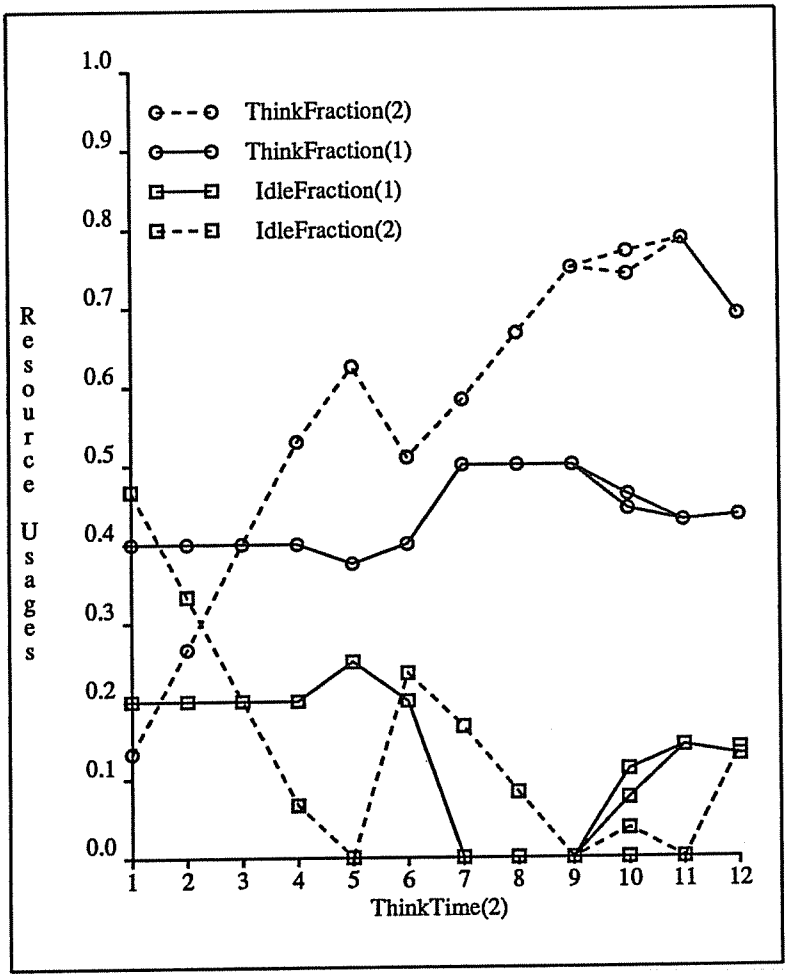

Figure 5.2: Performance Measures for varying deterministic think times. Two Classes of Philosophers. $ThinkTime(1) = 3 \quad \forall j \; f_j = 1, DineTime(j) = 3$. Idle and think times.

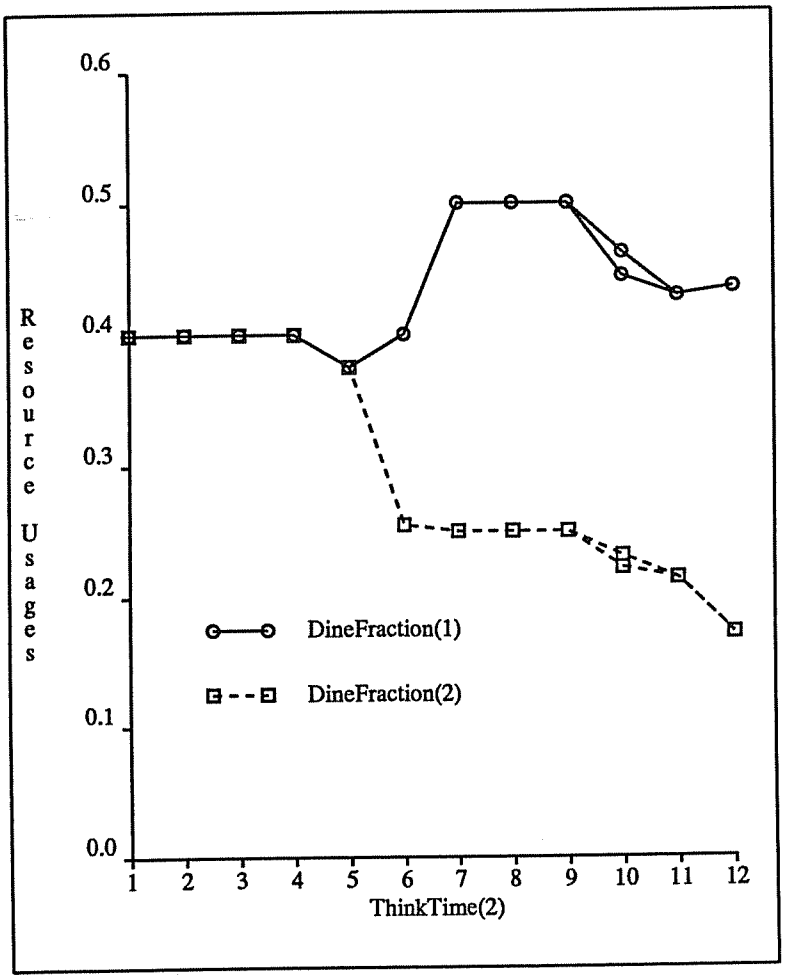

Figure 5.3: Performance Measures for varying deterministic think times. Two Classes of Philosophers. $ThinkTime(1) = 3 \quad \forall j \; f_j = 1, DineTime(j) = 3$. Dine time.

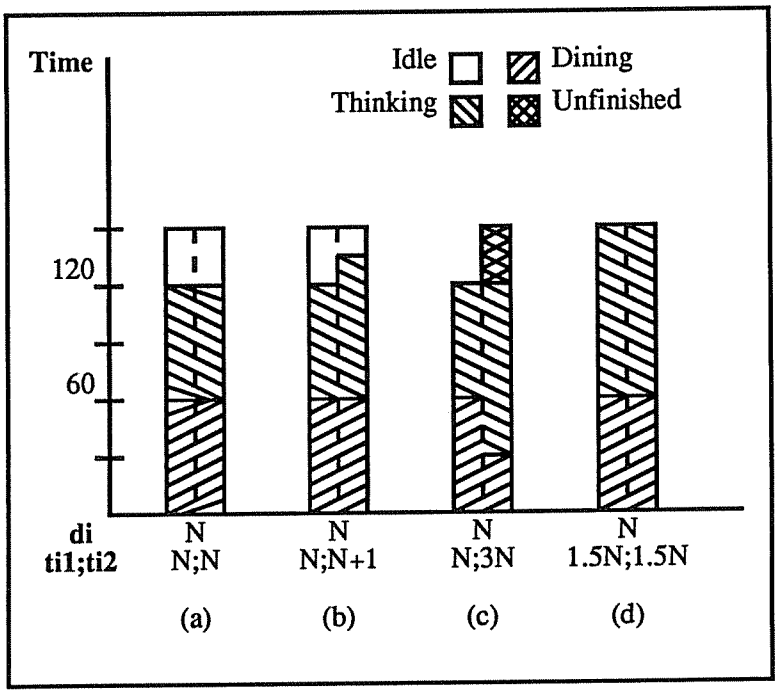

Figure 5.4: Performance for Selected Parameters

optimum behavior.

For the experiments above, we varied the think time while holding all dining times and firing frequencies constant. In the next experiment, we set the think time of the second class of philosophers to N, and vary the firing frequencies of the first class of philosophers. Figure 5.5 gives the results of these analyses. Note that the maximum possible value for DineFraction for the given parameter settings, is 50%. When the firing frequency for the aggressive philosophers is 5.0 the fraction of time they spend waiting for forks is reduced by 70% (to 0.06).

Finally, we repeated the first experiment (figure 5.2 and figure 5.3) with geometric think times. Figure 5.6 and figure 5.7 shows the performance estimates as a function of mean think time of the class two philosophers. The trends in the performance estimates as $ThinkTime(2)$ varies are qualitatively the same as in the deterministic model. However the performance curves are smooth, in contrast to the erratic variations in figure 5.2 and figure 5.3. The erratic performance in the deterministic models is due to cyclic dependencies. These dependencies also make multiple recurrent classes more likely in the deterministic models.

5.2 The CRAY-1

In order to evaluate architectural designs for highly parallel supercomputers, it is important to understand the behavior of the proposed designs under representative

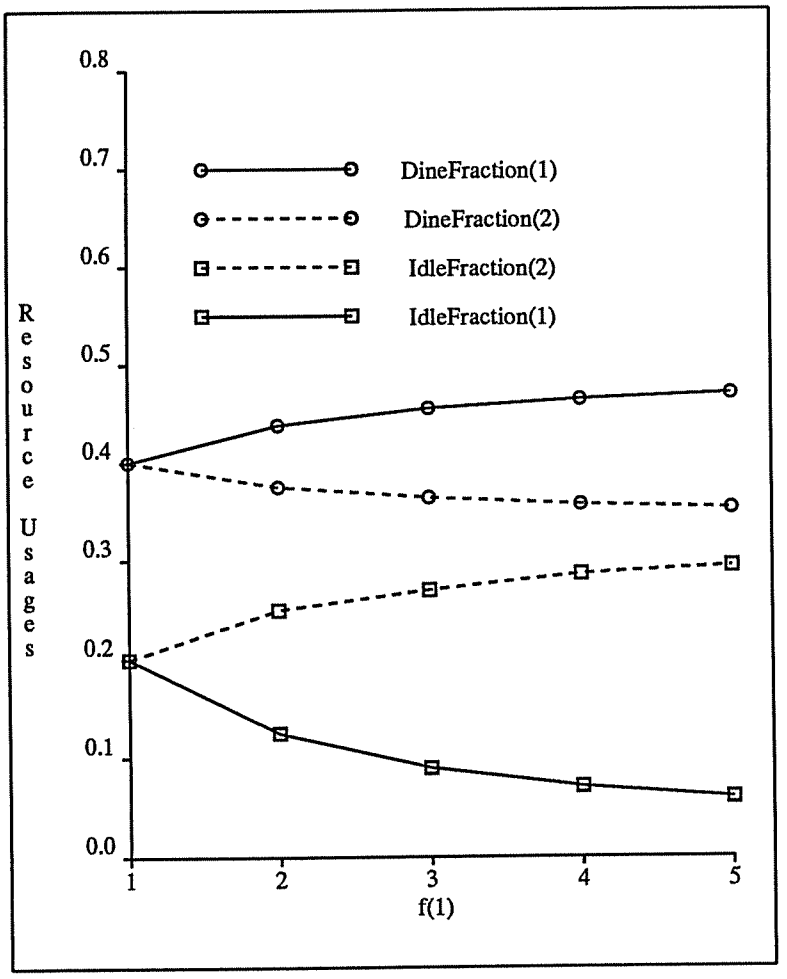

Figure 5.5: Performance with Aggressive Philosophers.

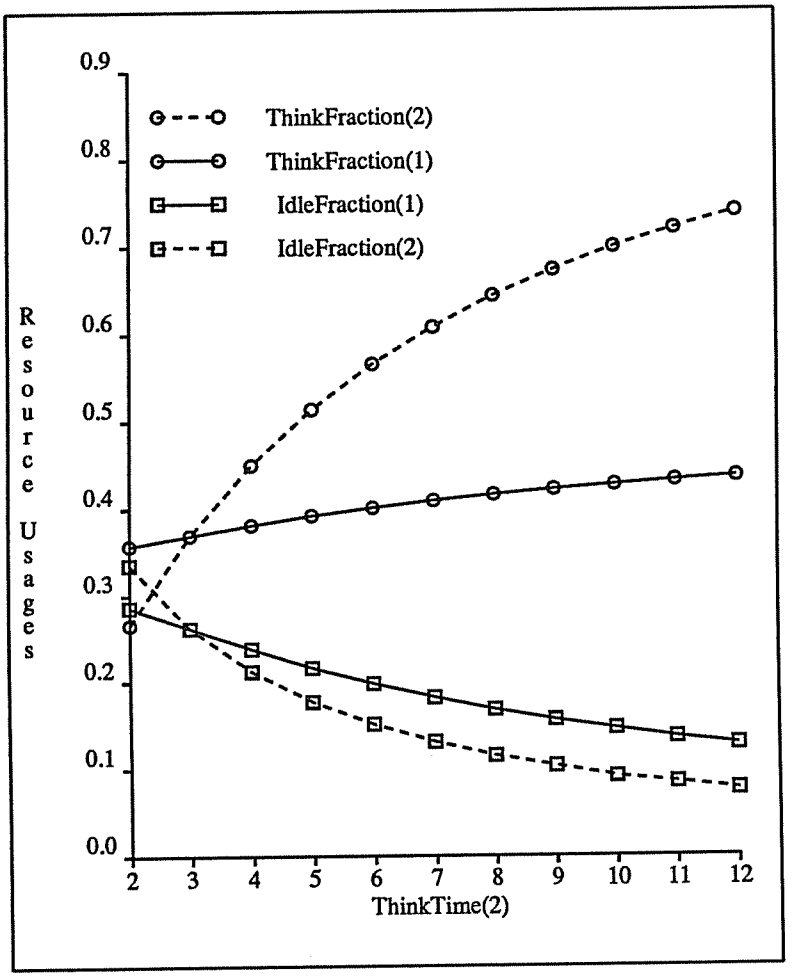

Figure 5.6: Varying geometric think times. Two Classes of Philosophers. $ThinkTime(1) = 3 \quad \forall j \; f_j = 1, DineTime(j) = 3$. Idle and think times.

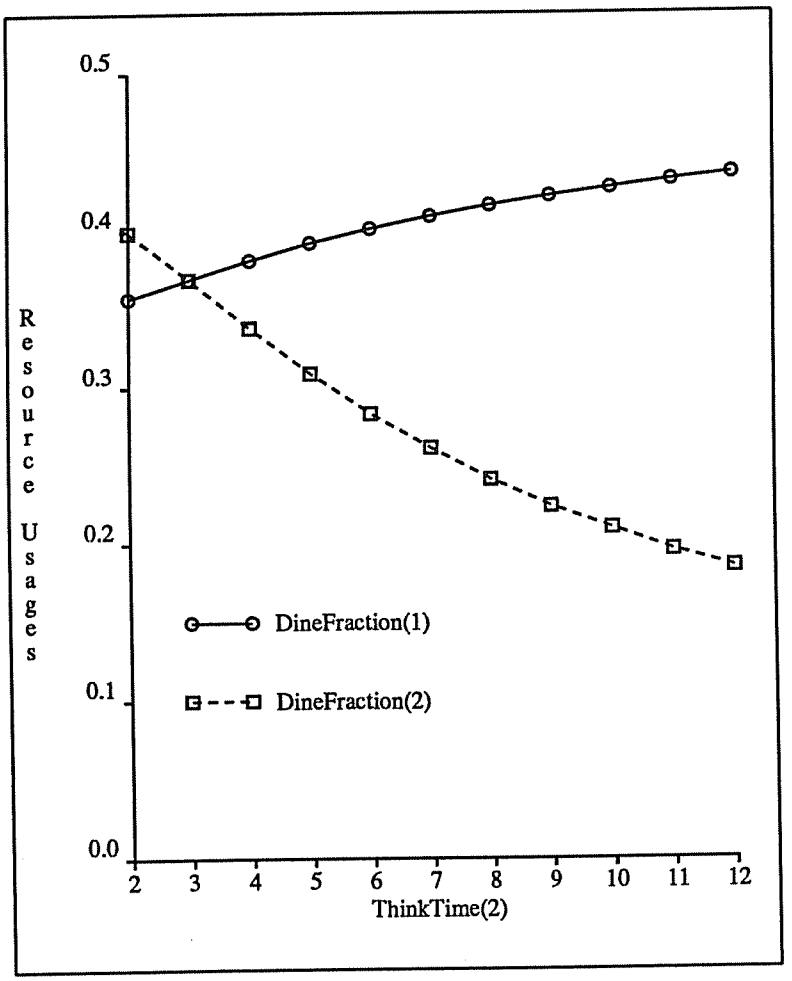

Figure 5.7: Varying geometric think times. Two Classes of Philosophers. $ThinkTime(1) = 3 \quad \forall j \; f_j = 1, DineTime(j) = 3$. Dine times.

workloads (i.e. benchmarks). The development of accurate analytical models for designs which can be parameterized for different workloads, would be a major step towards the goal of choosing a good design rapidly. One goal of our GTPN model is to aid in the creation of such analytical tools.

In this section we present some preliminary results on the use of the GTPN model for analyzing a subset of the CRAY-1^{TM} supercomputer. Our basic approach to performance modeling of pipelined machines is to specify deterministic delays for each pipelined functional unit, but to represent the instruction mix and resource demands for programs probabilistically. This approach closely parallels current methods for evaluating these machines using detailed simulation programs and system measurement. A well-known set of scientific benchmarks, the Lawrence Livermore Loops, and a detailed simulator for the scalar mode of the CRAY-1S [PAN83], were used in our study. We derive parameters for our model directly from execution traces of the benchmark programs, and validate predicted performance with simulation results.

5.2.1 The CRAY-1 Model

The CRAY-1S can operate in two modes: vector mode and scalar mode. For our initial experiment, we have investigated the scalar mode operation for one important type of instructions (i.e. the set of instructions that use the *S* registers). We assume the workload only contains the classes of instructions represented in the model. In the six loops studied this covers between 28% and 52% of the instructions in a program.

Our GTPN for this model is shown in Figure 5.8 [1]. The initial marking has one token on place *GetInst* and eight tokens on place *Regs* (representing the eight S registers which are initially not in use). Except for the first transition, labels on the transitions are only used to indicate firing durations which are greater than 0.

Each instruction conceptually goes through three stages of execution. In stage one, the instruction is *issued*, which takes a fixed number of clock periods (≥ 1) depending on the opcode. In stage two, the instruction *holds* until all of its resource needs are met. Resource needs may include source registers, a destination register, and a destination bus (i.e. the *S* bus). To simplify the presentation of the net, the portion of the net that models blocking for register resources is shown as a dotted rectangle in the figure. The label in the rectangle, (m,n), indicates that the instruction needs m source registers and n destination registers. In stage three, the instruction *executes*. Stage three takes a fixed number of clock periods (possibly zero), again depending on the opcode. When an instruction completes stage two, the next instruction can start stage one. Consequently, only one instruction can be in the first two stages, but many instructions may be in stage three.

The opcodes represented in the model are grouped into 6 *classes*, according to their issue times, resource needs, and execution times. The frequency expressions

[1] Places that have the same name in the drawing are the same place in the model.

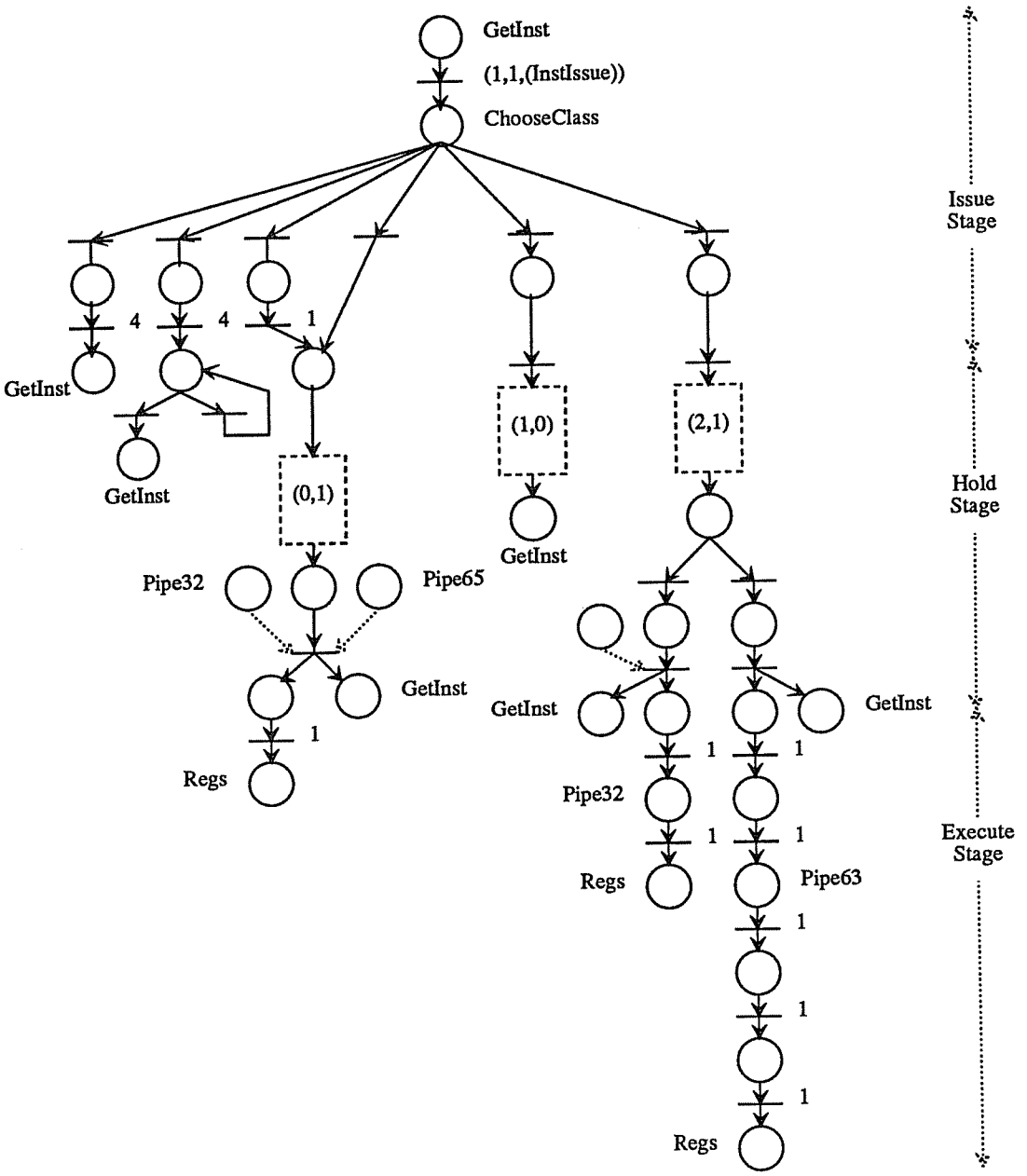

Figure 5.8: The CRAY-1 GTPN Model

for the 6-way branch in the instruction issue stage correspond to the frequency of occurrences of each instruction class in the workload.

Register contention is modeled (in the dotted rectangles) as a sequence of Bernoulli trials, with parameter P. The current model computes the probability P based on the number of tokens on *Regs*, assuming that register requests are independent and uniformly distributed. Note that the m source register tokens are immediately put back on *Regs* after the successful acquisition of registers, since these registers are only needed for the first clock period of the instruction's execution. An instruction can also block due to bus contention. This is modeled using inhibitor arcs bridging from one execution pipeline to another. For example, a token on *GetBus3* (sixth instruction class) indicates the instruction needs the S bus 3 clock cycles later. This token must wait if place *pipe63* contains a token.

The resource of interest for performance estimates in this model is *InstIssue*. The utilization estimate for InstIssue will represent the instruction issue rate (i.e. the number of instructions issued per clock period) for the model.

5.2.2 Experiments

We have run a preliminary experiment to compare the GTPN model estimates with the detailed performance simulation measures for six Lawrence Livermore Loop benchmarks. An execution trace for each benchmark was filtered to delete any instructions not represented in the GTPN model. Each trace was then analyzed to compute opcode frequencies and register usage frequencies. The opcode frequencies for each trace were used to parameterize the model. The GTPN analyzer was then used to compute the instruction issue rate. The execution traces were also run on the detailed performance simulator which also calculated the instruction issue rate. The level of agreement between the analytic and simulation results is a measure of the accuracy of the GTPN model.

Figure 5.9 shows the results for the above experiment. Several observations can be made. First, the GTPN model estimates are quite accurate for 3 of the 6 benchmarks. Second, the TPN model shows an absolute error as high as 0.2-0.25 for some benchmarks, as compared with the detailed simulator. Third, nevertheless, the model estimates are in error in a predictable way in the following sense. All of the GTPN predictions are optimistic (i.e. issue rates are too large). This can be explained by the fact that the model assumes independent uniform register usages. Statistical analyses of the benchmark execution traces indicate strong dependencies in the register access patterns, particularly for Loops 2 and 6. The register dependencies in the benchmarks accounts for larger "hold" times and lower instruction issue rates. Modeling dependencies in the selection of instruction class does not appreciably improve on the performance estimates. An encouraging sign is that the amount of error in the current model estimates is roughly proportional to the amount of dependency of register accesses computed from the benchmark traces.

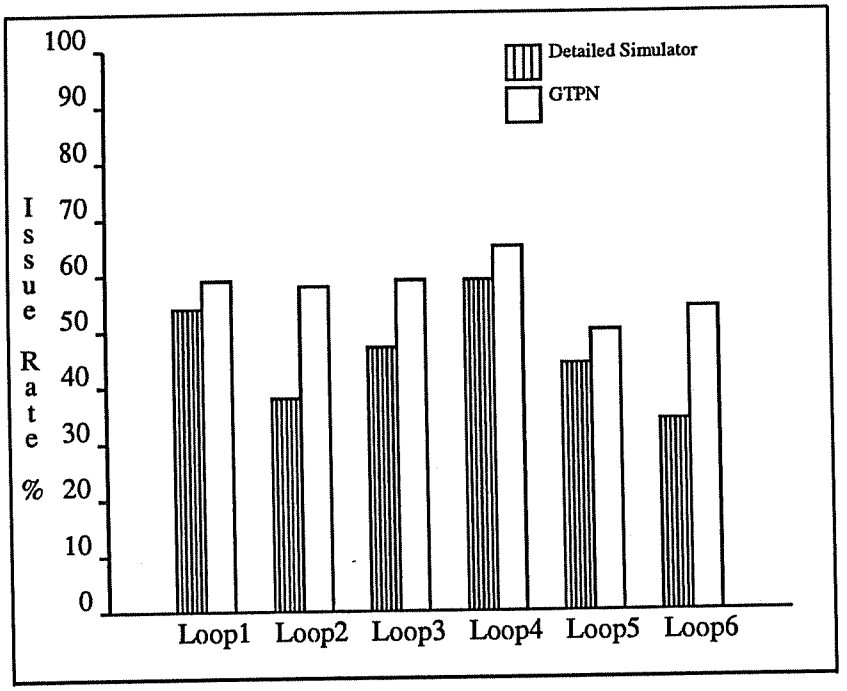

Figure 5.9: Instruction Issue Rate for the Loop Benchmarks

Chapter 6

Multiprocessor Memory and Bus Interference

Due to their potential for increased performance through parallelism, multiprocessor systems (systems with more than one processor and with shared memory) have been studied for many years. In this chapter we are interested in MIMD multiprocessors with multiple independent memory modules and with a single-stage multibus interconnection network. A key issue in such a system is the extent to which contention for the shared memory modules and the buses causes performance degradation. An extensive literature has developed addressing this issue with stochastic models [BHA75,GOY84,TOW83]. Until now, however, no one has used exact solution methods to derive performance measures for any model that contains four important and realistic properties. These properties are:

1. both bus and memory contention are considered
2. the amount of time spent actively accessing memory per request is a constant
3. the processing time between memory requests is variable and non-zero
4. the distribution of accesses across the memory modules is not necessarily uniform

We have developed a model which can be used to derive performance estimates for a system containing these properties. The solution method is based on the global state transition diagram (i.e. discrete time Markov Chains). Previous researchers [TOW83,YEN82] have concluded that this approach is computationally infeasible except for very small systems. We demonstrate that this approach is feasible for useful size systems if the Markov Chain is properly formulated. In particular,

we derive results for several models (with up to 16 processors and 16 memories) with the four properties listed above.

We do not mean to underestimate the importance of approximate solution techniques and simulation. As more functionality is encoded in a model, the Markov Chain generated using the GTPN analyzer can still become prohibitively large. Our point is that the GTPN allows symmetries to be discovered and used to reduce the model complexity. Thus, with careful model formulation, the Markov Chain does not become prohibitively large as soon as with earlier models. Because of the smaller state spaces, there is more opportunity for exact results.

By using the GTPN model we reached several new conclusions about the effect of memory and bus interference in multiprocessors. First, a widely used definition of the *processing power* of a multiprocessor does not accurately reflect the true increase in power of a multiprocessor over a single processor. Second, if the real system has a constant memory access time and any number of buses, then to assume that it has an exponentially distributed access time can lead to large errors in the estimation of the probability distributions of processing power. We introduce a new definition of processing power that does accurately reflect the increase in power. Third, with respect to speedup, we have identified the phenomenon of critical memory interrequest times in multibus systems. As long as the mean interrequest time is longer than the critical value, only a very few buses are needed to attain nearly the same performance as a crossbar.

In section 6.1 we describe the behavior of the multiprocessors we study and review the relevant previous work in stochastic modeling of these systems. In section 6.2 we present our multiprocessor model and results. Section 6.3 summarizes the important contributions of our work and suggests some directions for future research.

6.1 Background

6.1.1 Multiprocessor Characteristics

Figure 6.1 illustrates the multiprocessor systems we consider. The shared memory is divided into independent modules, each of which permits only one access at a time. The processors are connected to the memory modules through a single-stage multibus (in contrast to multiple-stage networks such as banyan networks).

The process associated with each processor can be in three states: running on its processor, waiting for a memory module, or accessing a memory module. The processing time between memory requests is the processor's *interrequest time*. In much of the literature we reference below, the interrequest time is assumed to be a geometric random variable. In this case, the parameter of that random variable is the *memory request probability*(MRP). The MRP is the probability that an actively executing processor will generate a request in the next memory cycle. A process

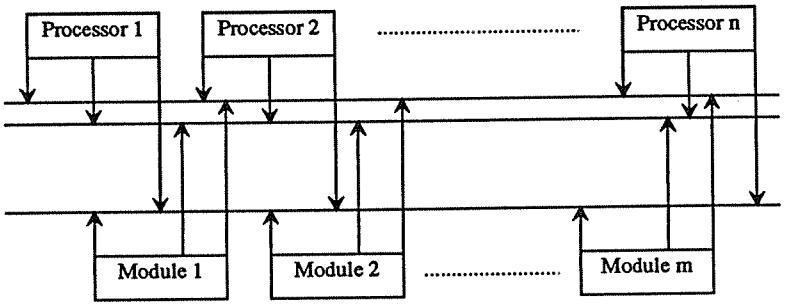

Figure 6.1: The Multiprocessor System

with an outstanding request blocks until it obtains an arbitrary bus and the desired memory module. The amount of time spent actively accessing memory per request is the *memory access time* which is a constant in our models. The distribution of accesses across the memory modules by a processor is that processor's *memory access probabilities.*

Memory utilization is the fraction of time that a memory module is being accessed by some process. *Processor utilization* is the fraction of time that a processor has its associated process running on it (versus accessing a memory or waiting for a memory). *Processor productivity* is the probability that a typical process is doing productive work (executing on its processor or accessing a memory, not waiting for a memory). Memory utilization summed over all memories is the expected number of busy memory modules. In the literature, this has often been called the *effective memory bandwidth.* Memory bandwidth and memory utilization, however, are not the same, so we will not use the term effective memory bandwidth. Processor utilization summed over all processors is sometimes called *processing power* [AJM84,AJM82a,AJM82b,AJM83]. Effective memory bandwidth and processing power (as defined above) are the main performance estimates obtained in the studies cited below.

6.1.2 Previous Results: Bhandarkar

Bhandarkar [BHA75] is an important early work. He uses discrete time Markov Chains to obtain performance estimates for up to a 16 processor/16 memory system. The key assumptions of his model are:

1. all of the processors are statistically identical
2. the memory access time (memory cycle time)is a constant
3. MRP is 1, i.e. a process is never actively executing on its own processor. When its current request is satisfied, it always generates another request at the start of the next cycle.
4. the interconnection network is a crossbar

5. requests made by each processor have an independent and equal probability of being directed to any one of the modules, i.e. *uniform memory access probabilities.*

In spite of these simplifying assumptions, the state space of his Markov Chain grows rapidly. For example, the 16 processors/16 memories system has 300,540,195 states. Bhandarkar did not attempt to solve the steady state equations for a system with 300,540,195 states. He solves the steady state equations for a much smaller state space that takes advantage of the fact that the processors are identical. However, the only way he was able to compute the transition probabilities in the smaller state space was by building and collapsing the larger state space.

Bhandarkar considered loosening assumption 3 so that the MRP is less than one, i.e. a processor might spend some time executing between requests. This would have implied a larger state space so he did not attempt an exact solution.

Most of the research in less restrictive stochastic models of multiprocessors since Bhandarkar's work has focused on approximate solution techniques. The studies are divided into categories according to which of the above assumptions (2, 3, 4, or 5) are relaxed.

6.1.3 Less Restrictive Models

Bhandarkar [BHA75], Strecker [STR70], and Wulf and Bell [WUL72] deal with the basic case of a constant memory access time, uniform access probabilities, a crossbar, and a MRP of 1. One logical change is to consider to what extent performance is degraded due to having fewer buses than in a crossbar. Towsley [TOW83] gives approximate solutions and simulation values for this case. Alternatively, a crossbar could be assumed and a MRP less than one considered. This is a reasonable change, because presumably each processor has some local memory or a cache that it is using for most of its memory activity. Baskett and Smith [BAS76], Rau [RAU79b], Yen, Patel, and Davidson [YEN82], and Towsley [TOW83] give approximate solutions for this case.

More recent studies combine these two changes, i.e. they have constant access time, uniform access, a MRP less than one, and multibuses. Lang, Valero, and Alegre [LAN82] provide simulation results. Bhuyan [BHU84], and Mudge, Hayes, Buzzard, and Winsor [MUD84], Towsley [TOW83], and Goyal and Agerwala [GOY84] give approximate solutions.

Some studies assume an exponentially distributed memory access time, an exponentially distributed interrequest time, and use continuous time Markov Chains in the solution. Approximate solutions of these models are given by Bhandarkar and Fuller [BHA73], Marsan and Gerla [AJM82a], Marsan, Balbo, and Conte [AJM82b], Marsan, Balbo, Conte, and Gregoretti [AJM83], Önyüksel and Irani [ONY83], and Jacobson and Lazowska [JAC82]. Exact solutions are in Irani and Önyüksel [IRA84], Molloy [MOL81], and Marsan, Balbo, and Conte [AJM84]. Several of these studies

[MOL81,AJM83,AJM84] are of special interest because they use a form of Petri Nets, called Stochastic Petri Nets, to derive their continuous time Markov Chains. We defer further discussion until section 6.2. Mudge and Al-Sadoun [MUD84] provides an approximate solution that allows the memory access time to be any discrete time random variable that has first and second moments.

The last group of studies consider nonuniform access probabilities. All assume constant cycle time, and a crossbar. Those that only allow a MRP of one are Sethi and Deo [SET79] and Du and Baer [DU83]. The papers that allow memory request probabilities less than one are: Hoogendoorn [HOO77], Mudge and Makrucki [MUD82], Siomalas and Bowen [SIO83], Towsley [TOW83], and Bhuyan [BHU84]. All the solutions are approximate except one of the ones given in Du and Baer. The exact solution method in Du and Baer is a modification of Bhandarkar's exact method. Perhaps this is why they only consider a crossbar and a MRP of 1.

In section 6.2 we develop a GTPN model of multiprocessors which can be modified easily (primarily by changing a few parameters) to reflect the various assumptions made in the above studies. We will compare the performance estimates obtained from exact analysis of the GTPN with some of the results cited in this section. We will also use the GTPN model to obtain results not previously reported.

6.2 Multiprocessor Analysis

We have analyzed the performance of multiprocessor systems using the GTPN. In this section we define our performance measure of speedup, describe the GTPN model used, and summarize the results of our experiments. We modeled several systems that have been studied previously for the sake of comparison. First, we conducted four sets of validation experiments. Second, we compared our results with those of Marsan, Balbo, and Conte [AJM84] which assume exponential access time. The important measure of *speedup* has not been studied in the stochastic modeling literature cited in section 6.1. Consequently, our third set of experiments look at speedup for some representative systems. Finally, we examine the performance of a particular class of non-uniform access probabilities called *favorite memory*.

6.2.1 Measures and the GTPN net

Recall that previous evaluations of multiprocessors using stochastic models have studied the expected number of busy memory modules and a measure of processing power defined as: processor utilization summed over all processors. We are more interested in a different measure of processing power: processor productivity summed over all processors. To avoid confusion in the discussion below, we will call our measure *speedup*, since it is the same as the speedup measure used in the non-stochastic literature on multiprocessors. We will use the term processing power

in the sense defined in previous studies. We argue that speedup is a more important single measure of system performance because the goal of multiprocessing is speeding up a program, not achieving high memory or processor utilization. The expected number of busy memory modules and processing power are also easily computed for our GTPN models as we show below.

The GTPN model used in the analysis assuming uniform access probabilities and a multibus is shown in Figure 6.2 and Table 6.1. The net for the non-uniform access case is a slight modification of this one. The net shown is for a system with three processors, two memories, and one bus (P2). It is the model in Marsan, Balbo, and Conte [AJM84], modified to support discrete time.

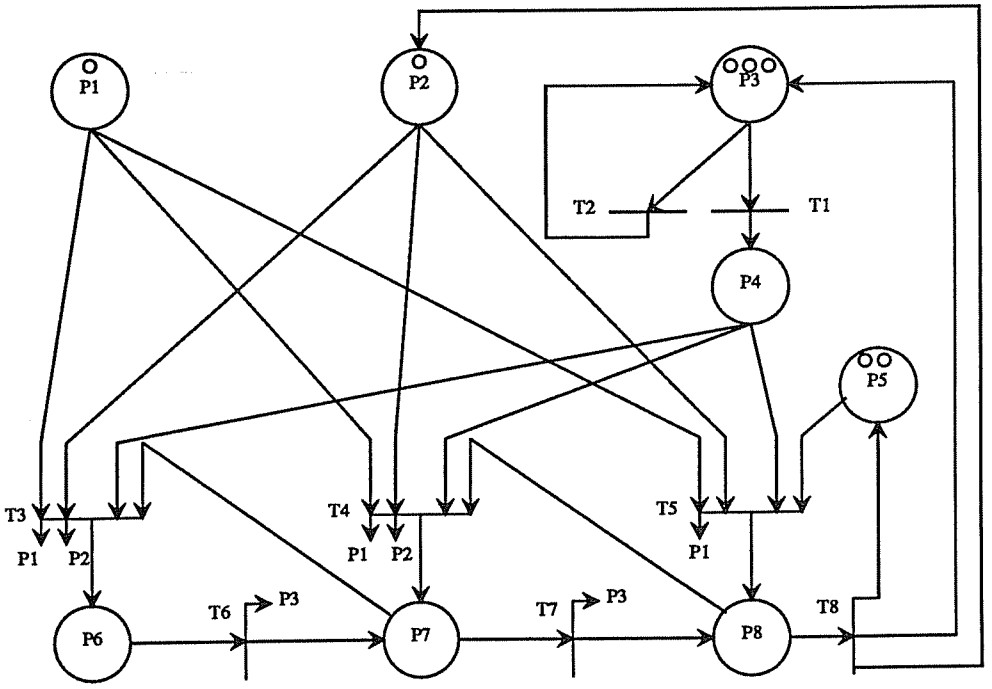

Figure 6.2: GTPN net for a 3 processor/2 memory/1 bus system. Uniform access.

Transition	Duration	Frequency	Cnt Combs	Resources
T1	0.0	MRP	yes	(Spd,PP)
T2	1.0	1 - MRP	yes	()
T3	0.0	P7/3	no	()
T4	0.0	P8/3	no	()
T5	0.0	P5/3	no	()
T6	1.0	(((P2 = 0) | (P4 = 0) | (P5 = 0))	no	(Spd, MemBusy)
		& (P3 = 0) & (T1 = 0)) * 1.0		
T7	1.0	same as T6	no	(Spd, MemBusy)
T8	1.0	same as T6	no	(Spd, MemBusy)

Table 6.1: The attributes of each transition in the multiprocessor net.

The tokens are shown in the initial state. The tokens in P5 represent free memories. The tokens in P3 represent processors that are active locally. The tokens in P2 represent free buses. Transitions $T1$ and $T2$ implement a geometric processing time between memory requests. (Note that the GTPN can represent geometric, as well as constant, holding times.) With probability equal to the memory request probability, each token in P3 moves to P4. A token in P4 represents a processor making a memory request. The places along the bottom represent lengths of memory queues. Because the memory modules are statistically identical we are just interested in the possible combinations of queue lengths. For example, a token in the leftmost place signifies that all three processors are waiting for the same memory module. In this case, there can be no other tokens along the bottom places. We want each token on P4 to have its memory request uniformly distributed among the memory modules. If only one transition can start firing at a time, then the frequency expressions for $T3$, $T4$, and $T5$ ensure uniformity. The place $P1$ is used to enforce that only one token on P4 at a time moves to the bottom row (with zero delay). As tokens move across the bottom, processors have their memory requests granted and return to P3. The last processor to use a memory module (T8) returns the bus token to P2.

The frequency expressions for the transitions along the bottom enforce that none of the transitions along the bottom row start firing until all possible tokens on P4 are moved into the memory subsystem. In these frequency expressions a vertical bar represents a logical or. When the number of tokens on $P2$ or $P4$ or $P5$ is zero and the number of tokens on $P3$ is zero, and there are no firings of $T1$, the expression evaluates to one, otherwise it evaluates to zero. The Cnt Combinations column of Table 6.1 contains the value of the flag that determines how probabilites for maximals are to be calculated for each transition that may appear in a maximal. Three resources are used to derive performance measures. *Spd* generates speedup. *PP* generates processing power. *MemBusy* generates the expected number of busy memory modules.

6.2.2 Model Validations

Processors	Memories	Bhandarkar	GTPN
2	2	1.5000	1.5000
4	4	2.6210	2.6210
6	6	3.7809	3.7809
8	8	4.9471	4.9471
10	10	—	6.1150
12	12	—	7.2835
14	14	—	8.4527
16	16	—	9.6225

Table 6.2: Expected number of busy memory modules. Crossbar. MRP = 1.

Buses	Towsley Approx	Towsley Sim	Exact
8	7.93	7.96	7.977
9	8.73	8.80	8.825
10	9.27	9.34	9.357
11	9.53	9.58	9.566
16	9.62	9.66	9.623

Table 6.3: Expected number of busy memory modules. 16 processor/16 memories/Multibus. MRP = 1.

We first consider four previous studies that assume constant cycle time and uniform access, in order to validate our model. Bhandarkar [BHA75] gives exact numerical results for the expected number of busy memory modules up to a 8 processor/8 memory/crossbar and memory request probability (MRP) of 1. In Table 6.2 we present his numbers and the results from our GTPN; they agree.

The GTPN model for 16 processors and 16 memories yielded a Markov Chain with 8115 states. Bhandarkar's approach yielded a Markov Chain with 300,540,195 states. As mentioned in section 6.1.2, Bhandarkar needed this large state space as a means of indirectly reaching a smaller state space. The GTPN allows us to describe the system such that we can directly derive the smaller state space. This explains the difference in state space sizes. A similar direct method was used by the GSPN [AJM84]. We feel that it is quite likely that deriving the smaller state space without the GTPN is possible. Our point is that the GTPN aids in seeing and expressing the symmetry which allows the direct derivation.

We use the Power Method, an iterative sparse matrix algorithm, to solve for our results. The iterations terminate when the sum over all states of the absolute value of the difference between the last two iterations, is less than the convergence criterion. With our default convergence criterion, 5×10^{-5}, all of our values agreed with Bhandarkar's except in the 8 processor/8 memory case, where we reached 4.9469. We repeated the analysis with a smaller convergence criterion, 5×10^{-6}, and reached Bhandarkar's value. We have used our default convergence criterion in all of the other experiments reported in this paper. The default should be accurate to at least three digits. We round all of our remaining results to three digits. This should also be sufficiently accurate since all of the remaining comparisons are to simulations and approximate solutions.

Towsley [TOW83] gives approximate solutions and simulation values for the expected number of busy memory modules for a 16 processor/16 memory system with a MRP of one and a multibus interconnect. Our exact results are compared with his results in Table 6.3. Our values are in each case within the 90% confidence interval of his simulation, and provide further evidence that the approximate analysis is accurate. A reasonable conclusion is that up to 6 buses can be removed from the crossbar with only a small performance degradation. Lang, Valero, and Alegre is another study that arrives at a similar conclusion.

Buses	Lang Sim	Bhuyan Approx	Mudge Approx	Exact
1	1.00	1.00	0.98	1.000
2	2.00	1.97	1.88	2.000
3	2.87	2.79	2.57	2.898
4	3.33	3.27	2.99	3.352
5	3.45	3.44	3.16	3.458
6	3.47	3.47	3.22	3.469
7	3.47	3.47	3.23	3.469
8	3.47	3.47	3.23	3.469

Table 6.4: Expected number of busy memory modules. 8 processor/8 memory. MRP = 0.5.

Buses	Mean IRT	Goyal Sim	Goyal Approx	Exact
1	8	1.00	0.9997	0.9998
	16	0.8584	0.8719	0.8588
	32	0.4794	0.4811	0.4745
2	4	2.0000	2.0000	1.9983
	8	1.6616	1.6655	1.6560
	16	0.9316	0.9331	0.9365
	32	0.4852	0.4847	0.4793

Table 6.5: Expected number of busy memory modules. 16 processor/16 memory.

We now present two validations that involve a MRP of less than one and a multibus. First, we consider a 8 processor/8 memory system with a MRP of 0.5. Expected number of busy memory modules is the measure reported in previous papers. Table 6.4 gives the simulation values in Lang, Valero, and Alegre [LAN82], the approximate values of Bhuyan [BHU84], the approximate values of Mudge, Hayes, Buzzard, and Winsor [MUD84], and our exact results. Our exact values are within the 99% confidence intervals of the simulation results in all cases, and provide still better values for evaluation of approximate results. Note that, again, the number of buses can be reduced substantially from a crossbar with minimal effect on performance.

Second, we consider a 16 processor/16 memory system with one or two buses. In Table 6.5 we show Goyal and Agerwala's [GOY84] values and ours. In this table we adopt their convention of using the mean interrequest time (mean $IRT = \frac{1}{MRP} - 1$) instead of the memory request probability. Our values and theirs agree within the range of statistical error.

6.2.3 Comparison with Exponential Memory Access Time Models

Recall from section 6.1 that several studies have assumed an exponential memory access time and have derived exact performance estimates using continuous time Markov Chains. There are two possible reasons for the exponential assumption.

One, is that the multiprocessor under study has an exponential memory access time. Two, is that the multiprocessor under study has a constant memory access time, but that assuming an exponential memory access time is a reasonable approximation which yields models that can be solved exactly. In this paper we are interested in the stochastic modeling of multiprocessors with constant memory access times. Consequently, we are interested in the second reason. One would expect from queueing theory (as noted in Marsan and Gerla [AJM82a]) that the model with constant access time will give higher predictions for speedup. We conducted several experiments to see how large a difference the assumption of exponential access time makes.

Marsan, Balbo, and Conte [AJM84] gives exact results for a 12 processor/2 bus system. They vary the number of memories and the *load*. They assume that the interrequest time is exponentially distributed with rate λ and the memory access time is exponentially distributed with rate μ. The load is the ratio, ρ, of λ to μ. Our approach can be compared to theirs. We assume a constant memory access time and an interrequest time which is, strictly speaking, a modified geometric random variable. The important step is to make our models as similar as possible, so that only the difference in modeling the memory access time is observed. In particular, we need to represent the interrequest time distribution accurately.

In the limit, as the time length of a trial goes to zero, a modified geometric random variable is identical with the exponential random variable with same mean. Consequently, if trials are "reasonably frequent", then a modified geometric random variable is a good approximation to the exponential random variable with the same mean. We can approximate the exponential memory interrequest time arbitrarily closely in our GTPN model, by decreasing the duration of transition T2 and adjusting the frequency expressions for transitions T1 and T2 appropriately. Furthermore, for a selected duration of transition T2 (greater than zero), the variance of the modified geometric distribution is larger than the variance of the exponential distribution we are approximating. The increased contention due to this larger variability will result in lower estimates of processing power than if the exponential interrequest time were represented exactly. Since we expect (and observe) that constant memory access time model will have a higher processing power than the exponential memory access time, the differences we observe due to approximating the interrequest time will be conservative. (We verified this experimentally.)

In each of the experiments we conducted, a memory access time of 1 and an interrequest time of $1/\rho$ approximated using a duration of 1 for T2, yields a reasonably accurate representation. We note that this selection of parameters was used successfully in the validation against Goyal and Agerwala (in Table 6.5). Goyal and Agerwala's simulation assumed that the interrequest time is exponentially distributed.

Figure 6.3 shows their estimates of processing power and ours, as the number of memories is varied, for a 12 processor/2 bus system with load of 0.3. The difference in estimates is 6% at 2 memories and decreases to 2% at 10 memories. Thus, the

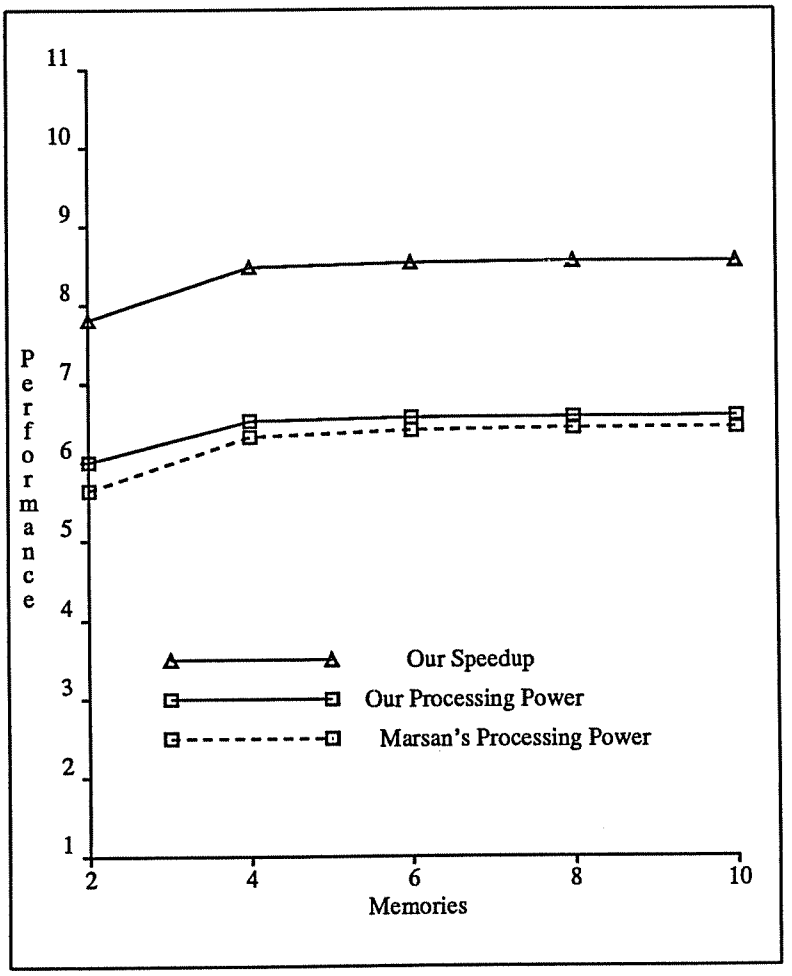

Figure 6.3: Expo. access time vs. constant cycle time. 12 processors/2 buses. Load is 0.3. MRP = 0.231.

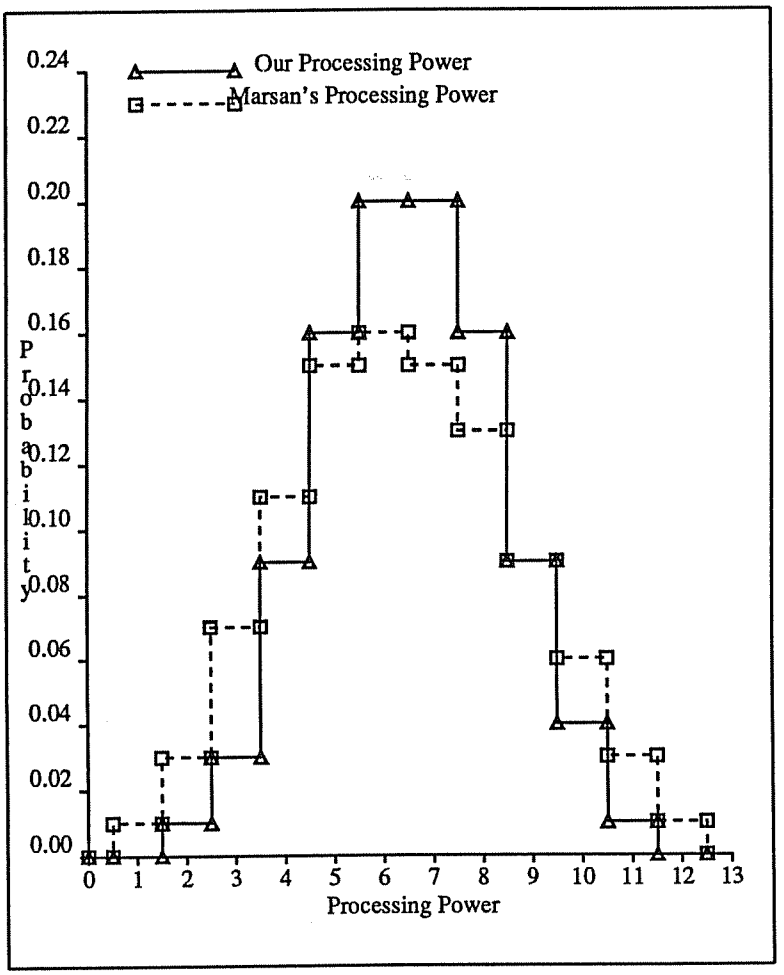

Figure 6.4: Probability Distributions for the 6 memory case

exponential access time assumption underestimates but provides a good approximation of the expected value for processing power. Our estimates for *speedup* are also shown. These results are qualitatively similar, but substantially larger than the processing power results. If we are really interested in speedup, processing power is a poor approximation.

Though expected values are important, the nature of the probability distribution of processing power is useful in characterizing multiprocessor behavior. In Figure 6.4 we show the constant access time and approximating exponential access time probability distributions for a 12 processor/2 bus/6 memory system with load of 0.3. The distributions are substantially different. As one might expect, the distribution assuming an exponential access time has a higher variance.

Marsan and Chiola [AJM85a], concurrently with our work, have introduced deterministic firing times into the GSPN under certain restricted conditions. Those restricted conditions imply that their multiprocessor models can only allow one bus. They reach conclusions similar to ours for the one bus case. Thus, our work may be viewed as a generalization of theirs to the case of an arbitrary number of buses.

6.2.4 Critical Memory Request Probability

We now describe the analyses we conducted that are not comparisons with previous studies. Our first set of experiments measured *speedup* for a 10 processor/10 memory system. The memory request probability is varied from 0.1 to 1.0. The number of buses is 1, 2, 3, 4, and 10. Our results in Figure 6.5 suggest an important conclusion about the effect of the number of buses on speedup. When the number of buses is small, a *critical memory request probability* appears to exist. The horizontal line drawn at speedup = 8.75 indicates approximately where this critical MRP lies on each curve. Below that probability, speedup is close to that with a crossbar(even for just two buses). Above that probability, speedup rapidly decreases and is equal to the number of buses in the limiting case. This rapid decrease is clearly due to the lack of buses. The drop is more gradual as the number of buses increases and is to a larger and larger extent due to memory contention instead of bus contention. We note that a functional relationship may exist between the number of processors, memory modules, and buses, and the critical MRP. Further study is required to determine whether this is true.

Our results are more specific than the conclusion reached by Lang, Valero, and Alegre. With respect to the measure of expected number of busy memory modules, they concluded that good performance is possible with the number of buses equal to one half the number of processors with a MRP of 0.5. Note that their conclusion is supported by Figure 6.5. Furthermore, we conclude that as long as the memory request probability stays below the critical value, only a few buses are needed to have close to the performance of a crossbar.

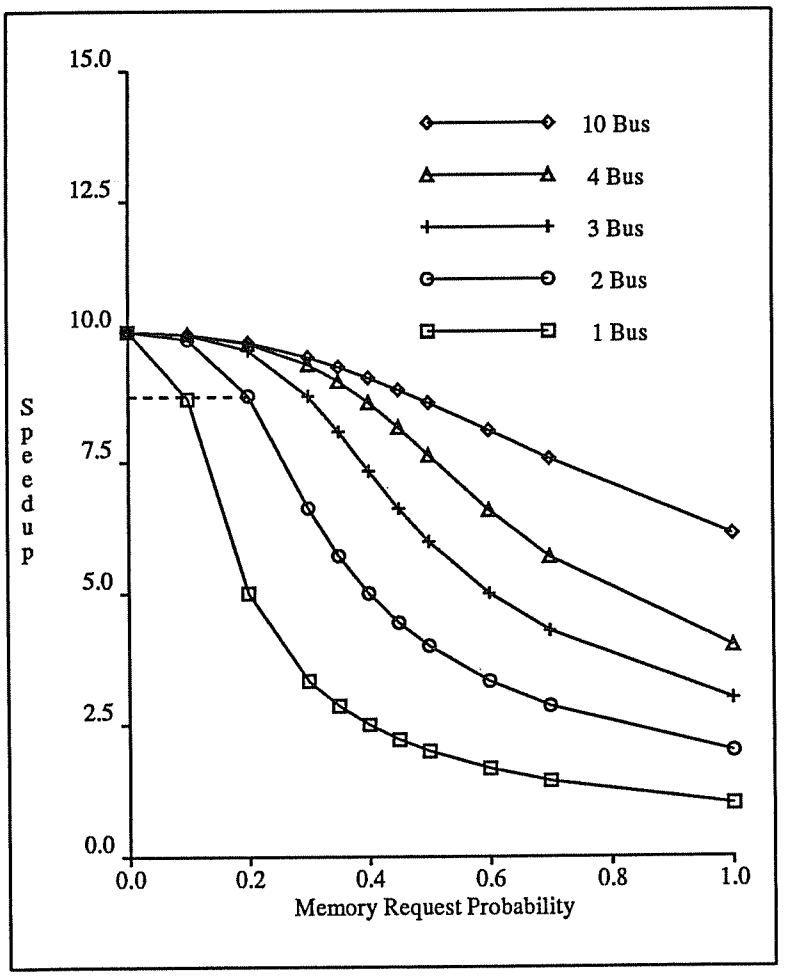

Figure 6.5: Measure is speedup. 10 processor/10 memory. Uniform access.

6.2.5 Non-Uniform Access Probabilities

All of the experiments above assume uniform access probabilities. Many authors have argued that this assumption is reasonable if the memory modules presumably are interleaved by the low-order bits of the memory addresses. Rau's [RAU79a] trace driven simulations, however, show that, at least in some cases, even with memory module interleaving, accesses are not uniform. Consequently, several studies have considered the non-uniform case. One version of non-uniformity that is of interest is called *favorite memory*. In favorite memory, there is one memory module, say module i, that is accessed with a different frequency than the other modules by all processes. Module i has probability, α, of being accessed while the probability of each other module being accessed is uniformly distributed over $1 - \alpha$, for all processors.

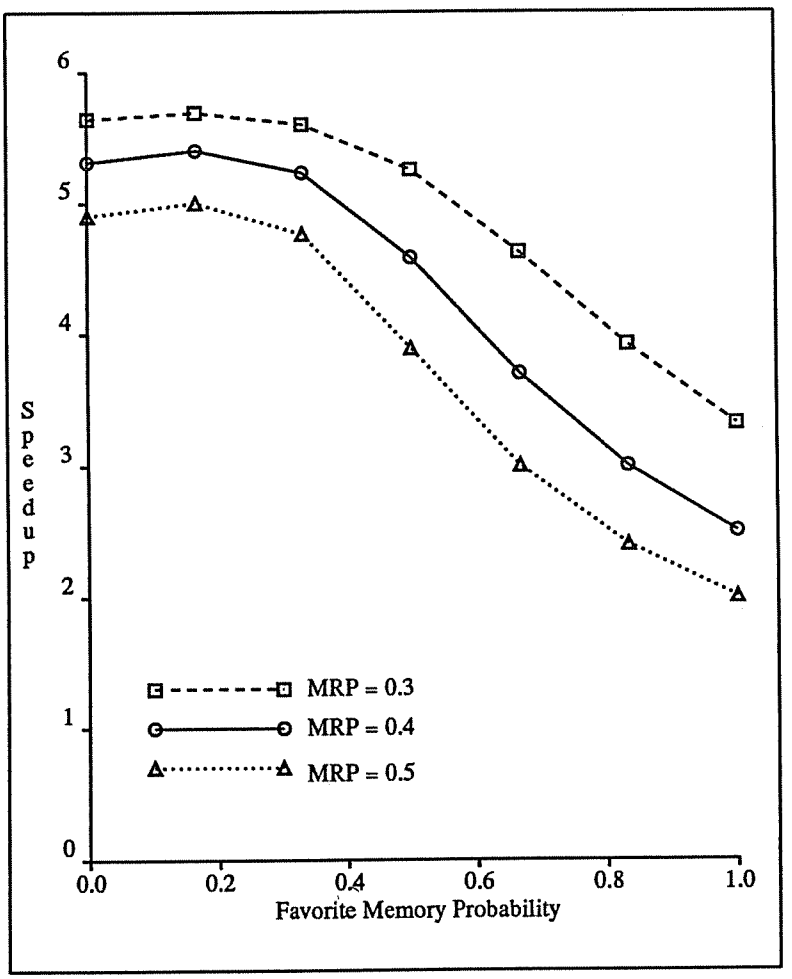

Figure 6.6: Favorite Memory nonuniform accesses. 6 processor/6 memory/3 bus.

We conducted an experiment assuming a favorite memory. We considered a system with 6 processors, 6 memories, and 3 buses. Results are given in Figure 6.6

for memory request probabilities of 0.3, 0.4, and 0.5. Each curve has seven data points for when zero, one sixth, two sixths, up to six sixths of the memory requests are directed to the favorite memory. Note that a modest favoritism (i.e. two sixths) has only a small effect on speedup. As expected, the speedup decreases as the favoritism increases and as the memory request probability increases. In addition, as the memory request probability increases the importance of favoritism increases, causing speedup to decrease more rapidly.

This favorite memory experiment illustrates the need to develop approximate solution techniques based on the GTPN. Identifying one memory module as favorite causes a significantly larger state space than when all the modules are identical. For example, this 6 processor/6 memory/3 bus system has 3384 states while a 6 processor/6 memory/3 bus system without a favorite memory has only 496 states.

6.3 Conclusion

We have presented exact performance estimates for models of multiprocessors for which only approximate and simulations estimates existed. These models include the important properties of constant memory access time, memory request probabilities less than one, and bus contention. One form of non-uniformity in the memory access probabilities was also treated. We derived these results by using Generalized Timed Petri Nets(GTPN). The GTPN is efficient for moderate size state spaces. For example, a multiprocessor model with 12 processors, 10 memories, 2 buses, and a geometric interrequest time of 5 time units has 2026 reachable states and requires 274 seconds to build the reachability graph and analyze it for performance estimates. The results we have derived illustrate the advantages of the GTPN model in specifying instantaneous, constant, and geometric holding times in analytical system models. If constant delays are not needed, or if they satisfy restrictions in current SPN models, then the SPN models may be more advantageous due to smaller state spaces.

The previous stochastic modeling studies of multiprocessor memory and bus interference have measured expected number of busy memory modules, and processing power as defined by: processor utilization times the number of processors. We suggest a better measure of processing power which is equivalent to the measure of speedup that is commonly used in other bodies of literature on multiprocessors.

Our multiprocessor performance estimates provided several important insights. One is that assuming an exponential access time for a model of a multiprocessor with constant memory access time and any number of buses causes only a small underestimation of the expected value of processing power. However, the probability distributions for processing power differ substantially. The distribution assuming an exponential access time has a higher variance.

Two, is that at low request rates only a few buses are needed to have almost the performance of a crossbar. However, when only a few buses are used, a critical

request rate exists. Exceeding that critical request rate causes a dramatic collapse in performance.

Chapter 7

Cache Consistency Protocols

7.1 Introduction

MIMD multiprocessors with multiple independent memory modules and a single stage multibus interconnection network form an important class of architectures. To minimize contention for the shared global memory modules and the buses, the processors are usually assumed to also have local memories. If the local memories are used as caches, the problem of maintaining consistency among the caches immediately arises [SMI82]. Two classes of dynamic cache consistency protocols have developed. The first class allows a general interconnection network but requires that a global directory be maintained by the shared memory modules. The second class requires that the interconnection network be a shared bus, but cache consistency is maintained in a distributed manner by the caches. We are interested in comparing the performance of protocols in this second class: the *shared bus cache consistency protocols.*

Several shared bus cache consistency protocols have recently been proposed [GOO83,FRA84,MCC84,PAP84,RUD84,KAT85]. The Write-Once protocol, designed for the Multibus(TM), was the first protocol to appear. Other proposals since then have suggested modifications to the Write-Once protocol which may improve system performance. Several of the proposals contain some analysis of the expected performance gains for the proposed protocol. However, it is not clear which of the modifications within a protocol are primarily responsible for the performance improvement. In some cases, the proposed modification requires a more complex (i.e. more expensive) bus or cache controller. A study is needed to determine how much increase in performance can be expected for each proposed modification. Archibald and Baer's simulation study [ARC85] is the most comprehensive performance comparison of the published protocols to date. In particular, they provide a uniform description of the protocols and they identify the important differences between

the protocols. We propose in this chapter a more precise model of the protocols which improves the relevance of the performance comparison. Furthermore, instead of modeling the protocols proposed in the literature exactly, we isolate four key enhancements to Write-Once which have been combined in various ways in these protocols. We then study the performance gains for each of the four enhancements.

In section 7.2 we describe our assumptions about the general characteristics of the multiprocessor. In section 7.3 we describe a Basic protocol similar to Write-Once, four enhancements to the Basic protocol, and how these enhancements are combined in protocols proposed in the literature. In section 7.4 we describe our GTPN models of the Basic and enhanced protocols. In section 7.5 we present our results, and in section 7.6 we summarize our work.

7.2 Multiprocessor Characteristics

Figure 7.1 illustrates the multiprocessor configuration. Each processor has one local cache. A single shared bus connects the caches and the main memory. A processor is directly connected only with its local cache. Each local cache is directly connected to its own processor and, through the shared bus, with main memory and all the other caches.

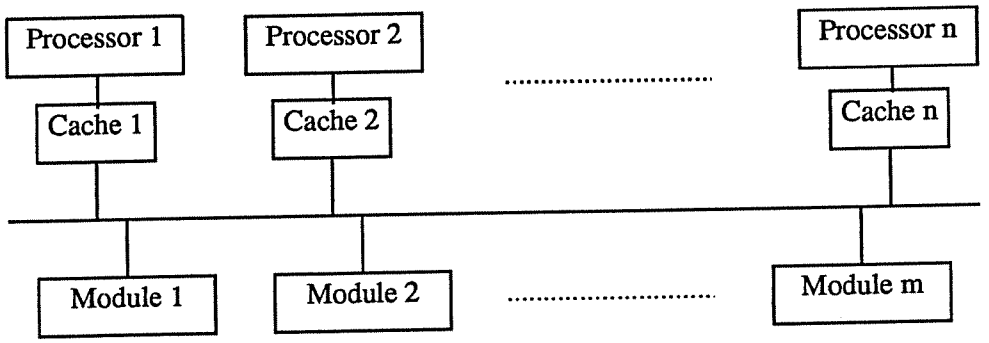

Figure 7.1: The Multiprocessor Configuration

The cycle times of the processors, caches, and the bus are the same and define the basic time unit. The main memory cycle time is four of these basic time units. Below the term *cycle* refers to the basic time unit.

7.2.1 Operation of the Memory and Bus

The cache memories and main memory are divided into *words* which are organized into *blocks*. A cache entry is a physical location within cache memory that consists of: 1) the cache's copy of a memory block, and 2) a state. The number of words in a block is the *blocksize*. The main memory is divided into *blocksize* modules interleaved on the low-order address bits.

A block is transferred to or from main memory by a sequence of transfers of one word per bus cycle. The words in a block are always transferred in the same order, as opposed to transferring the needed word first for a memory read operation. On a main memory write, the bus is released as soon as the word(s) are transferred to memory, although the memory module(s) will be busy for additional cycles to complete the write operation. Thus, memory contention can occur although there is only one bus.

An arbitrary number of the caches can simultaneously read the information on the bus.

7.2.2 Operation of the Cache

The cache is used for both instructions and data. When a processor makes a cache request, it holds until the request is satisfied. A cache request is either a read (CPU-READ) or write (CPU-WRITE) for a particular word that is in a block that might or might not have a valid copy in the local cache. If the processor's cache can service the request locally and immediately, then servicing takes one cycle.

A cache has two independent parts: a bus monitor and a controller. For every bus transaction, each cache's bus monitor determines if the referenced block has a valid copy in the local cache. In the case of such a block match, the cache's bus monitor signals the cache controller.

In a given cycle a cache controller can simultaneously receive a request from its processor and a signal from its bus monitor requiring action. To avoid a race condition, the controller can service only one of the requests at a time. The bus monitor request always has priority. Similarly, loading a block from the bus cannot be done simultaneously with the cpu read or write operation that initiated a load request.

There are four types of bus transactions: READ, READ-MOD, WRITE, and INVALIDATE. A READ (READ-MOD) transaction is started by a cache that has a miss on a read (write) from its processor. A WRITE transaction is due to a cache writing a word to main memory via the bus. An INVALIDATE transaction is started by a cache that wants to invalidate all the copies other caches have of a particular block. For all four types of transactions, the main memory address of the affected block is on the bus's address lines.

Each bus transaction type has an associated bus control line. There is one other bus control line, *shared*. It can be raised by more than one cache simultaneously. A cache raises the *shared* line to announce that it has a valid copy of the block involved in the transaction.

The READ and READ-MOD transactions implicitly involve main memory supplying a block. However, it is possible for any cache to inhibit main memory and to respond instead. Arbitration schemes determine which cache will respond if more than one can, and which cache gains control of the bus to initiate a new transaction.

7.2.3 States of the Cache Entries

Each cache entry has three bits of state information. The first bit indicates if the entry contains a valid or invalid copy of a main memory block. The second and third bits implement the two binary attributes that define the state of a valid copy of a block: *number of copies*, and *need to write back*. *Number of copies* can be ONLY or NOT-ONLY. ONLY means that the cache knows that it is the only cache with a valid copy. NOT-ONLY means that the cache does not know that ONLY holds. Other valid copies might or might not exist. *Need to write back* is WBACK or NO-WBACK. WBACK means that on replacement this cache has to write-back to main memory its copy of the block. The WBACK bit is a generalization of the traditional "dirty" bit, which indicates that the contents of the block have been written since the last time the block was written to main memory. In some of the more sophisticated protocols, multiple valid copies may exist of a "dirty" block, yet only one of these copies is in state WBACK.

7.3 The Protocols

We define a Basic protocol which does not use the *invalidate* or *shared* control lines. We then describe four enhancements to the Basic protocol. In Section 7.3.3, we describe how the various protocols proposed in the literature are related to the Basic protocol and the four enhancements.

The protocols are specified by defining the actions taken by a cache in response to requests from the bus or local processor. Actions are defined for the type of request, and the state of the relevant block copy in the local cache memory. Recall that bus transactions have priority over processor requests. Thus a processor request is blocked when there is a bus transaction that matches a valid local block copy. Lost cycles due to cache, bus, or memory contention are not mentioned in the protocol definition.

7.3.1 The Basic Protocol

The cache takes the actions 1 and 2 in response to bus transactions:

1. READ or READ-MOD

   (a) No valid copy: Do nothing. If no cache has a valid copy, then main memory supplies the block in 3 + *blocksize* cycles.

   (b) (ONLY or NOT-ONLY, NO-WBACK): One of the caches is chosen to supply the block and main memory is inhibited. (Note that no other cache can have a copy in state (ONLY, WBACK)). On the cycle after the bus request, the supplier puts the block on the bus for *blocksize*

cycles. The block copies in all these caches go to state (NOT-ONLY, NO-WBACK) or are invalidated for a READ and READ-MOD, respectively.

(c) (ONLY, WBACK): At most one cache can be in this block substate. It inhibits main memory. On the cycle after the bus request, the supplier puts the block on the bus for *blocksize* cycles. It then writes the block to main memory for *blocksize* cycles. The block copy goes to state (NOT-ONLY, NO-WBACK) or is invalidated for a READ and READ-MOD, respectively. Note that (NOT-ONLY, WBACK) is impossible in the Basic protocol.

2. WRITE

   (a) Any valid copy: Block copy is invalidated.

Actions 3 and 4 are taken in response to processor requests:

3. CPU-READ

   (a) Any valid copy ("Read Hit"): Spend 1 cycle locally supplying the processor.

   (b) No valid copy ("Read Miss"): In the current cycle put a READ on the bus with the desired block address. For the *blocksize* (+3) next cycles the supplier (another cache or main memory) supplies the block. When the block has been supplied, then if the supplier was a cache block in state (ONLY, WBACK), then the supplier will use the bus for the next *blocksize* cycles for write-back to main memory. When the supplier is finished with the bus, if a requester's cache entry in state (ONLY, WBACK) had to be replaced, then write it to main memory in *blocksize* cycles. When the requester is finished with the bus, then in the next cycle it supplies the desired word to its processor. Set block copy state to (NOT-ONLY, NO-WBACK).

4. CPU-WRITE

   (a) No valid copy: Same as 1.a. except for three changes: 1) at the start READ-MOD is raised, 2) the last cycle of servicing the request is an update in the cache instead of a processor read, and 3) the new block state is (ONLY, WBACK).

   (b) (ONLY, NO-WBACK or WBACK) Spend 1 cycle locally updating the word. Set block copy to state (ONLY, WBACK).

   (c) (NOT-ONLY, NO-WBACK) Spend 1 cycle updating the word locally. Then spend 1 cycle using the bus to write the word to main memory with the WRITE line raised. This will be referred to as the *broadcast write*. Set block copy state to (ONLY, NO-WBACK).

7.3.2 Enhancements

We consider four independent enhancements. The first three reduce the times blocks or words are written to main memory towards the minimum of only on replacement. The fourth removes the restriction that when there is one writer of a block that there are no other readers. Each of these enhancements is presented below as a change to the Basic protocol. Their qualitative effects on performance are also assessed. In section 7.3.2 we consider the interactions of the enhancements.

Reduced Memory Writes

The three enhancements that reduce memory writes are:

1. A cache raises the *shared* line when supplying a block to another cache. If the *shared* line is not raised (i.e. memory is the supplier), the receiving cache marks the block as ONLY. In this case, the broadcast write (action 2c above) will be needed in fewer cases.

2. The cache supplier on a READ or READ-MOD request does not write its copy to main memory even if its copy is in state WBACK.

3. Instead of a broadcast write, the cache raises the *invalidate* line for one cycle.

Enhancement two implies the some cache's copy must stay in state WBACK so that the updates are not lost. On a READ-MOD the supplier is going to invalidate its copy so the requester must be responsible. On a READ the supplier must be responsible since the requester might be a read-only cache.

Enhancements one and two clearly improve performance since they decrease bus accesses and/or memory writes. The effect of enhancement three is less clear. Raising the *invalidate* line has a lower cost than a main memory write. On the other hand, if the only write to that block is that one write to that one word, then using the *invalidate* line implies that block write-back is needed on replacement. Which approach is better depends on the probability that there is only one write to the block before replacement.

Readers and Writers

The fourth enhancement allows multiple valid cache copies even if a data block is modified. We refer to this as *multiple readers/writers*. The Basic protocol invalidates other copies on the first write. The first write occurs on a READ-MOD, or on a CPU-WRITE to an in-cache copy that has only been read previously. Thus the changes to the Basic protocol are:

1. On a READ-MOD, the requester broadcasts the updated word, and all valid copies go to (NOT-ONLY, NO-WBACK).

2. On all CPU-WRITEs to NOT-ONLY blocks, the cache broadcasts the updated word and writes it to main memory. All other caches with valid copies update them. The state of the block in all caches remains (NOT-ONLY, NO-WBACK).

Enhancement 4, alone, changes the Basic protocol to a pure "write-through" protocol. Unless enhancement 4 is combined with enhancement 1, which uses the *shared* line to notify the writing cache that it has the only copy of a block, the performance of enhancement 4 is in most cases lower than the Basic protocol. From now on we only consider enhancement 4 as combined with enhancement 1. The requester on a READ-MOD only broadcasts the updated word if the *shared* line is raised (by the supplier). Each cache then continues to broadcast writes if the *shared* line is raised on the previous broadcast write. Otherwise, the writer goes to block state (ONLY, WBACK).

The advantage of allowing multiple valid copies is that when a cache wants to read or write its copy that copy is always valid. The disadvantage is that all writes to NOT-ONLY blocks have to be broadcast. This slows the writing cache because the bus has to be obtained and main memory has to be free. The caches with the other copies are also slowed because they may have to block local CPU requests to update their copies.

Whether the advantage or disadvantage dominates depends on the access pattern. A good access pattern for multiple readers/writers is when the majority of the caches with valid copies have high frequencies of accessing their copies and those accesses have a fine temporal granularity of interleaving. A bad access pattern is when one cache frequently writes its copy and the other caches very infrequently read or write their copies.

An extension to the fourth enhancement is possible. The idea is to dynamically switch between the multiple readers/writers approach and the invalidation approach depending on the quality of the access pattern. The key is to find a simple, but accurate dynamic measure of access pattern quality. The measure proposed in the literature [RUD84] is whether or not two subsequent writes to the given block are by the same cache. If so, the interleaving of writes is considered not fine enough to merit the multiple readers/one writer approach.

Combinations of Enhancements

The above enhancements are presented, except as noted, as independent changes to the Basic protocol. When enhancements are combined, the changes to the Basic protocol are somewhat different. The most noteworthy difference occurs when the third and fourth enhancements are combined. Instead of the *invalidate* line being raised, a write is broadcast but not written to main memory. Consequently, some cache has to take responsibility for write-back on replacement. We assume the broadcaster takes responsibility by going to state WBACK. The other caches remain in state NO-WBACK.

7.3.3 Protocols in the Literature

Our goal is to develop a general framework within which the key traits of the protocols that have appeared in the literature can be identified and evaluated. Given that framework, we are able to quantitatively assess the performance of the Basic Protocol and of the various enhancements to it. We do not claim that the protocols in the literature exactly conform to the framework above. We do, however, feel that the correspondence is close enough to be of interest. Our performance models are described in the next section. Here we relate the model to the specific protocols in the literature.

The Basic protocol is essentially the Write-Once protocol [GOO83], except that another cache will supply the block on a READ or READ-MOD request even if its copy is in state NO-WBACK. The Synapse protocol [FRA84] modifies the Basic protocol by including enhancement 3. The "ownership-based protocol" [KAT85] builds on the Synapse protocol by adding enhancement 2. Papamarcos and Patel's protocol [PAP84] uses enhancements 1 and 3. In addition, their protocol assumes main memory is updated on the same bus transaction as the cache supply, which should give similar performance to enhancement 2. The Dragon proposal [MCC84] combines all four enhancements. Rudolph and Zegall's RWB protocol [RUD84] uses enhancements 1, 3, and 4, including the extension to dynamically switch between the invalidation approach and the multiple readers/one writer approach.

7.4 The Protocol Models

We have created GTPN models to estimate the performance of the Basic protocol and five protocols which include combinations of enhancements described in the section 7.3.2. Enhancement 1 can be implemented easily if the bus contains the additional control line we have called *shared* (e.g. as provided by Futurebus). Thus, we have named the protocol which includes enhancement 1 only, the Smart Basic protocol. The four additional protocols we have studied are called +(2), +(3), +(4), and +(2,3,4), where the numbers refer to the enhancements included in addition to Smart Basic. In this section we briefly describe the GTPN models and the workload parameters. The performance estimates obtained by solving the models are presented in Section 7.5.

7.4.1 The Basic Protocol: Net

The net for the Basic protocol is shown in Figure 7.2. Each token in place P1 represents a processor that has just completed an instruction cycle. We assume that all processors are stochastically homogeneous (i.e. their memory access behaviors are statistically identical). The tokens in places P2 and P9 indicate that the bus and memory are available, respectively.

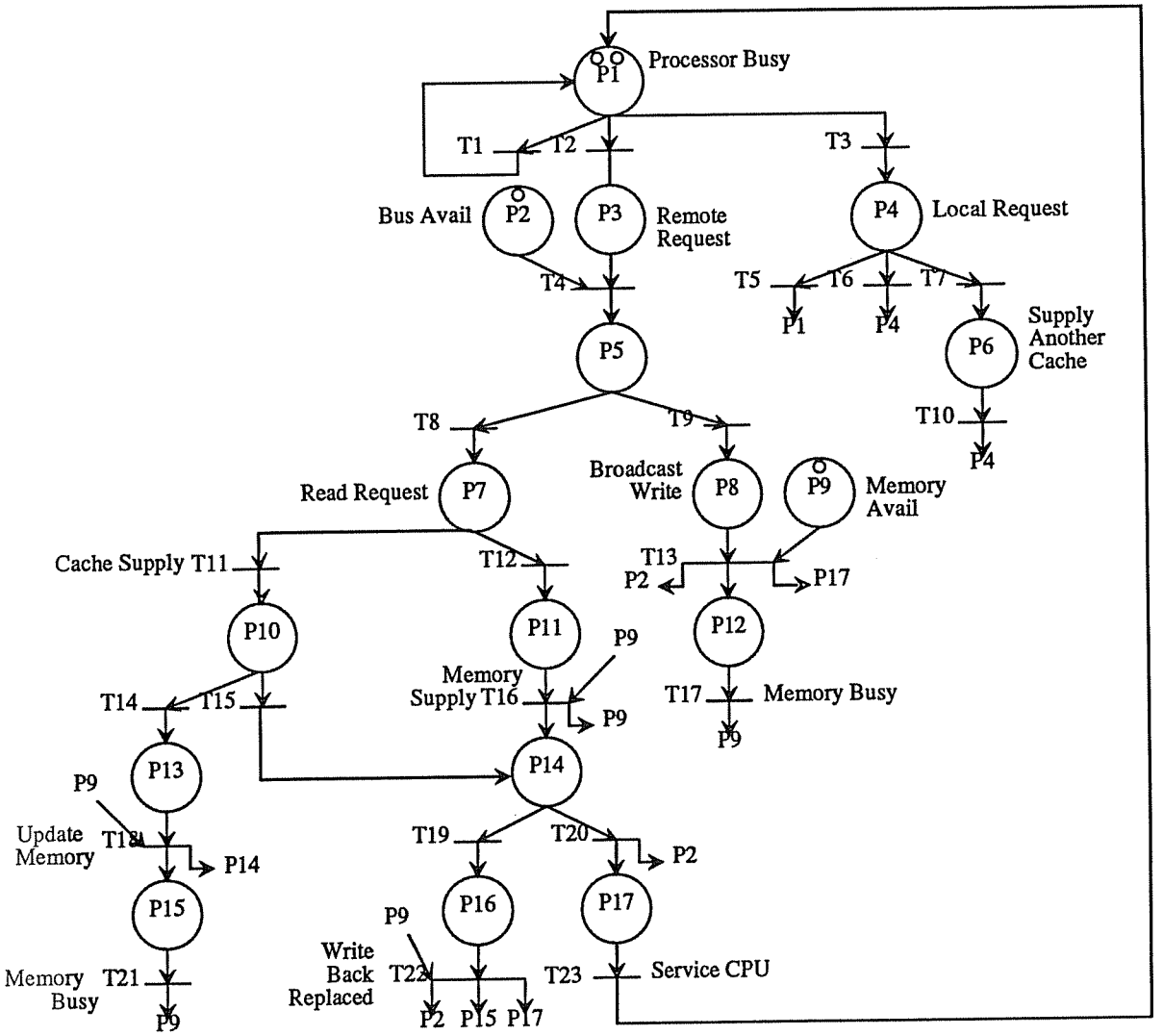

Figure 7.2: GTPN Model for the Basic Protocol

Each processor token acts independently. During each cycle, the processor continues instruction execution (T1), makes a cache request that can be serviced locally (T3), or makes a cache request that requires the bus (T2). Requests that are serviced locally may be blocked if the cache is servicing a bus request from or supplying data to another cache (T6 or T7). CPU requests that require the bus are either broadcast writes (T9, T13, and T17) or read (i.e. READ or READ-MOD) requests (T8). Read requests are either supplied by another cache (T11) or by main memory (T16). If supplied by another cache, the cache supplier will write the block back to main memory if it was in state WBACK (T18 and T21). After the cache supplier is through, the requesting cache may need to write back a replaced block (T22 and T21). Finally, the CPU request is serviced by the local cache (T5 and T23).

The attributes associated with the transitions in the Basic Protocol model are given in table 7.1 and table 7.2. The reader is encouraged to check that the durations associated with each transition make sense for the model description above. For example, transition T8, which represents a LOAD request on the bus, has a duration of 1 cycle. Transition T10 has a duration of 0 because it is inhibited by certain net markings which will be discussed below.

The frequency expressions may be marking-dependent. That is, they may contain the names of places and/or transitions, which evaluate to the marking of the place or transition in a given state. For example, the frequency of transition T6 evaluates to zero, unless T8 or T13 is firing (i.e. some other cache has a signal on the bus). Similarly, transition T10 is inhibited if T14 or T18 is firing or if P13 contains a token (i.e. a cache request is being supplied by another cache). This should also make sense from the model description above. The probabilities named in the frequency expressions are derived from basic workload parameters, such as the probability that a CPU request is a read (R) or write (W) for a private (P) or shared (S) block. The parameters and the derivation of the frequency expressions are discussed in the next section on the model workload.

The resources associated with the net transitions identify performance measures to be calculated. The long run expected number of usages of each resource is calculated automatically during analysis of the model. We are interested in three measures: bus utilization (Bus), processing power (PP), and speedup (Spd). Each of these resources is "in use" when any of the associated transitions are firing. Bus utilization is straightforward. It is the long run expected fraction of time that T8, T11, T13, T16, T18, or T22 is firing. Both processing power and our speedup measure [HOL85b] can be defined as the average fraction of time a processor is productive times the number of processors (for the homogeneous case). The difference is that processing power only counts the cycles the processor spends executing (T1) as productive, whereas our speedup includes the one cycle required for each cache request (T5 and T23). Processing power is most often used in the performance modeling literature, and is included in our model to show how it is specified. The speedup measure compares the effective computing power of the multiprocessor with a uniprocessor that has an infinite cache. We feel that speedup is a better

Transition	Frequency
T1	1-ProcReq
T2	(PSRWM + PSWHumod) × ProcReq
T3	(PSRH + PSWHamod) × ProcReq
T4	1.0
T5	1 - Freq(T6) - Freq(T7)
T6	0.5 × (SRMiss + SWMiss) × T8
	+0.5 × (ShRead + ShWrite) × T13
T7	1/#processors × (T14 | P13 | T18)
T8	PSRWM / (PSRWM + PSWHumod)
T9	1-Freq(T8)
T10	1-(T14 | P13 | T18)
T11	CSupSR × SRMiss + CSupSW × SWMiss
T12	1 - Freq(T11)
T13	1.0
T14	WBCSupSW × SWCSup
T15	1-Freq(T14)
T16	1.0
T17	1.0
T18	1.0
T19	RepP × Priv + RepSW × ShWrite
T20	1-Freq(T19)
T21	1.0
T22	1.0
T23	1.0

Table 7.1: The Frequency Attribute of Basic Protocol Model Transitions

Transition	Duration	Cnt Combs	Resources
T1	1.0	yes	(Spd,PP)
T2	0.0	yes	()
T3	0.0	yes	()
T4	0.0	no	()
T5	1.0	yes	(Spd)
T6	1.0	yes	()
T7	0.0	yes	()
T8	1.0	no	(bus)
T9	0.0	no	()
T10	0.0	no	()
T11	blocksize	no	(bus)
T12	0.0	no	()
T13	1.0	no	(bus)
T14	0.0	no	()
T15	0.0	no	()
T16	3 + blocksize	no	(bus)
T17	1.0	no	()
T18	blocksize	no	(bus)
T19	0.0	no	()
T20	0.0	no	()
T21	max(4 - blocksize, 0)	no	()
T22	blocksize	no	(bus)
T23	1.0	no	(Spd)

Table 7.2: The Other Attributes of Basic Protocol Model Transitions

measure of the perceived performance of memory intensive computations on the multiprocessor.

Two approximations are made in the Basic protocol model. First, the amount of time memory is held for a block write (T18 and T21, or T22 and T21) is sufficient for the first module to finish the write operation. The other blocksize-1 modules will still be busy for up to blocksize-1 cycles into the future. The model is accurate if the next bus access is a load request, but will be slightly inaccurate if a (one-word) broadcast write occurs to one of the modules that is still busy. Second, the amount of time memory is held for a broadcast write (T13 and T17) is approximated to be 2 cycles. The module written to is busy for 4 cycles, but this module may not be the module addressed on the next memory access. Both approximations are necessary because we are representing blocksize memory modules with a single token in place P9. We have verified that the model results are not sensitive to these approximations.

7.4.2 The Basic Protocol Workload

The performance of the Basic protocol, and the relative improvement in performance for each of the protocol enhancements, is highly dependent on the amount of data sharing that occurs dynamically in the workload. For example, if there is no data sharing, then enhancement 1 might yield some performance improvement over the Basic protocol, but enhancements 2 and 4 will have no effect. Thus, the workload model and parameter values are important considerations.

In the absence of experimental data for multiprocessor workloads, we have based our model on the workload model proposed by Dubois and Briggs [DUB82]. They define the workload for an MIMD machine to be the merge of two memory access streams: one stream for private and shared read-only blocks, and one stream for shared writeable blocks. We have separated the first stream into two streams, one for private, and one for shared read-only blocks. This allows us to define more fundamental workload parameters for our more detailed protocol model. Thus, we view the stream of memory requests in our model to be the merge of three streams, for: 1) private blocks, 2) shared read-only blocks, and 3) shared writeable blocks.

The fundamental parameters for this workload model are shown in table 7.3 and table 7.4. "ProcReq" is the probability that a processor makes a memory request in a given cycle (see transitions T1-T3 in table 7.1 and table 7.2). We assign interrequest times to be geometrically distributed with mean 2.5. This is based on the assumption that a large fraction of interrequest times will be in the range of 0-2 cycles, but several instructions which occur occasionally, such as multiply, can be much longer.

We consider three levels of data sharing: 1%, 5%, and 20%. The probabilities that a memory request is for private (P), shared read-only (SR), and shared write-able (SW) data are shown for each of these cases in the table. The 1% sharing case has probabilities such as might occur if only the operating system shares data. The

Parameter	Meaning
ProcReq	Processor request
Priv,ShRead, ShWrite	processor request to a P, SR, SW block
HitP, HitSR, HitSW	hit given P, SR, SW
ReadP, ReadSW	read given P, SW
AmodPWH, AmodSWH	block copy is already modified
	given PWH, SWH
CSupSR, CSupW	cache supplier given SR, SW
WBCSupSW	cache supplier write back given SW
RepP, RepSW	write back replaced block given P, SW
SmartRepP	RepP given SmartBasic
+2RepSW, +23RepSW	RepSW given +(2), +(2,3)
MrwHitSW	HitSW given multiple readers/writer

Table 7.3: Meaning of Fundamental Workload Parameters

Parameter	Value
ProcReq	0.286
Priv,ShRead, ShWrite	0.99, 0.00, 0.01
	0.95, 0.03, 0.02
	0.80, 0.15, 0.05
HitP, HitSR, HitSW	0.95, 0.95, 0.5
ReadP, ReadSW	0.7, 0.5
AmodPWH, AmodSWH	0.7, 0.3
CSupSR, CSupW	0.95, 0.5
WBCSupSW	0.3
RepP, RepSW	0.2, 0.5
SmartRepP	0.3
+2RepSW, +23RepSW	0.6, 0.7
MrwHitSW	0.95

Table 7.4: Values of Fundamental Workload Parameters

20% case has probabilities indicating a tightly-coupled parallel computation that frequently accesses shared read-only data, and thus has good potential for speedup on a multiprocessor. The 5% sharing is an intermediate case.

The parameters for private blocks were chosen according to data reported from extensive uniprocessor cache simulations by Smith [SMI85a]. This includes the hit ratio (HitP), read ratio (ReadP), and Smart Basic probability of write back on replacement (SmartRepP). The RepP parameter for the Basic protocol is somewhat lower because private data is written through to memory on the first write. The hit ratio for SR blocks is assumed to be the same as for P blocks, as in the Dubois and Briggs model. Hit ratio and read ratio for SW data were chosen to be pessimistic estimates which should show maximum performances differences among the different protocols.

The probabilities that another cache will supply blocks on a miss for SR or SW data, are set equal to the hit ratios. This assumes a high temporal degree of sharing. The remaining workload parameters for SW data were also chosen to be conservative.

The derivation of probabilities used in the frequency expressions of Table 7.1, from the fundamental workload parameters, is given in Table 7.5. For example, a memory request can be served by the local cache (transition T3) if it is a read hit for a private or shared block (PSRH), or if it is a write hit for an already modified block (PSWHamod), as derived in Table 7.5. The interested reader should be able to follow the remaining derivations. Note, in general, "H" stands for hit, "M" stands for miss, and "umod" stands for unmodified in the tables. "SRMiss" should be read as "Shared Read given Miss".

Probability	Derivation
PWH	Priv × (1-ReadP) × HitP
SWH	ShWrite × (1 - ReadSW) × HitSW
PSRH	Priv × ReadP × HitP + ShRead × HitSR
	+ ShWrite × ReadSW × HitSW
PSRWM	Priv × (1- HitP) + ShRead × (1- HitSR)
	+ ShWrite × (1 - HitSW)
PSWHamod	AmodPWH × PWH + AmodSWH × SWH
PSWHumod	(1 - AmodPWH) × PWH +
	(1 - AmodSWH) × SWH
Denom	ShRead × (1 - HitSR) + ShWrite × (1-HitSW)
	+ Priv × (1-HitP)
SRMiss	(ShRead × (1 - HitSR))/Denom
SWMiss	(ShWrite × (1 - HitSW))/Denom
SWCSup	(ShWrite × (1-HitSW))/(ShWrite ×
	(1-HitSW) + ShRead × (1-HitSR))

Table 7.5: Intermediate Workload Values

7.4.3 The Other Protocols

The Smart Basic protocol requires no change in the net, but requires three changes in the transition frequencies. A write-hit to an unmodified private block no longer needs the bus, since the cache will always know it has the only copy of a private block. Thus, the PWHumod term moves from transition T2 to transition T3. This will increase the probability that a private block has to be written back on replacement (RepP becomes SmartRepP for T19). Furthermore, the second term in the frequency expression for T6 changes to 0.5 × T13, since all broadcast writes are to shared blocks. These changes are also made for the remaining protocols. Note that a write-hit to an unmodified shared block (SWHumod) will also not require the bus if the cache has the only copy of the shared block. The probability of this is difficult to estimate, so we choose the conservative approach of assuming the probability is negligible.

Protocol +(2) requires one change in the net, and a further change in the transition frequencies. The column headed by transition T14 is removed and RepSW increases to +RepSW (T19), since the cache supplier does not write back.

The +(3) protocol also requires a change in the net. The place P9 is removed as an input and the column headed by P12 is removed as the output of transition T13, since the invalidation signal does not require a main memory write. Also, +RepSW is used instead of RepSW, since blocks are only written back on replacement. Note that we estimate that the effect of enhancement 3 on RepSW is comparable to the effect of enhancement 2.

Enhancement 4 requires a change in one firing duration, and several changes to the transition frequencies. A requester must broadcast the updated word if a cache supplies the block for a READ-MOD request. Thus, the firing duration of T15 becomes 1.0. Since all write hits to shared blocks are broadcast, SWHamod moves from T3 to T2, and SWH replaces SWHumod in transition T8. Second, HitSW is increased to MrwHitSW, because the shared blocks remain valid.

The +(2,3,4) protocol requires all of the above changes, and uses the ++RepSW parameter instead of RepSW because both 2 and 3 are incorporated.

7.5 Protocol Performance

Using our GTPN model, we obtained a processing power estimate of 4.1 for the +(2,3) protocol, with 5% sharing and nine processors. This value agrees well with Papamarcos and Patel's approximate analysis [PAP84] for a block transfer time of 4 and similar workload. At 4 processors the bus is only 50% utilized for the +(2,3) protocol, and we find approximately a 10% increase in bus utilization for the Basic protocol. This agrees with the trace simulation results in [KAT85] for 8 kilobyte cache size. These results indicate that the model produces reasonable performance estimates.

The goal of our experiments is to compare the performance of the six protocols (Basic, Smart Basic, +(2), +(3), +(4), and +(2,3,4)), given reasonable values for architectural and workload parameters. Two architectural parameters considered are the *blocksize* and the main memory cycle time. Both are four in our initial experiments. The performance measure primarily used is *speedup*, as defined in Section 7.4.1.

7.5.1 Effect of Enhancements

Figure 7.3 shows the results of our initial experiments. The curves start below one on the y-axis because we have analyzed the uniprocessor with cache misses (but with frequency 0 for T6, T7, and T11). The bottom dashed, dotted, and solid curves are for the Basic protocol. The next set of curves are the speedup for the Smart Basic protocol. The curve for Smart Basic with 20% sharing is almost hidden by the curve for Basic with 1% sharing. The top three curves are the speedup for the +(4) protocol.

The curves for the +(2) and +(3) protocols are nearly identical to the Smart Basic protocol. The speedup for +(2) is at most 2.5% greater than Smart Basic. The speedup for +(3) is at most 0.1% less than Smart Basic. For the sake of readability, these curves are not shown in the figure. Similarly, the +(2,3,4) protocol curves are not drawn because they are indiscernable from the +(4) protocol. Table 7.6 summarizes the speed-up estimates for all of the protocols for the 10-processor case.

Protocol	Percent Sharing		
	1%	5%	20%
Basic	5.602	5.371	4.868
SmartBasic	6.718	6.310	5.582
+(2)	6.746	6.370	5.715
+(3)	6.716	6.306	5.576
+(4)	6.913	6.983	6.929
+(2,3,4)	6.911	6.993	6.972

Table 7.6: Summary of Speed-Up Estimates for 10 Processors

Clearly, adding enhancement one substantially improves system performance. Enhancements two and three have negligible effect when added to Smart Basic. In fact, for the workload parameters used, the effect of enhancement 3 is very slightly negative. Enhancement 4 shows a small improvement over Smart Basic for our workload with 1% sharing, but shows more substantial improvement as sharing is increased to 5% and 20%.

The spread between the solid, dotted, and dashed lines indicates that the amount of sharing has a significant effect on the performance of the Basic, Smart Basic, +(2), and +(3) protocols. This is reasonable because the number of broadcast writes and the average hit rate are proportional and inversely proportional, respectively, to the

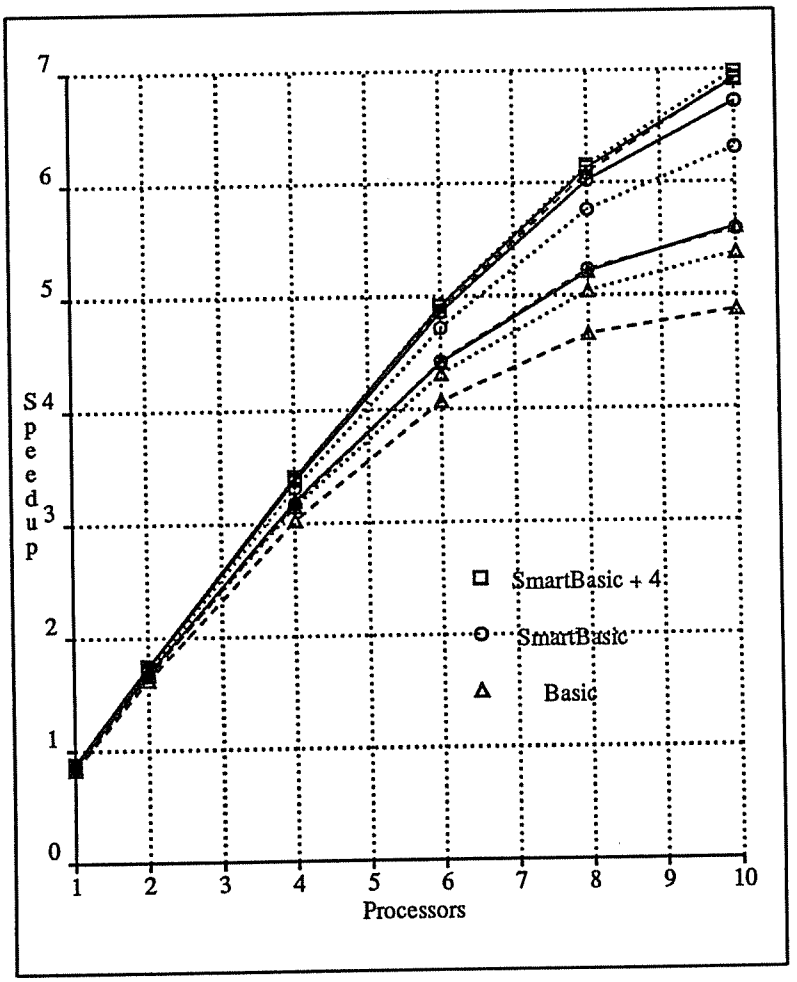

Figure 7.3: Comparison of Protocols for 1% (solid), 5% (dotted), and 20% (dashed) Sharing

amount of sharing in these protocols. In contrast, the amount of sharing has little influence when enhancement 4 is included. This is because the hit rate for shared data was assumed to be high for enhancement 4.

The effectiveness of enhancement 4 in Figure 7.3 requires further study because our model assumed a good access pattern for multiple readers and writers (see section 7.3.2). To model a bad access pattern we reran +(4) at 20% sharing without increasing the hit rate. The results were very slightly lower than the Smart Basic results, because there are more broadcast writes. Thus, enhancement 4 helps significantly (especially at large amounts of sharing) when there is a good access pattern and has negligible effect when there is a bad access pattern. It appears that dynamically switching between this and the invalidation approach (enhancement 3) offers no advantages.

The curves in Figure 7.3 terminate at ten processors because of rapid growth in the size of the state space. Table 7.7 shows this growth for the Basic Protocol. Nonetheless, none of the curves will rise much further because of bus saturation. Bus utilizations vary from 89% to 98% at ten processors for the workloads and protocols in Figure 7.3. In section 7.5.3 we show that the assumed hit rates in our initial workload are primarily responsible for the bus saturation at 9-10 processors.

Processors	States
1	33
2	355
4	2,364
6	7,961
8	19,856
10	41,159

Table 7.7: State Space Growth in the Basic Protocol

7.5.2 Effect of Blocksize

Figure 7.4 shows our experiments which varied the blocksize. The experiments in Figure 7.3 assumed a blocksize of four words. Blocksizes of one and eight are considered here, for the Smart Basic protocol. The one word *blocksize* case requires a change in the protocol. In this case, there is no need to load a block on a write miss, since whatever value is loaded is completely overwritten. However, the other caches still must be notified. Thus, a write miss can be implemented as a broadcast write. The net does not have to be changed. The write miss probability (PSWM), however, is moved from transition T8 to transition T9.

The key effect of changing blocksize on the workload is on the hit rate. Based on Alan Smith's results for an 8 kilobyte cache [SMI85b], we reduce HitP and HitSR from 95% to 90% for a blocksize of one, and increase these values to 97% for a blocksize of eight. HitSW is reduced to 40% for blocksize one, and increased to 55%

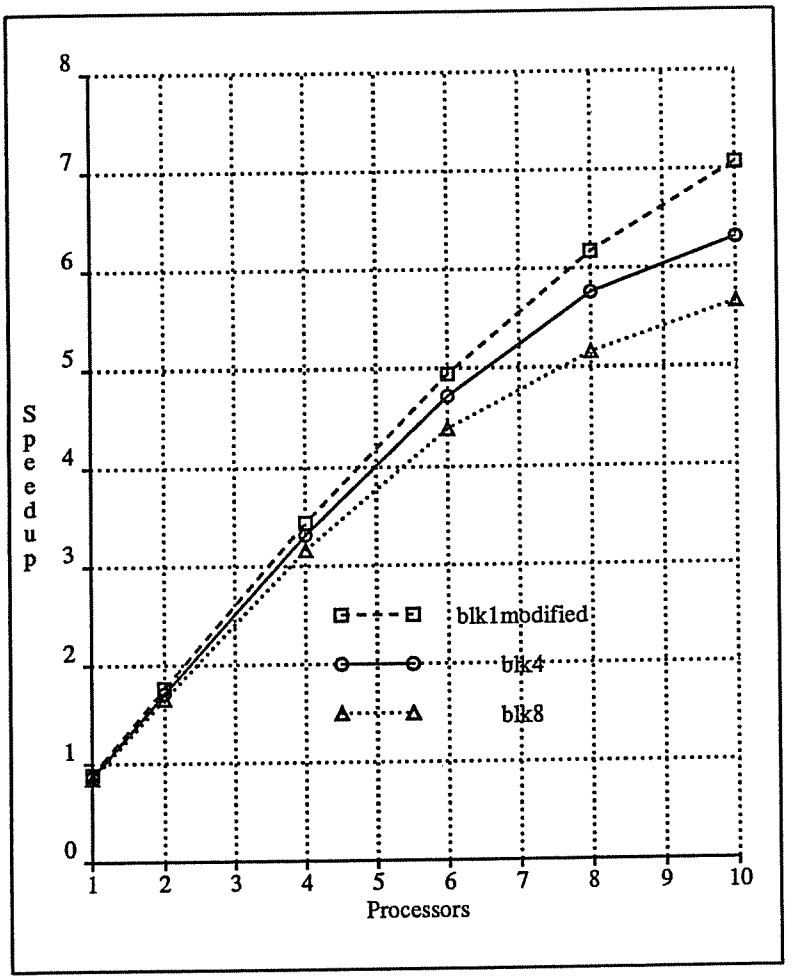

Figure 7.4: Varying Block Size. SmartBasic at 5% sharing.

for blocksize eight, compared with 50% for blocksize four. Other parameters, such as AmodPWH, AmodSWH, WBCSupSW, RepP, and RepSW, were also suitably modified.

Blocksize of one, with the protocol appropriately modified, performs best in these experiments. However, smaller blocksizes have substantial disadvantages due to a larger number of memory accesses during task switching, which we have not modeled. A study of blocksize effects which includes task switching in the workload would be worthwhile.

7.5.3 Effect of Hit Rate and Memory Speed

Figure 7.3 indicates that at most ten or so processors should be put on a single bus. These results are likely to be very conservative because we have chosen many of our parameter values very conservatively. Two parameter changes seem especially worthwhile to consider. The first change increases the hit rate for private and SR blocks from 95% to 99%. The second change assumes a faster main memory. In particular, we investigate a shared memory cycle time of two instead of four. Note that main memory accesses still require a minimum of 4-5 cycles, due to acquiring the bus and memory, transmitting the bus request, and servicing the CPU when the request is satisfied. The resulting four curves are shown in Figure 7.5 for the Smart Basic protocol at 20% sharing.

Though both changes are significant, the hit rate increase is clearly more important. This is also demonstrated when bus utilization is considered. At nine processors the bus utilization for the original model is 96%. The utilization for the model with main memory cycle time of two and 95% hit rate is 90%. The utilization for the model with main memory cycle time of two and 99% hit rate is 53%. Thus, the faster memory probably only allows one or two more processors. The higher hit rate should allow several more processors, perhaps six or eight, to be added. Recall that this is for the 20% sharing level. Further increases in the number of processors that can be supported can be expected for the reduced sharing workloads.

7.6 Conclusion

We have used an exact analytic technique, based on Generalized Timed Petri Nets, to derive performance estimates for shared bus cache consistency protocols. Using the GTPN model, we were able to specify both constant delays and instantaneous events for detailed bus and memory activity, which must be represented to evaluate these protocols. The GTPN can be solved for steady-state performance estimates with these parameters, in contrast to previous Stochastic Petri Net models. Performance estimates were obtained automatically using Markov Chain techniques.

The workload in these studies was based on the model used by Dubois and Briggs [DUB82], but included separate specification of private and shared read-only

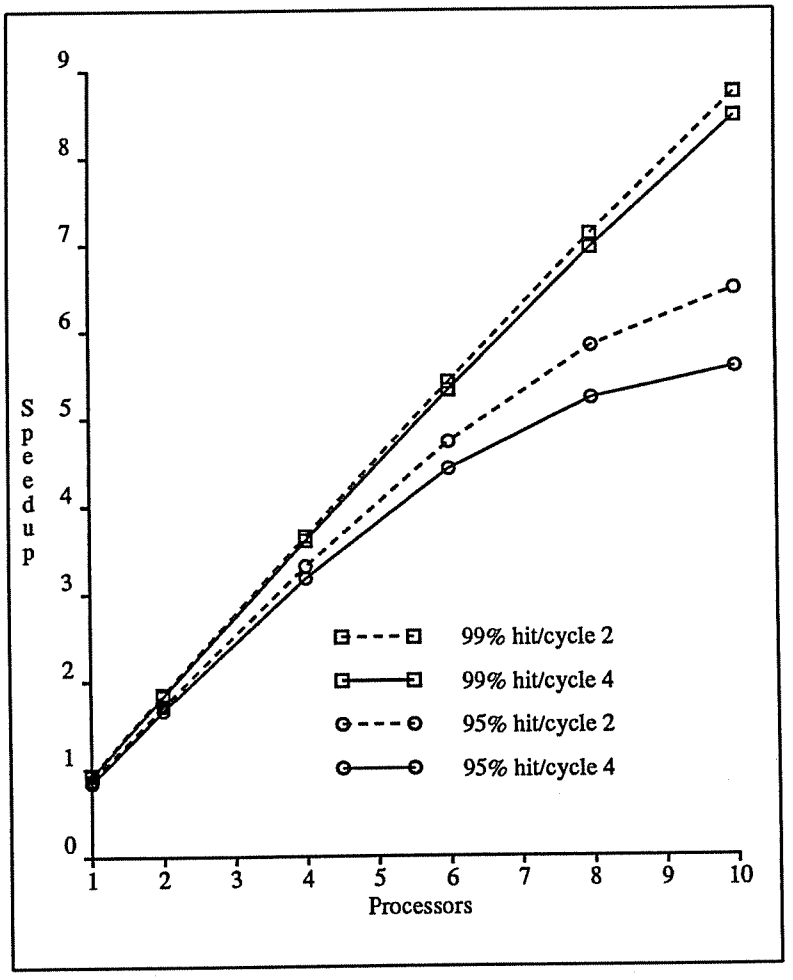

Figure 7.5: Effect of Hit Ratio and Main Memory Cycle Time. SmartBasic at 20% sharing.

data access behavior. We considered three levels of data sharing in our workloads: 1%, 5%, and 20%. Parameter values for private data were based on uniprocessor cache simulations by Smith [SMI85a]. We found that the level of sharing, and the assumed cache hit rate for private and shared read-only data, have a substantial impact on the performance estimates.

We evaluated four enhancements to the Write-Once protocol which have appeared in the literature. One enhancement makes use of the *shared* bus line which is raised by a cache when supplying another cache, as proposed in [PAP84,MCC84,RUD84]. This enhancement improves system performance by as much as 15-20% at all levels of data sharing studied. Another enhancement (enhancement 4 in our models) allows multiple readers and writers for a data block, as proposed in [MCC84,RUD84]. This enhancement further improves performance by as much as 10–20% for the 5% and 20% sharing workloads, as long as there is a fine temporal interleaving of accesses to the data block by more than one cache. Also, this enhancement does not decrease performance appreciably in other workloads we studied. The remaining two enhancements which have been proposed in the literature do not show significant performance improvement for any of the workloads studied.

Chapter 8

Conclusion

8.1 Summary

We have developed a modeling technique and used that technique in several modeling studies. The modeling technique, Generalized Timed Petri Nets, significantly advances the previous techniques that support the representation of deterministic time. We described in chapter 1 the previous techniques that, like ours, are based on Petri Nets. The work by Zuberek and Razouk and Phelps includes deterministic time but imposes restrictions to simplify constructing the state space and analyzing the state space. Our approach removes all of these restrictions except that the state space be finite.

In chapter 2 the GTPN model was presented and the algorithms used in constructing the state space were discussed. The key algorithms involve finding next states efficiently and computing the probabilities of next states. In chapter 3 the methods for analyzing the state space were discussed. The GTPN is viewed as a stochastic process with an embedded discrete-parameter Markov Chain. Methods are suggested for handling the general case where there are possibly multiple recurrent classes, transient classes and some recurrent classes are periodic. Some of the numerical issues involved are also discussed. Handling the general case is especially significant, because in non-pathological nets we have generated state spaces with transient classes, multiple recurrent classes, and periodic recurrent classes. Certain of the Dining Philospher nets had transient classes and multiple recurrent classes. A periodic recurrent class was generated from a GTPN net modeling the handling of inter-process communication by a front-end processor [RAM86].

Four modeling studies were conducted using the GTPN. Chapter 5 discussed the studies of the Dining Philosophers Problem and of the scalar mode of the CRAY-1. Chapter 6 presented the study of multiprocessor memory and bus interference. Using the GTPN exact performance estimates were derived for many cases for which

exact estimates were previously considered to be computationally infeasible. Chapter 7 treats the study of shared bus cache consistency protocols. This study isolates the key features of these protocols and determines the effect on performance of each feature. No such comparison previously existed.

In conclusion, we feel that we have developed a promising modeling technique and reached some interesting conclusions concerning the performance of parallel architectures. In the next section we consider possible future directions.

8.2 Future Research Directions

There are a number of possible future research directions.

1. More General Resource Usage Estimates

   Some straightforward changes to the GTPN are needed to support modeling certain situations. In particular, resource usages should not just occur due to transition firings. A token in a place should be allowed to count as a resource usage. More generally, resource usages should be arbitrary expressions containing the names of transitions and places as well as arithmetic, logical, and relational operators.

2. Investigating Numerical Methods

   Alternative numerical methods could be investigated for improving the performance of the state space analysis. With respect to computing the stationary probability distribution, currently the Power Method is used. To ensure a unique dominant eigenvalue shifting and scaling of the transition probability matrix is done. The current scaling factor is that suggested in [WAL66]. Investigating the effect of other scaling factors on the convergence rate appears worthwhile. The current shifting and scaling ensures that the unique dominant eigenvalue is the real number 1. A simple shift by an arbitrary positive real number also ensures a unique dominant eigenvalue though the value of that eigenvalue is no longer 1. Implementing the simple shift and comparing the two implementations is worth doing. As discussed in chapter 3, in the case of reliability models based on continuous-time markov chains, a study by Stewart and Goyal [STE85] has shown that the Successive Overrelaxation (SOR) method is superior to the Power Method in many cases for finding the stationary probability distribution. Comparing these approaches in the setting of the GTPN is clearly a direction for future research.

3. Parallel Solution of the GTPN

   The GTPN tool constructs a state space sequentially and then solves it sequentially. Since handling the largest possible state space in a reasonable amount of time is important, parallelizing the tool appears very useful. The state space

analysis is almost totally numerical and so the algorithms for parallelizing it fall in the domain of parallel numerical analysis.

Constructing the state space, on the other hand, is an issue more peculiar to the GTPN. Expanding different parts of the state space concurrently seems straightforward except for the issue of checking for duplicates. A shared memory paradigm seems reasonable. The array of already constructed states is accessible to each *expander* of the state space. The addition of a new state to the state space array by an expander must be implemented as an atomic event to ensure mutual exclusion. This is needed because adding a state has two parts: checking for duplicates (hash and then search a small set of past states) and then either adding the new state or updating the duplicated state.

Whether this parallel algorithm improves or degrades performance depends on the architecture of the machine on which the program executes. The expanders need to be on different physical processors to have true concurrency. Unfortunately, this might cause the access time to the shared memory containing the state space array to be lengthened. The computational cost of constructing a next state is small enough that little additional communication cost can be tolerated in order to maintain a reasonable communication to computation ratio. Consequently, the architecture must be quite tightly coupled.

4. Approximate Models Using the GTPN

Parallelizing can only help to a limited extent. *State space explosion* remains a major problem in the GTPN. In many situations a net detailed enough to be a good model has too large of a state space. Consequently, approximation techniques should be considered. This, in fact, has already been done successfully. In [RAM86] Kishore Ramachandran uses the GTPN to model support for interprocess communication by front-end processors. A single net would have too large a state space. Instead the model contains two nets: one for the server and one for the client. Each net represents the other net by a state-dependent transition. Initial estimates are made for the firing duration of these two transitions. The nets are iteratively solved until convergence.

Extensive research has been done concerning approximate methods in queueing network models. De Souza e Silva, Lavenberg, and Muntz's paper [SOU84] is an excellent review of this work. It would be interesting to consider the applicability of the queueing network work to the GTPN.

5. New Problem Domains

Other modeling studies should be conducted. It is important to characterize the problem domains in which the GTPN is useful. Other areas, besides parallel architectures, could be considered. Network protocols and concurrent

software have been modeled using other forms of Petri Nets [GR85,STO85]. The GTPN may be useful in those areas also.

Two more specific possibilities are program behavior at the basic block level and reliability analysis. If a program is viewed as a sequence of basic blocks, then the program's mean time to completion could be computed. In this application the important issue is the assignment of the probabilities between basic blocks. In some cases (for example, *for* loops with immediate values as limits) probabilities can be determined. In other cases, bounding probabilities should be chosen.

The applicability of the GTPN to reliability analysis is suggested by the work by Marsan and Chiola [AJM85b]. In that work they investigate the reliability of a multiprocessor using the GSPN. The key is that the probability distribution (not just the mean) for each performance measure needs to be computed. Given the distribution, we can state, for example, not only the mean number of operational processors, but also what percentage of the time are at least x processors operational. The GTPN does compute the probability distribution for each performance measure, so the GTPN might also be useful in reliability analysis.

6. Alternative Model Description Languages

   The use of a form of Petri Nets as the system modeling language might not be essential. The analysis of the state space and perhaps some of the state space construction algorithms seem independent of the modeling language. The effectiveness of other modeling languages should be explored.

7. Verification

   Verification of logical properties has been studied using Petri Nets (as well as other formalisms, such as temporal logic). Perhaps the GTPN can be used for verification as well as performance analysis.

Bibliography

[AJM84] Ajmone Marsan, M., G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets", *ACM Trans. on Computer Systems*, Vol. 2, May 1984, pp. 93-122.

[AJM82b] Ajmone Marsan, M., G. Balbo, and G. Conte, "Comparative performance analysis of single bus multiprocessor architectures," *IEEE Trans. Comput.*, vol. C-31, pp. 1179-1191, Dec. 1982.

[AJM83] Ajmone Marsan, M., G. Balbo, G. Conte, and Gregoretti, "Modeling Bus Contention and Memory Interference in a Multiprocessor System," *IEEE Trans. Comput.*, vol. C-32, pp. 60-72, Jan. 1983.

[AJM85a] Ajmone Marsan, M., G. Chiola, and G. Conte, "Generalized Stochastic Petri Net Models of Multiprocessors with Cache Memories," *1st Int. Conf. on Supercomputing Systems*, December 1985.

[AJM85b] Ajmone Marsan, M. and G. Chiola, "On Petri Nets with Deterministic and Exponential Transition Firing Times," private communication, 1985.

[AJM85c] Ajmone Marsan, M. et al., "On Petri Nets with Stochastic Timing," *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[AJM82a] Ajmone Marsan, M. and M.Gerla, "Markov models for multiple-bus multiprocessor systems," *IEEE Trans. Comput.*, vol. C-31, pp.239-248, Mar. 1982.

[ARC85] Archibald, J., and J.-L. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors," private communication, July 1985.

[BAS76] Baskett, F.S. and A.J. Smith, "Interference in multiprocessor computer systems with interleaved memory," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 327-334, June 1976.

[BER79] Berman, A. and Plemmons, R.J., *Nonnegative Matrices in the Mathematical Sciences*, Academic Press, New York, NY, 1979.

[BHA75] Bhandarkar, D.P., "Analysis of memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-24, pp. 897-908, Sept. 1975.

[BHA73] Bhandarkar, D.P. and S.H. Fuller, "Markov chain models for analyzing memory interference in multiprocessor systems," *Proc. 1st Annu. Symp. Comp. Arch.*, pp. 1-6, Dec. 1973.

[BHU84] Bhuyan, L.N., "A Combinatorial Analysis of Multibus Multiprocessors," in *Proc. 1984 Int. Conf. on Parallel Processing*, Aug. 1984, pp.225-227.

[CHR83] Chretienne, P., "Les réseaux de Petri temporisés," Thèse d'état, Paris-6 University, June 1983.

[CHI85] Chiola, G., "A Software Package for the Analysis of Generalized Stochastic Petri Net Models," *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[CIN75] Cinlar, E., *Introduction to Stochastic Processes,* Prentice-Hall, Englewood Cliffs, NJ 1975.

[COO83] Coolahan, J.E. and N. Roussopoulos, "Timing Requirements for time-driven systems using augmented Petri nets," *IEEE Trans. on Software Engineering*, Vol. SE-9, September 1983, pp. 603-616.

[CUM85] Cumani, A., "ESP—A Package for the Evaluation of Stochastic Petri Nets with Phase-Type Distributed Transition Times", *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[DUB82] Dubois, M., and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Trans. on Computers*, Vol. C-31, November 1982, pp. 1083-1099.

[DU83] Du, H.C. and J.L. Baer, "On the performance of interleaved memories with non-uniform access probabilities," in *Proc. 1983 Int. Conf. on Parallel Processing*, Aug. 1983, pp. 429-436.

[DUG84] Dugan, J.B., K.S. Trivedi, R.M. Geist, and V.F. Nicola, "Extended Stochastic Petri Nets: Applications and Analysis," *Performance 84*, pp. 507-519, Paris, France, December 1984.

[FRA84] Frank, S.J., "Tightly Coupled Multiprocessor System Speeds Memory Access Times," *Electronics*, Vol. 57, no. 1, January 1984, pp. 164-169.

[GOO83] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," in *Proc. of 10th Int. Symp. on Computer Architecture*, June 1983, pp. 124-131.

[GOY84] Goyal, A. and T. Agerwala, "Performance Analysis of Future Shared Storage Systems," *IBM J. Res. Develop.*, vol.28-1, pp. 95-108, Jan. 1984.

[GR85] Gressier, E., "A Stochastic Petri Net Model for Ethernet," *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[HOL85a] Holliday, M. A., and M. K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis," *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[HOL85b] Holliday, M. A., and M. K. Vernon, "Exact Performance Estimates for Multiprocessor Memory and Bus Interference", to appear in *IEEE Trans. on Computers*, also Technical Report #594, Comp. Sci. Dept., UW-Madison, May 1985.

[HOL85c] Holliday, M.A. and M.K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis (extended version)," to appear in *IEEE Trans. of Software Engineering*, also Tech. Rep. 593, Comp. Sci. Dept., UW-Madison, May 1985.

[HOL86] Holliday, M.A. and M.K. Vernon, "The GTPN Analyzer: Numerical Methods and User Interface,", submitted for publication, also Tech. Rep. #639, Comp. Sci. Dept., UW-Madison, April 1986.

[HOO77] Hoogendoorn, C.H., "A general model for memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-26, pp. 998-1005, Oct. 1977.

[IRA84] Irani, K.B. and I.H. Önyüksel, "A Closed Form Solution for the Performance Analysis of Multiple-Bus Multiprocessor Systems," *IEEE Trans. Comput.*, vol. C-33, pp. 1004-1012, Nov. 1984.

[JAC82] Jacobson, P.A. and E.D. Lazowska, "Analyzing queueing networks with simultaneous resource possession," *Commun. Ass. Comput. Mach.*, vol. 25, pp.142-151, Feb. 1982.

[JOH82] Johnson, L.W. and R.D. Riess, *Numerical Analysis, Second Edition*, Addison-Wesley, Reading, MA, 1982.

[KAT85] Katz, R., S. Eggers, D.A. Wood, C. Perkins, and R.G. Sheldon, "Implementing a Cache Consistency Protocol," *Proc. of 12th Int. Symp. on Computer Architecture*, June 1985, pp. 276-283.

[KEM76] Kemeny, J.G. and J.L. Snell, *Finite Markov Chains*. New York, NY: Springer-Verlag, 1976.

[LAN82] Lang, T., M. Valero, and I. Alegre, "Bandwidth of crossbar and multiple-bus connections for multiprocessors," *IEEE Trans. Comput.*, vol. C-31, pp.1227-1234, Dec. 1982.

[MCC84] McCreight, E., "The DRAGON Computer System: An Early Overview," *NATO Advanced Study Institute on Microarchitecture of VLSI Computers,* Urbino, Italy, July 1984.

[MOL81] Molloy, M.K., "On the integration of delay and throughput measures in distributed processing models," Ph.D dissertation, Univ. California, Los Angeles, 1981.

[MOL82] Molloy, M. K., "Performance Analysis Using Stochastic Petri Nets", *IEEE Trans. on Computers*, Vol. C-31, Sept. 1982, pp. 913-917.

[MOL85] Molloy, M. K., "Discrete-Time Stochastic Petri Nets", *IEEE Trans. on Soft. Engr.*, Vol. SE-11, April 1985, pp. 417-423.

[MUD84] Mudge, T.N., J.P. Hayes, G.D. Buzzard, and D.C. Winsor, "Analysis of Multiple Bus Interconnection Networks," in *Proc. 1984 Int. Conf. on Parallel Processing*, Aug. 1984, pp. 228-232.

[MUD82] Mudge, T.N. and B.A. Makrucki, "Probabilistic analysis of a crossbar switch," in *Proc. 9th Int. Symp. on Comput. Arch.*, Apr. 1982, pp. 311-320.

[NAT80] Natkin, S., "Reseaux de Petri Stochastiques", These de Docteur-Ingenieur, CNAM-Paris, June 1980.

[NUT72] Nutt, G.J., "Evaluation nets for computer systems perf. analysis," in 1972 *Fall Joint Computer Conf., AFIPS Conf. Proc.*, vol.41., Montvale, N.J.: AFIPS Press, 1972, pp. 279-286.

[ONY83] Önyüksel, I.H. and K.B. Irani, "A Markovian queueing network model for performance evaluation of bus-deficient multiprocessor systems," in *Proc. 1983 Int. Conf. on Parallel Processing*, Aug. 1983, pp.437-439.

[PAN83] Pang, N. and J.E. Smith, *CRAY-1 Simulation Tools*, Tech.Rep. ECE-83-11, Dept. of ECE, UW-Madison, Dec. 1983.

[PAP84] Papamarcos, M., and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. of 11th Int. Symp. on Computer Architecture,* June 1984, pp. 348-354.

[PET81] Peterson, J.L., *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[PET62] Petri, C.A., "Kommunikation mit Automaten," *Schriften des Rheinisch-Westfalischen Institute fur Instrumentelle Mathematick an der Universitat Bonn,* Heft 2, Bonn, W. Germany, 1962; translation: C.F. Greene, Supplement 1 to Tech. report RADC-TR-65-337, Vol. 1, Rome Air Development Center, Grifiss Air Force Base, NY 1965.

[PIS84] Pissanetzky, S., *Sparse Matrix Technology,* Academic Press, Orlando, Florida, 1984.

[RAM86] Ramachandran, K., "Hardware Support for Interprocess Communication," Ph.D. thesis, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1986.

[RAM80] Ramamoorthy, C.V. and G.S. Ho, "Performance Evaluation of Asynchronous Systems using Petri nets," *IEEE Trans. Software Engr.*, vol. SE-6, pp. 440-449, Sept. 1980.

[RAM74] Ramchandani, C., "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets," Ph.D. Thesis, MIT, 1974.

[RAU79a] Rau, B.R., "Program Behavior and the Performance of Interleaved Memories," *IEEE Trans. Comput.*, vol. C-28, pp.191-199, March 1979.

[RAU79b] Rau, B.R., "Interleaved memory bandwidth in a model of multiprocessor computer systems," *IEEE Trans. Comput.*, vol. C-28, pp. 678-681, Sept. 1979.

[RAZ84] Razouk, R.R. and C.V. Phelps, "Performance Analysis Using Timed Petri Nets," in *Proc. 1984 Int Conf. on Parallel Processing*, pp. 126-129, August, 1984.

[RUD84] Rudolph, L., and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. of 11th Int. Symp. on Computer Architecture,* June 1984, pp. 340-347.

[SAU81] Sauer, C.H. and K.M. Chandy, *Computer Systems Perf. Modeling*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[SED81] Sedgewick, R., *Algorithms.* pp. 428-430, Reading, MA: Addison-Wesley, 1983.

[SEN81] Seneta, E., *Non-negative Matrices and Markov Chains, Second Edition,* Springer-Verlag, NY, 1981.

[SET79] Sethi, A.S. and N.Deo, "Interference in multiprocessor systems with localized memory access probabilities," *IEEE Trans. Comput.*, vol. C-28, pp. 157-163, Feb. 1979.

[SIO83] Siomalas, K.O. and B.A. Bowen, "Performance of crossbar multiprocessor systems," *IEEE Trans. Comput.*, vol. C-32, pp.689-695, July 1983.

[SMI82] Smith, A.J., "Cache Memories," *Computing Surveys*, Vol. 14, no. 3, pp. 473-530, September 1982.

[SMI85a] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice", *Proc. of 12th Int. Symp. on Computer Architecture*, June 1985.

[SMI85b] Smith, A. J., "Line (Block) Size Choice for CPU Cache Memories", Technical Report CSD 85/239, Computer Science Division, Univ. of Calif. at Berkeley, 1985.

[SOU84] de Souze e Silva, E., S.S. Lavenberg, and R.R. Muntz, "A Perspective on Iterative Methods for the Approximate Analysis of Closed Queueing Networks," Tech. Report RC 10141, August 1983, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[STE73] Stewart, G.W., *Introduction to Matrix Computations,* Academic Press, New York, NY, 1973.

[STE78] Stewart, W.J., "A Comparison of Numerical Techniques in Markov Modeling," in *Communications of the ACM*, Vol.21, No.2, February 1978, pp.144-152.

[STE85] Stewart, W.J. and A. Goyal, "Matrix Methods in Large Dependability Models," Tech. Report RC 11485, November 1985, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[STO85] Stotts, P.D. Jr., and T.W. Pratt, "Hierarchical Modeling of Software Systems with Timed Petri Nets," *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[STR70] Strecker, W.D., "Analysis of the instruction execution rate in certain computer structures," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1970.

[SYM80] Symons, F.J.W., "Introduction to numerical Petri nets, a general graphical model of concurrent processing systems," A.T.R., vol. 14, Jan. 1980.

[TAR76] Tarjan, R.E., "Graph Theory and Gaussian Elimination", in *Sparse Matrix Computations*, J.R. Bunch and D.J. Rose, editors, New York, NY: Academic Press, 1976.

[TAY84] Taylor, H.M. and S. Karlin, *An Introduction to Stochastic Modeling*, Orlando, FL: Academic Press, 1984.

[TOW83] Towsley, D., "An approximate analysis of multiprocessor systems," in *Proc. 1983 ACM Sigmetrics Conf. on Meas. and Mod. Comput. Syst.*, Aug. 1983, pp. 207-213.

[VER86] Vernon, M.K. and M.A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," to appear in *Performance '86 and ACM SIGMETRICS '86 Joint Conference on Computer Performance Modeling, Measurement and Evaluation*, also Technical report #618, Computer Sciences Dept., UW-Madison, November 1985.

[WAL66] Wallace, V.L. and R.S. Rosenberg, "The Recursive Queue Analyzer," Systems Engineering Dept., Tech. Report #2, Univ. of Michigan, Ann Arbor, MI, 1966.

[WIL65] Wilkinson, J.H., *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, Great Britain, 1965.

[WUL72] Wulf, W.A. and C.G. Bell, "C.mmp – A multi-mini-processor," in *Fall Joint Comput. Conf. AFIPS Conf. Proc.*, vol. 41, pt. 2, 1972, pp.765-777.

[YEN82] Yen, D.W., J.H. Patel, and E.S. Davidson, "Memory interference in synchronous multiprocessor systems," *IEEE Trans. Comput.*, vol. C-31, pp. 1116-1121, Nov. 1982.

[ZUB80] Zuberek, W.M., "Timed Petri nets and preliminary performance evaluation," in *Proc. 7th Annu. Symp. Comput. Architecture*, pp. 88-96, 1980.