Order Number 1353098

Performance considerations relating to the design of interconnection networks for multiprocessing systems

Hokens, Earl Edwin, III, M.S.

The University of Arizona, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PERFORMANCE CONSIDERATIONS RELATING TO THE DESIGN OF INTERCONNECTION NETWORKS FOR MULTIPROCESSING SYSTEMS

by

Earl Edwin Hokens III

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE
WITH MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 3

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Earl E Hoel III

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Ahmed Louri
Assistant Professor of
Electrical and Computer Engineering

5/6/93
Date

ACKNOWLEDGMENTS

I would like to acknowledge the guidance and direction given by my advisor Dr. Ahmed Louri. I also acknowledge the contributions of Dr. William Sanders, Bruce McLeod, Doug Obal, Luai Malhis, and Latha Kant, for their assistance with *UltraSAN*, and SAN model development. I would also like to thank John Staab, and Brian Estberg for the assistance given with their HP calculators for solving equations, and everyone else who assisted in the preparation and proof-reading of this thesis.

DEDICATION

I would like to dedicate this thesis to all of my friends and family both alive and dead, especially my grandmother Elizabeth Koval, and friend John "Lush" Vogl.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The interconnection network is the single most important element of a multiprocessor. Choosing the best interconnection scheme for a given sized multiprocessor is no easy task. This paper presents a two phase analysis to assist in the design of multiprocessor interconnection networks. The two phases of the analysis are analytical, and simulation. The analytical phase introduces a new metric called the network bandwidth requirement, or *nbr*. This is an estimate for the interconnecting network link speed required for a multiprocessor of a given size. The *nbr* result is used in the simulation phase of the analysis. Various multiprocessor configurations are simulated using stochastic activity networks to verify the results of the analytical phase. The *nbr* is shown to be an excellent estimate for the interconnection network link speed required in a multiprocessor. The simulation may also be used to analyze various multiprocessor characteristics.

CHAPTER 1

INTRODUCTION

Processing need is surpassing current limits of single processor technology [1]. This is pushing the envelope of computer design deeper into the realm of parallel processing where the single most important issue is the interconnection network [1, 2, 3, 4].

After selecting a processor for use in a multiprocessing computer, a designer must mate it with an appropriate interconnection network. An interconnection network is appropriate when it meets or exceeds the required performance and cost requirements. Many factors must be considered in this choice, processor characteristics, and current technological limits to name a couple.

There has been extensive research on the subject of interconnection networks for parallel processors to determine which are the most capable. However, previous research has concentrated on the interconnection network as an isolated entity and reports a throughput for the interconnection network as a whole [5, 6, 7, 8, 9, 10], where throughput is in transactions per cycle. The throughput is for the entire interconnection network, the units are transactions per cycle (transactions are not all the same size), and an interconnecting network link speed is not considered. This work complements previous work by extending it to predict an interconnection link speed. This thesis will give a computer

designer insights on how to select an interconnection network for use in a tightly coupled multiprocessor, and suggest an interconnecting link speed required to sustain a reasonable level of performance. While this analysis is concerned with multiprocessors, it can also be applied to multicomputers and computer networks.

Our approach to this problem is from a different perspective than previous research; our analysis centers on the individual interconnecting links as part of a complete multiprocessor. Rather than calculating the throughput for the entire network, we calculate a bandwidth requirement in words per cycle for the interconnecting links. Processor characteristics, memory issues, data consistency issues are all considered in addition to interconnection network issues.

We propose a two step method for this problem that results in an estimate for the interconnecting link speed. The first step is an analytical method that provides the estimate for the interconnecting link speed required to sustain processor demands. We refer to this estimate as a new metric called the network bandwidth requirement, or *nbr*. The units for the *nbr* are words per cpu cycle. To simplify this step, memory characteristics and contention issues were deferred to the second step, which is simulation.

The main purpose of simulation is to verify the analytical results. Simulation also allows for a more detailed analysis of multiprocessor issues considered in the analytical phase as well as the analysis of issues not considered in the analytical phase. These issues include contention, memory configurations, and data consistency. Using the estimate for the interconnecting link speeds obtained in the analytical portion, a model is developed using stochastic activity networks, or SANs [11, 12, 13]. The processor, interconnection

network and memory are modeled providing performance analysis for each part of the multiprocessor.

The processing element in this thesis is assumed to be a reduced instruction set computer (RISC) type whose access to memory is limited to load and store instructions. The interconnection network types evaluated are: bus, crossbar, mesh and multistage interconnection network (MIN).

Chapter 2 discusses the background information necessary for this thesis. Chapter 3 derives the analytical equations that represent the mechanism by which requests are generated and propagated, as well as the *nbr* expression. Chapter 4 discusses the analytical results and how they were generated. Chapter 5 presents the modeling and simulation phase using the simulation tool *UltraSAN* [11, 12, 13]. Chapter 6 presents the simulation results and discusses their implications. Chapter 7 concludes this thesis.

CHAPTER 2

BACKGROUND

In order to clearly understand the material presented in this thesis, some characteristics and assumptions about the processors, memory, and the various interconnection networks are stated in this chapter. We will first discuss processor characteristics, then interconnection network characteristics, and finally memory characteristics.

2.1 PROCESSORS

With the advent of the microprocessor, large scale parallelism has become feasible [14]. The latest entry in the microprocessor arena are the Reduced Instruction Set Computer processors, or RISC processors. We have selected RISC type features for our processor representation. This selection was made for several reasons [15], the main reason being RISC architectures in general have higher bandwidth requirements than their Complex Instruction Set Computer counterparts. This is because the reduced complexity of RISC instructions require more instructions to be executed to accomplish the same task. It also allows instructions to be executed faster. Therefore, more data is required by a processor in less time.

There are three major points considered for processor representation: (1) word size, (2) instruction issue rate, and (3) cache issues. 1. The word size is defined to be the width of the interface to the processor chip. 2. The data request rate is slightly more complex to represent. However, data manipulating instructions in RISC architectures are typically register to register and they generate no data transfers off of or on to the processor chip. Only load and store instructions can generate these transfers. Therefore, we need only consider load and store instructions. Also note that the maximum rate which data needs to be sent to (or from) the processor is bounded by the maximum rate it can be requested (or sent). 3. The final point, cache issues, is comprised of several parts. It is assumed that our processors contain an on chip cache. The time penalty for a data hit in this on chip cache is assumed to be zero. The other item for the cache is the cache line size. The purpose of a cache is to reduce memory latency [16, 17, 18]. This is accomplished by not just reading data one word at a time and sending it to the processor, but by reading several words, a line, for each requested word and placing these into the cache for future requests. The line size is assumed to be an integer multiple of the word size.

2.2 INTERCONNECTION NETWORKS

In order to support processor requests, the connection between memory and the processor(s) must be appropriately matched. This connection is referred to as an interconnection network. The structure may range from a single link to any complex structure. In this thesis four interconnection network types are considered, bus, crossbar, mesh, and Multistage Interconnection Network (or MIN). While there are variations on the basic theme,

all exhibit certain characteristics that typify their class [19, 20, 2]. These properties are necessary for the analysis in this thesis.

It is desirable to maintain each processor's performance within a multiprocessor at the same level obtained in a single processor system. This was the driving assumption made in identifying characteristics for the interconnection network analysis presented here. In other words we would like to have each processor behave as if it were directly connected to its own memory. There are two major factors that must be considered in estimating the speed required of the interconnection network's links in order to appear as a single dedicated link: the number of elements contributing requests to each link, and the average distance these requests must travel to reach their destinations. A designer must also realize that there are factors that work against this goal. One such factor is contention. Contention serves to reduce the actual bandwidth that could theoretically be obtained. Unfortunately, contention is inherent in a multiprocessor system, and there is no way to eliminate the possibility of contention once an interconnection network is selected. For our analytical phase we optimistically assume contention does not exist, although we fully understand that it does. The reasons for this are discussed more later.

2.3 MEMORY

The final portion of a multiprocessor system is memory. There are many ways memory can affect the traffic through an interconnection network. Obviously, if memory can only service requests half as fast as they arrive, there will be less traffic than if it could service them at the same rate. Like contention this can only serve to reduce the actual bandwidth

obtained, it cannot increase the bandwidth appreciably. For this reason our analysis assumes that the memory services requests at least as fast as they arrive. It is up to the designer to maximize the speed at which memory can service requests. This can be done through such vehicles as memory interleaving, using faster technologies and additional cache levels.

Keeping these issues in mind we can now proceed to the derivation of the equations. Some of the issues mentioned above will be discussed in more detail when needed. Please note that issues not considered in the analytical phase are considered in the simulation phase.

CHAPTER 3

ANALYTICAL PHASE

3.1 MULTIPROCESSOR MODEL AND ISSUES CONSIDERED

The assumed multiprocessor model is the shared-memory multiprocessor shown in

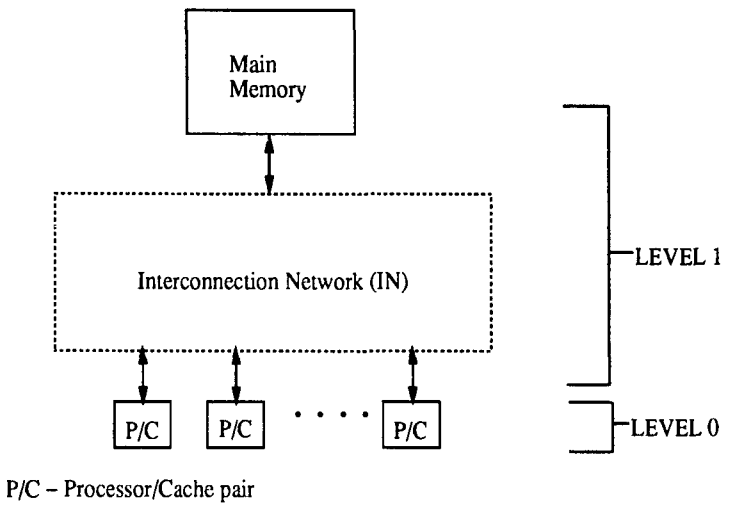


Figure 3.1: Multiprocessor model.

Figure 3.1. The processing element P/C, is a processor with private cache. The processor cache pairs are assumed to be identical in every way. They are all of the same speed, have the same hit ratio, and so on. Level 1 consists of the interconnection network (IN) and connects the P/C elements of level 0 to main memory. The interconnection network

may be a bus, mesh, MIN, or crossbar. There are no assumptions about the uniformity of memory access times.

The purpose of the analytical equations is to provide a speed estimate for the interconnection links. Four factors have been identified for this analysis: (A) cache line size, (B) memory cycle, (C) contention, and (D) data consistency issues. The first part of this section discusses how these issues relate to this analysis. The equations are then derived after defining some terms.

A. Cache Line Size

Cache line size is an important item to consider due to the nature of data transfer. When a data miss occurs the entire cache line that contains the referenced word is transferred, not just the requested word. A *word* is defined as the width of the data bus(link). If the cache line is larger than *one* word, multiple link transactions must occur to completely transfer the cache line. Therefore, when a data line is sent from main memory we would like to have the entire transfer complete in the same time a one word transfer would take. We assume that memory tries to send all the words in a line at once. There are other methods which first send the requested word, and then the rest of the line. These are not considered here as they serve to reduce the amount of interconnection network traffic by deferring some transfers until the network is free.

B. Memory Cycle

Memory cycle is the amount of time it takes for the memory to satisfy a request. The mechanism employed to access information, i.e. interleaving memory, overlapping requests, etc, may be analyzed in the simulation phase of the analysis. The rationale

for deferring this until simulation is that the focus of the analytical phase is on the speed requirements for the interconnection links. Regardless of how much time it takes to satisfy a request the same number of transactions will be traversing the interconnection network. To develop a maximal speed estimate for the links memory is assumed to be able to satisfy a request immediately. Memory can only act as a bottleneck reducing the overall throughput, and can never improve it substantially. Reducing the memory cycle time only reduces the latency by the change in cycle time, typically a small change. Deferring memory analysis until simulation also reduces the complexity of the equations. Furthermore, given that the object of the analytical equations is to generate a speed estimate for the links, including the memory cycle becomes impossible. Including the memory cycle causes the equations to develop a time dependency. To accurately determine when a link is affected, assumptions about the relative speeds of the memory, the processor and the interconnection links must be made which defeats the purpose of this analysis.

C. Contention

Contention comes in two flavors: module or resource contention, and link contention. Once again including contention causes the equations to develop a time dependency. A particular link or resource is contended for when *two* requests arrive for processing at the same time. To determine this, it must be known when and where each transaction originated, what its destination is, the path it takes and how long it takes. An extremely large task, which once again assumes the speed of the interconnecting links are known.

A probabilistic approach where each transaction has a certain possibility of blocking is also possible. This type of analysis was done in [21]. The analysis in [21] assumes the cycle

time through the network is known, and determines the throughput of the network rather than the interconnecting link bandwidth as is done in this thesis. A similar probabilistic approach could be taken, however, to accurately determine the probability of blocking P_b, the cycle time through the network must be known, once again defeating the purpose of this analysis. Instead of calculating P_b, a representative value could be assumed and used to increase the processor request rate by an amount corresponding to the expected number of requests rejected. This new value could provide a more "realistic" speed for the links but how realistic depends on the assumption made for the value of P_b and the rate of requests, which is also required in [21]. Rather than choose representative values of P_b and request rate, we have assumed the processor is issuing requests 100% of the time. being fully aware that a certain percentage of these will be rejected. Previous work [22. 23] suggests that typically requests are issued between 10 and 60% of the time. Also P_b ranges between 0 and 60% for the bus, the crossbar, and the MIN. The mesh construction differs significantly from those of the bus, crossbar and MIN, and contention does not exist in the mesh as it does in the other network types since requests are queued and not discarded [2. 24]. A quick comparison of the numbers shows they overlap. This indicates that the assumption could break down under certain conditions. This break down will occur under high request rates, and networks that exhibit high contention. The analysis of contention is deferred to the simulation phase of the analysis.

D. Maintaining Data Consistency

Maintaining Data Consistency is perhaps the most influential of the factors listed [18, 25, 26, 27]. Data Consistency is the maintaining of data in such a way that only current up

to date data is used by any processor. A large amount of traffic is generated maintaining consistency. These issues will be discussed in depth in the following subsections.

3.1.1 ASSUMPTIONS AND DEFINITIONS FOR THE ANALYSIS

A. Assumptions

The analysis presented here makes some base assumptions, the first five of which are the same as in previous research [8, 7, 6, 5, 9].

1. Requests are uniformly distributed across memory.
2. Operation of the interconnection network is synchronous, messages begin and end simultaneously.
3. Requests by processors are independent of each other.
4. Requests issued at a cycle are independent of requests issued in the previous cycle.
5. Requests may only be issued at the start of the network cycle.
6. Cycle time of the interconnection network is assumed to be equal to the processor cycle.
7. Each processor is assumed to issue either a load or a store operation at its maximum rate.

Other assumptions used in the analysis will be made when appropriate.

B. Symbols

Before deriving the equations, we present a summary of symbols and their definitions as used in this thesis. Table 3.1 shows these symbols. Some of these terms will be defined in more detail when needed.

3.2 OVERHEAD CALCULATIONS FOR THE ASSUMED MODEL

Overhead represents the average number of transactions needed by each processor to maintain the data consistency of the system. A representative scheme was developed

Table 3.1: Symbols and their definitions used in the analytical phase for single level multiprocessors

Symbol	Definition
P_r	probability that a memory request is a read
P_s	probability that datum is shared
P_d	probability that a shared datum is dirty
S	percentage of processors sharing datum
$1-h$	miss rate of the processor cache
N	total number of processors in the system
D	total number of data transfers per memory request
L	size of the cache line transferred into the processor cache
N_L	number of transaction sources per link at level i
P_L	average path length a transaction must travel to reach its destination
oh_{rds}	average overhead caused by reads to dirty shared data
oh_{rs}	average overhead caused by reads to shared data
oh_{ru}	average overhead caused by reads to unshared data
$ohinv$	average overhead due to invalidations for each transaction
oh_{wu}	average overhead caused by writes to unshared data
oh	overhead incurred for each transaction
nbr	network bandwidth requirement (words/cycle)

through a survey of directory based coherency schemes [28, 25, 26] and used to develop these equations. There are six cases by which traffic may be induced in the interconnection network: (A)reading dirty shared data, (B)reading shared data, (C)reading unshared data, (D)writing shared data, (E)writing unshared data, and (F)data transfers. The analysis separates consistency transactions from data transfers to simplify equation development. The basis of the equations derived here are on a per processor contribution to interconnection network traffic.

A. Overhead Generated by Reading Dirty Shared Data

When a processor attempts to read dirty shared data, the dirty item may be stored in the processor's cache, or it may not. In either case a dirty item is considered unusable. For

the reading of dirty shared data the hit ratio is unimportant, because an attempt to read dirty data that is contained in a processor's cache is a forced miss. In the representative scheme an attempt to read dirty shared data is accomplished in *three* steps. First, the processor cache pair issues a read to main memory. Second, upon receipt of the read request, main memory recognizes that the data is dirty, and forwards the request to the processor with the clean copy which sends the cache line containing the data to the processor that initially requested the data. This is a net of *three* transactions. However, since data transfers are considered separately, overhead due to reading dirty shared data is:

$$oh_{rds} = 2 \times P_r \times P_s \times P_d \quad \text{words per memory request.} \tag{3.1}$$

In words, for each instruction issued, P_r of them are reads. Of the data being read, P_s is shared and P_d of the shared data is dirty. For every read of this type, *two* one word transactions are generated.

B. Overhead Generated by Reading Shared Data

When a processor attempts to read shared data no overhead is generated when the read hits because the datum is located in the processor's own cache, therefore, overhead is produced only when a read miss occurs. On a read miss to shared data *two* steps must occur in accordance with the representative scheme: First, the processor cache pair issues a read request to main memory, upon receipt of this read request, the main memory returns the cache line containing the datum to the processor. The main memory copy is assured to be valid since it is not dirty. As in the reading dirty shared data case. This is a net of *two* transactions. Once again the data transfer is considered separately, and

overhead due to reading shared data is:

$$oh_{rs} = P_r \times P_s \times (1 - P_d) \times (1 - h) \quad \text{words per memory request.} \tag{3.2}$$

In words, for each read instruction, P_s of the data accessed is shared, $(1 - P_d)$ of it is clean, and $(1 - h)$ of it is not contained in the cache. For read transactions of this type a single *one* word transaction is generated.

C. Overhead Generated by Reading Unshared Data

As in the previous case, when a processor attempts to read unshared data no overhead is generated on a data hit. We only need to be concerned with a data miss. The handling of a miss on a read of unshared data is the same as in case B, a request to memory and data returned from memory. Data is assured to be valid since no other processor is sharing it. Remembering that data transfers are considered later, overhead for reading unshared data is:

$$oh_{ru} = P_r \times (1 - P_s) \times (1 - h) \quad \text{words per memory request.} \tag{3.3}$$

In words, of the memory requests issued P_r are reads. Of the data being accessed $(1 - P_s)$ is not shared, and of that data $(1 - h)$ of it is not contained in the processors cache. When this situation occurs, a single *one* word transaction is needed to initiate the data transfer.

D. Overhead Generated by Writing Shared Data

On a write to shared data either a hit or a miss occurs. In either case all other memory copies must be invalidated, including the main memory copy. For each invalidation two transactions must occur, invalidation, and acknowledgement. In the case of a miss, a data

transfer must occur as well, this is considered later.

$$ohinv = (1 - P_r) \times (2P_s(1 + S \times N)) \quad \text{words per memory request.} \tag{3.4}$$

In words, for every write to shared data $2\times(1+S\times N)$ invalidations and acknowledgements are necessary.

E. Overhead Generated by Writing Unshared Data

A write to unshared data generates overhead on a write hit and a write miss. Upon a write hit, *two* transactions of *one* word are generated. First, the processor cache pair issues an invalidation to main memory. Second, main memory returns an acknowledgement to the processor. On a write miss, two transactions are also generated. First, the processor cache pair sends a read exclusive to the memory. Second, the memory returns the cache line that contains the datum. However, since the second transaction on a write miss to unshared data is a data transfer and is considered later, the overhead due to a write to unshared data is

$$oh_{wu} = (1 - P_r) \times (1 - P_s) \times ((1 - h) + 2h) \quad \text{words per memory request.} \tag{3.5}$$

By summing these equations a unique representation of the overhead may be obtained. This results in

$$oh = oh_{rds} + oh_{rs} + oh_{ru} + ohinv + oh_{wu} \quad \text{words per memory request.} \tag{3.6}$$

F. Data Transfers

A data transfer must be initiated for every *miss* and also every *hit* on shared dirty data. Reading shared dirty data causes *two* data transfers to be initiated, first the processor

sends a clean copy of the data to the main memory, which forwards a copy to the requesting processor. Note that this assumption does not pertain to a bus operating with a snoopy protocol [27]. However, since this scheme is assumed for all interconnection network types, the *nbr* values obtained for a bus interconnection will be inflated. This results in

$$D = (1 - h) + 2 \times (h \times P_s \times P_d) \quad \text{data transfers per memory request.} \tag{3.7}$$

One data transfer on a miss, and 2 for hits to dirty shared data. Another data transfer has not yet been considered, instructions must also be fetched. There is a fundamental difference between instructions and data. Instructions do not incur any consistency overhead because they are read only data. The incorporation of instruction fetches yields the following equation:

$$D = 2 \times ((1 - h) + h \times P_s \times P_d) \quad \text{data transfers per memory request.} \tag{3.8}$$

Two data transfers are necessary for both misses and hits to dirty shared data. Note that there is no actual correspondence between data misses and instruction fetch misses, but for simplification of the equation. it is assumed that both occur simultaneously.

Before proceeding further the completeness of these equations should be illustrated. In order to do this the probabilistic values relating to transaction type (read or write) and the data being accessed must be separated from the parameters that govern the number of overhead transactions generated and the filtering affects of the interconnection network. After eliminating the parameters that govern the number of transactions generated from the above equations, the following equations are left:

$P_r \times P_s \times P_d$ for reading dirty shared data.

$P_r \times P_s \times (1 - P_d)$ for reading shared clean data.

$P_r \times (1 - P_s)$ for reading unshared data.

$(1 - P_r) \times P_s$ for overhead invalidations.

$(1 - P_r) \times (1 - P_s)$ for writing unshared data.

Expanding these equations yields:

$P_r \times P_s \times P_d$ for reading dirty shared data.

$P_r \times P_s - P_r \times P_s \times P_d$ for reading shared clean data.

$P_r - P_r \times P_s$ for reading unshared data.

$P_s - P_r \times P_s$ for overhead invalidations.

$1 - P_r - P_s + P_r \times P_s$ for writing unshared data.

Summing these equations and cancelling terms yields:

$$\begin{aligned} 1 &= P_r \times P_s \times P_d + P_r \times P_s - P_r \times P_s \times P_d + \\ &\quad P_r - P_r \times P_s + P_s - P_r \times P_s + \\ &\quad 1 - P_r - P_s + P_r \times P_s \\ 1 &= 1 \end{aligned}$$

Therefore, the equations account for all types of instructions.

3.3 NETWORK BANDWIDTH REQUIREMENT - NBR

We now introduce a new metric called the network bandwidth requirement, or *nbr*. The *nbr* is the average number of words per cpu cycle each link should be capable of transferring to or from each source. A source may be a processor or a cache. Transactions may only

be created by the processor private cache pair at level 0. The equation for the *nbr* was derived through a realization of factors that increase traffic on an interconnecting link. Assume that one processor in the system is connected to one link. We then know that the link must handle $D \times L + oh$ transactions for each request the processor generates. However, if an interconnection link is not dedicated to a source, multiple sources are connected to each link and the *nbr* must be increased by a factor of the number of sources per link (N_L). In a multiprocessor there may not be a direct connection between the source and destination, and a transaction may have to traverse multiple links to reach its destination. Ideally, we would like the cycle time of a transaction through a multiple path to be the same as that of a direct path. P_L is the average path length a transaction travels. This implies that each link must be P_L times faster to match the performance of a direct link, ignoring switching delays. The P_L calculations are based on characteristics of the interconnection network. It is the average distance from any source to the main memory. The factors of the *nbr* discussed thus far result in the average number of words the interconnection links must transfer for each transaction generated. Let the transaction generation rate be known as nbr_0, which is the maximum rate at which memory requests can be issued by processors. This yields the following equation:

$$nbr = nbr_0 \times P_L \times N_L \times (D \times L + oh) \quad \text{words per cpu cycle.} \tag{3.9}$$

Only single level multiprocessor interconnection strategies, Figure 3.1, are evaluated in this thesis, however, the equations may be applied to hierarchical interconnection networks with extensions presented in the next section.

3.4 EXTENSIONS FOR THE HIERARCHICAL MULTIPROCESSOR MODEL

The model for the hierarchical shared-memory multiprocessor is shown in Figure

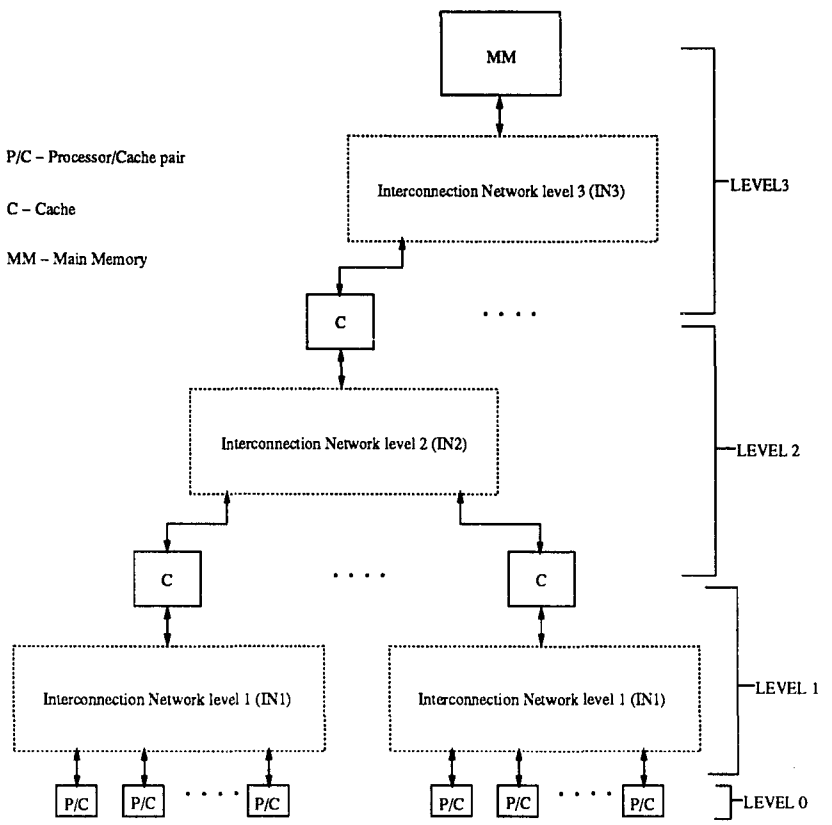


Figure 3.2: Hierarchical multiprocessor model.

3.2. The number of hierarchical levels may range from 1 to k where k is an arbitrary number. The processing element P/C, is a processor with private cache and corresponds to level 0 in Figure 3.2. Level 1 consists of the interconnection network type 1 (IN1) which connects the P/C elements of level 0 to level 1 memory elements, C. Level 2's interconnection network (IN2) connects the level 1 memory elements to level 2 memory

elements. This continues for all levels of the hierarchy. The interconnection networks at each level IN1, IN2, ... INk may be the same, or they may be different types. IN1 may be a bus while IN2 is a crossbar and IN3 a mesh, any configuration is possible, but not all may actually be realizable. Examples of some existing hierarchical multiprocessor systems include Stanford's DASH multiprocessor [29, 30], Cedar [31], Paradigm [32], and Hector [33].

Before presenting these equations, we present a summary of the symbols which differ in the hierarchical case, these are shown in table 3.2. Some of these terms will be

Table 3.2: Symbols and their redefinition for the hierarchical model.

Symbol	Definition
$1 - h_i$	global miss [18] at the level i cache
L_i	size of the cache line transferred into the level i cache
C_i	number of processors covered by the cache at level i
N_{L_i}	number of transaction sources per link at level i
D_i	number of data transfers per memory request at level i
P_{L_i}	average path length a transaction must travel to reach the main memory from level i
oh_{rds_i}	average overhead caused by reads to dirty shared data at level i
oh_{rs_i}	average overhead caused by reads to shared data at level i
oh_{ru_i}	average overhead caused by reads to unshared data at level i
$ohinv_i$	average overhead due to invalidations at level i for each transaction
oh_{wu_i}	average overhead caused by writes to unshared data at level i
oh_i	overhead incurred for each transaction at level i
nbr_i	network bandwidth requirement (words/cycle) at level i

defined in more detail when needed.

3.4.1 OVERHEAD EQUATIONS

Essentially, a simple substitution of the above variables are all that is needed with a few extensions. The six cases originally presented are now stated for the hierarchical model.

A. Overhead Generated by Reading Dirty Shared Data

There are no variables that depend on any hierarchical characteristics in this equation, requests of this type are assumed to travel up the entire hierarchy to the main memory. The equation is still:

$$oh_{rds_i} = 2 \times P_r \times P_s \times P_d \quad \text{words per memory request.} \tag{3.10}$$

B. Overhead Generated by Reading Shared Data

For transactions of this type, the hit ratio plays a role. The amount of traffic of this type at level i depends on the global miss rate of the previous level. This is because requests may be absorbed in caches of previous levels. If a request on its way to the main memory passes through a cache with a copy of the data item requested, the request may be intercepted. The equation for overhead generated by reading shared data now becomes:

$$oh_{rs_i} = P_r \times P_s \times (1 - P_d) \times (1 - h_{i-1}) \quad \text{words per memory request.} \tag{3.11}$$

C. Overhead Generated by Reading Unshared Data

Overhead of this type may be treated in the same way as the previous case of overhead generated by reading shared data. The equation is:

$$oh_{ru_i} = P_r \times (1 - P_s) \times (1 - h_{i-1}) \quad \text{words per memory request.} \tag{3.12}$$

D. Overhead Generated by Writing Shared Data

Unlike the other consistency cases, a write to shared data does not have a natural filter like the hit ratio. In the other cases, overhead transactions are filtered out by hits in downstream (closer to the main memory) caches. Invalidations, and their acknowledgements do not depend on hit ratios, they depend on the location of the processors that contain a copy of the datum. This is of importance in hierarchical interconnection networks. In hierarchical interconnections these values must be normalized by the percentage of processors not covered at that hierarchical level. For example consider the network of Figure 3.3. At level 1, each cache maintains the consistency of the processors directly

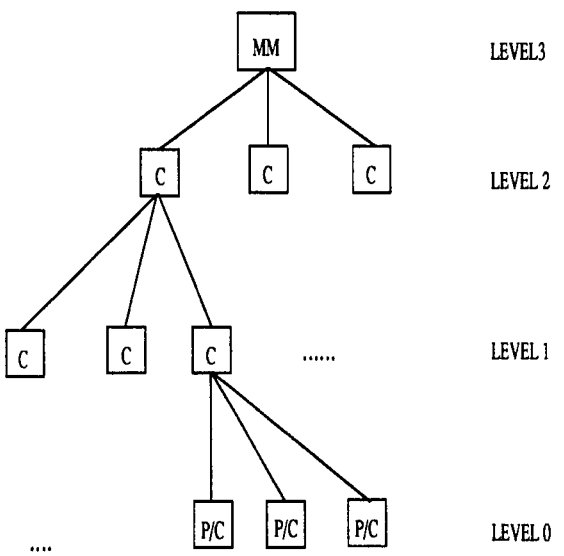


Figure 3.3: Sample interconnection used in the example of how to calculate C_i. P/C represents a processor cache pair, C is a cache and MM is main memory.

connected to it. In this example each cache maintains three processors. At level 2, each cache maintains the consistency of three level 1 caches, thereby maintaining nine processors, and so on until the highest level is reached. Therefore, at a level 1 cache, the

probability that an invalidation must be forwarded to level 2 is $(1 - \frac{3}{27})$. Or in general, the probability an invalidation at level $i-1$ needs to be forwarded to level i is $(1 - \frac{C_{i-1}}{N})$, where C_i is defined as before. The equation now becomes:

$$ohinv_i = (1 - P_r) \times (2P_s(1 + S * N)) \times (1 - \frac{C_{i-1}}{N}) \quad \text{words per memory request.} \quad (3.13)$$

E. Overhead Generated by Writing Unshared Data

A write to unshared data generates overhead on a write hit and a write miss. At level i the hit ratio of the level $i-1$ caches restrict traffic, so the equation now becomes:

$$oh_{wu_i} = (1 - P_r) \times (1 - P_s) \times ((1 - h_{i-1}) + 2h_{i-1}) \quad \text{words per memory request.} \quad (3.14)$$

By summing these equations the overhead at level i may be obtained. This results in

$$oh_i = oh_{rds_i} + oh_{rs_i} + oh_{ru_i} + ohinv_i + oh_{wu_i} \quad \text{words per memory request.} \quad (3.15)$$

F. Data Transfers

Once again it is the hit ratio of the caches at the previous level that restricts the amount of traffic at level i. Substituting in h_{i-1} for h we obtain the following equation:

$$D_i = 2 \times ((1 - h_{i-1}) + h_{i-1} \times P_s \times P_d) \quad \text{data transfers per memory request.} \quad (3.16)$$

3.4.2 NETWORK BANDWIDTH REQUIREMENT FOR THE HIERARCHICAL MODEL

Substituting in the variables that now have hierarchical implications and considering that as each transaction travels towards main memory it is joined by transactions from other sources, the hierarchical version of the *nbr* may be obtained. Referring back to

Figure 3.3, transactions from level 0 to level 1 join at a level 1 cache. Those transactions not filtered out will be forwarded to main memory over a fixed number of links, in this example *one*. The number of transactions is proportional to the number of processors covered by the level 1 cache. The number of transactions leaving a source at level i is C_{i-1} times higher. With this new factor, the *nbr* at level i is given by:

$$nbr_i = nbr_0 \times P_{L_i} \times N_{L_i} \times C_{i-1} \times (D_i \times L_i + oh_i) \quad \text{words per cpu cycle.} \qquad (3.17)$$

CHAPTER 4

RESULTS OF THE ANALYTICAL PHASE

In this thesis, we only consider single level interconnection networks. The probability variables P_r, P_s, P_d, and S were assumed to take on typical values found in simulation papers [27, 34, 35]. P_r was assumed to take on values between 0.4 and 0.8, P_s between 0.05, and 0.25, P_d between 0.1 and 0.5 and S between 0.06, and 0.1. The parameters were varied one at a time, those not varied were set at their initial value. The initial values are: $P_r = 0.8$, $P_s = 0.05$, $P_d = 0.1$, $S = 0.06$. This was done to isolate the affect each parameter has on the interconnection network traffic.

The hit ratio h also assumes typical values. The local hit ratio of a cache is defined in [18] as the number of cache hits divided by the number of requests. This value was assumed to be 0.9 for all caches. The hit ratio was not varied in the evaluation of the equations. It is intuitively obvious that as the hit ratio drops the amount of traffic into the network increases, and we were more concerned with the effect the other parameters had on the interconnection network traffic.

The average path length of a transaction was calculated for each type of network assuming an equal distribution of requests between the nodes. Average paths for the bus and a crossbar are always *one*. The mesh and multistage interconnection network, MIN,

are dependent on the number of nodes to be connected. In the case of the MIN it is $log_2(N)$ where N is the number of nodes connected by the MIN [36]. For the mesh it is $(\sqrt{N} - 1)$ [36]. In the hierarchical model defined previously, a different interconnection network may be used at each level, in which case, the average path length from the current level to the main memory is the sum of the average path lengths through each interconnection network type traversed in reaching main memory.

The number per link N_L was assumed to be *one* for the MIN, and crossbar. For the bus, the number per link is simply the number of elements interconnected by the bus. In the case of a mesh, each element is connected to *four* links, however, at the end of each link is another processor. Assuming each processor contributes an equal amount of transactions to each link it is connected, and there are two processors on each link, the N_L for a mesh was taken to be $\frac{1}{2}$. Notice that this does not consider edge nodes, and may generate some inaccuracy for the mesh.

4.1 RESULTS OF NBR EQUATION EVALUATION

The graphs of the *nbr* data maintain a consistent shape for interconnection networks of similar size while the *nbr* values are of course different. To eliminate redundancy, only a small sample of these graphs are shown. Figures 4.1, 4.2, 4.3, and 4.4 show the typical curves for S, P_s, P_r, and P_d for multiprocessors of size 2, 64, and 1024 respectively. As can be seen from these figures the influence each parameter has on interconnection network traffic relative to each other changes significantly as the multiprocessor size grows. The influence of each parameter can be measured by calculating the slope of each line in

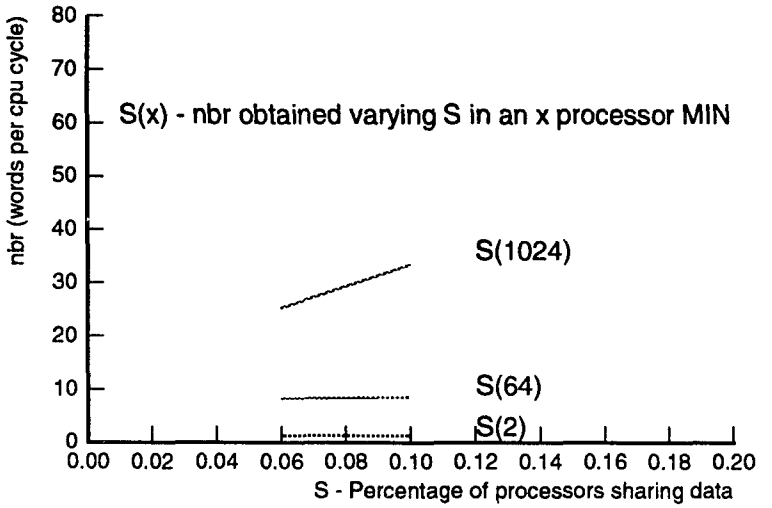


Figure 4.1: Typical curves for *nbr* vs. S for a MIN multiprocessing system.

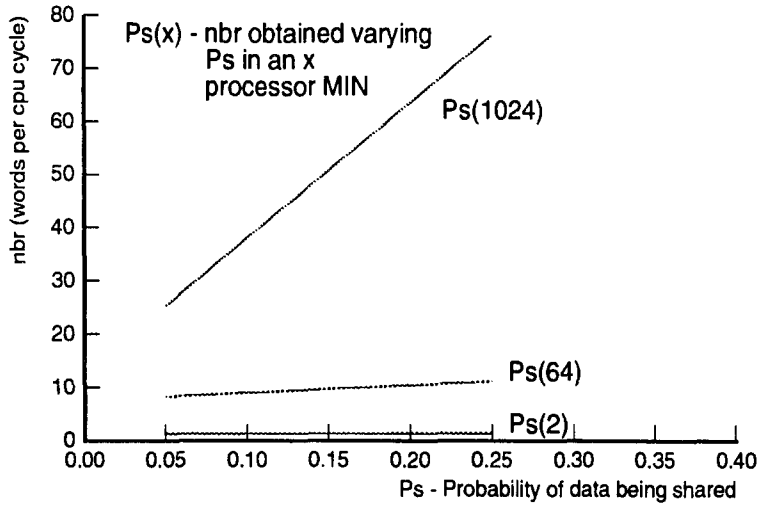


Figure 4.2: Typical curves for *nbr* vs. P_s for a MIN multiprocessing system.

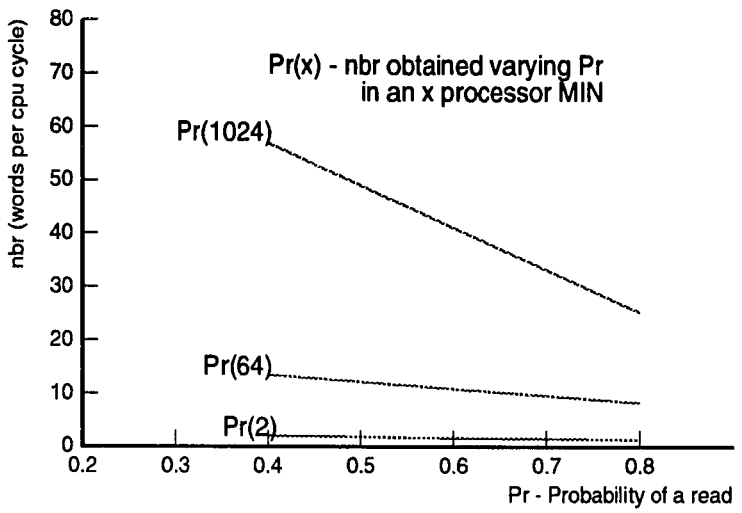


Figure 4.3: Typical curves for nbr vs. P_r for a MIN multiprocessing system.

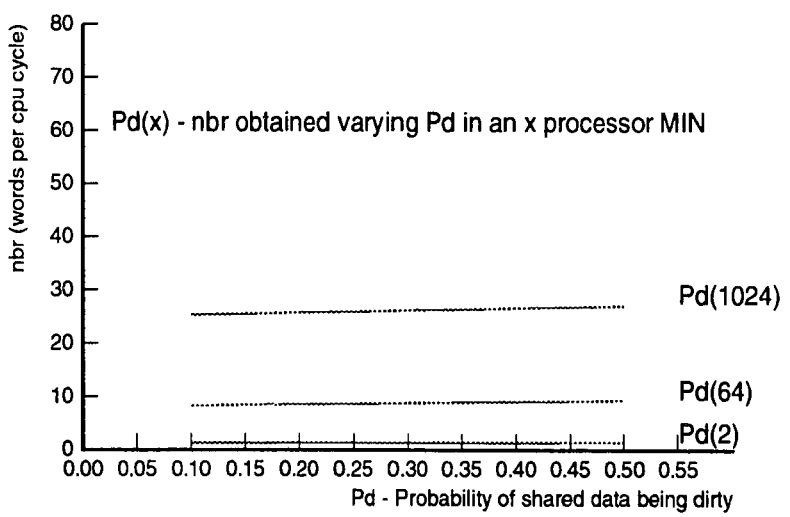


Figure 4.4: Typical curves for nbr vs. P_d for a MIN multiprocessing system.

the *nbr* versus probability graphs. For small multiprocessors, the parameter P_r exerts the greatest influence on interconnection network performance, however, as the number of processors grows, the effect of P_s, and S on the interconnection network performance also grow, eventually becoming the dominant parameters. Each parameter causes an increase in the *nbr* as the multiprocessor size increases. The slopes of the lines also increase with multiprocessor size. This observation is significant to a designer.

By graphing the slopes of these lines, termed influence, against the number of processors for each network type, the parameters with the largest impact on interconnection network traffic can be readily identified. These graphs are shown in Figures 4.5,

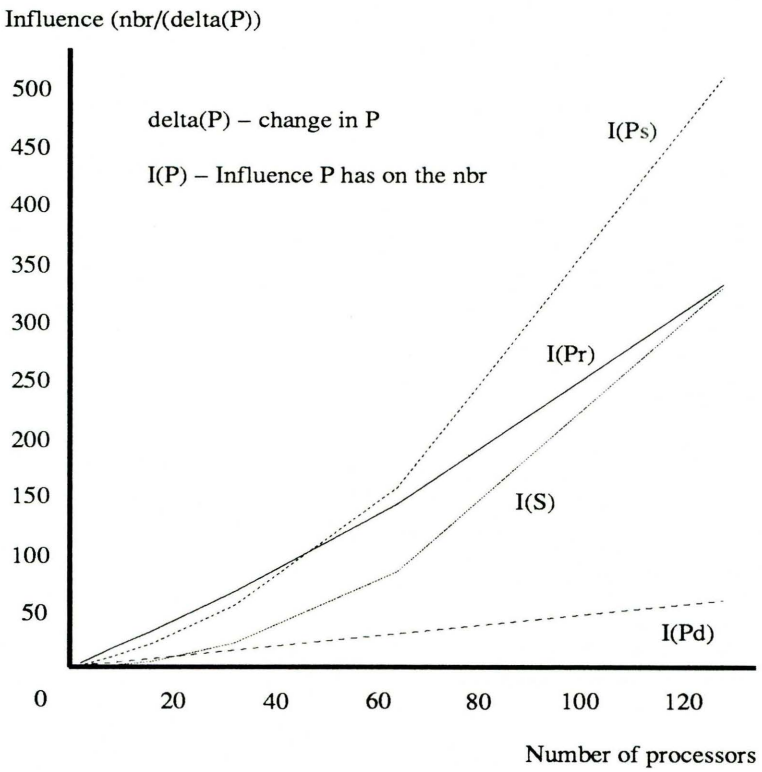


Figure 4.5: Influence of P_r, P_s, P_d, S, versus the number of processors for the bus multiprocessor

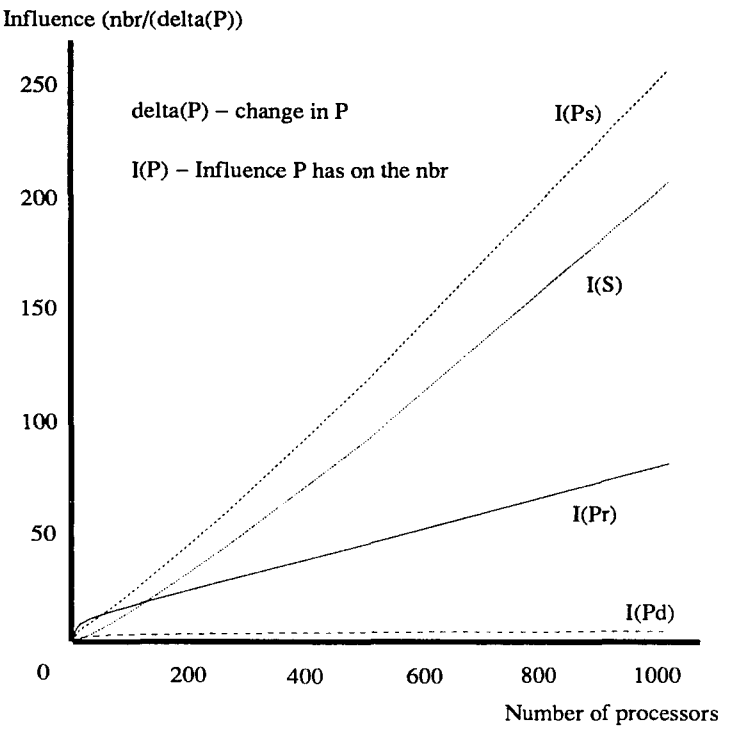


Figure 4.6: Influence of P_r, P_s, P_d, S, versus the number of processors for the MIN multiprocessor

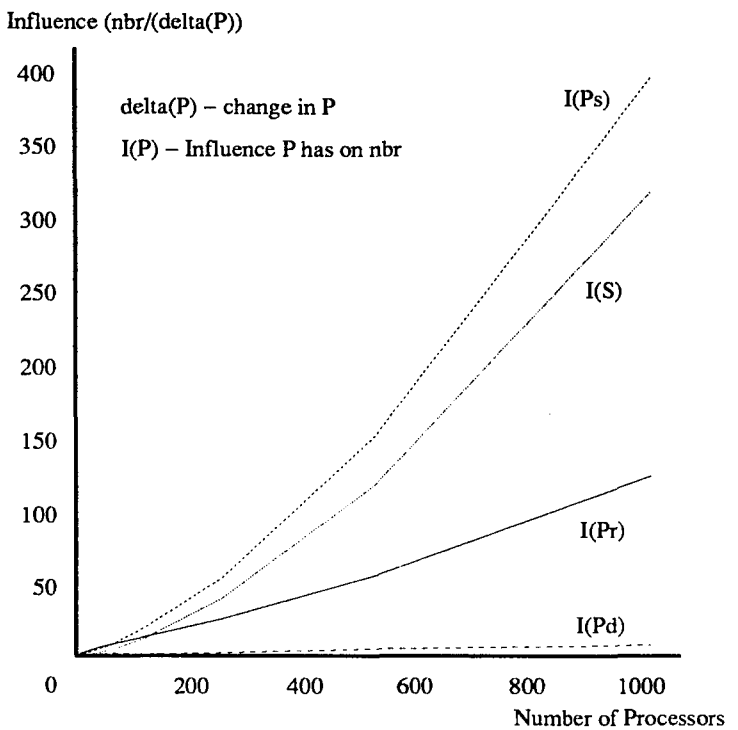


Figure 4.7: Influence of P_r, P_s, P_d, S, versus the number of processors for the mesh multiprocessor

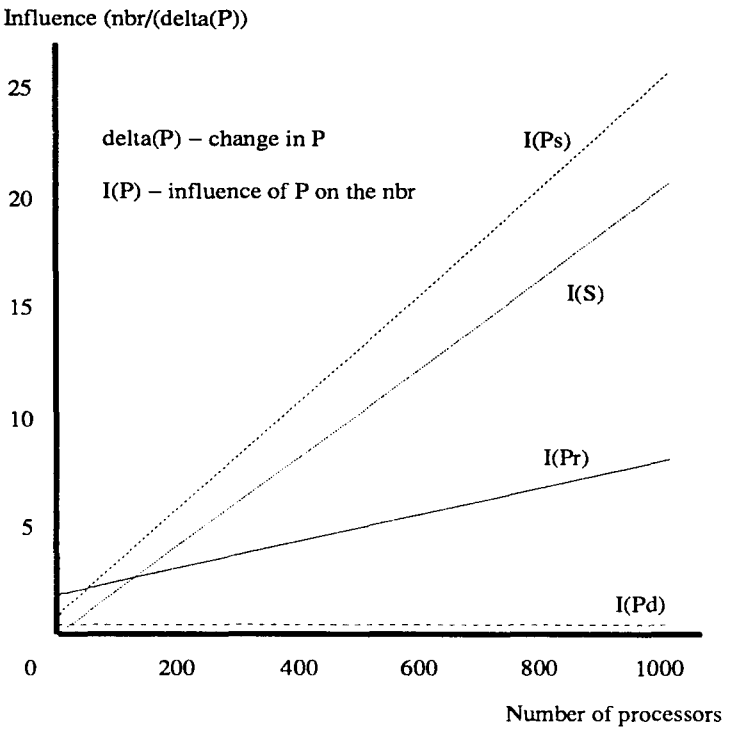


Figure 4.8: Influence of P_r, P_s, P_d, S, versus the number of processors for the crossbar multiprocessor

4.6, 4.7, and 4.8. The influence of P_d, denoted $I(P_d)$ on the network traffic remains essentially the same for each of the interconnection network types, and compared with the other parameters it has very little impact on network traffic. Therefore, a designer need not be overly concerned with controlling its impact on system performance since large fluctuations in P_d result in small increases in network traffic. The remaining parameters P_s, P_r, and S are issues that need to be addressed with some urgency. When designing a small scale multiprocessor, fluctuations in P_r have the greatest impact on network traffic, system performance, but its influence, denoted $I(P_r)$, increases at a much slower rate than that of influence of P_s, denoted $I(P_s)$, and the influence of S, denoted $I(S)$, as the multiprocessor size increases. Although the effect $I(P_r)$ has on system performance decreases with increasing size, it should be minimized for all multiprocessor sizes. Increasing multiprocessor system size magnifies the effect of $I(P_s)$ and $I(S)$ and they become the dominant factors requiring the most attention. Small fluctuations in P_s, S, and P_r result in substantial increases in network traffic, therefore, a designer must be concerned with controlling the influences of these parameters.

The *nbr* results for the initial values of the probability variables are graphed in Figures 4.9, and 4.10. In Figure 4.9 the results for both the mesh and crossbar are not shown. This was done to clarify the graph because their *nbr* results both fall below that of the MIN, which is also dwarfed by the *nbr* results of the bus multiprocessor. The conversion of the *nbr* results to an interconnecting link speed is very simple. The *nbr* represents the number of one word transactions each link must be capable of handling in a cpu cycle to sustain processor execution. Therefore, if the cpu has a 20MHz clock

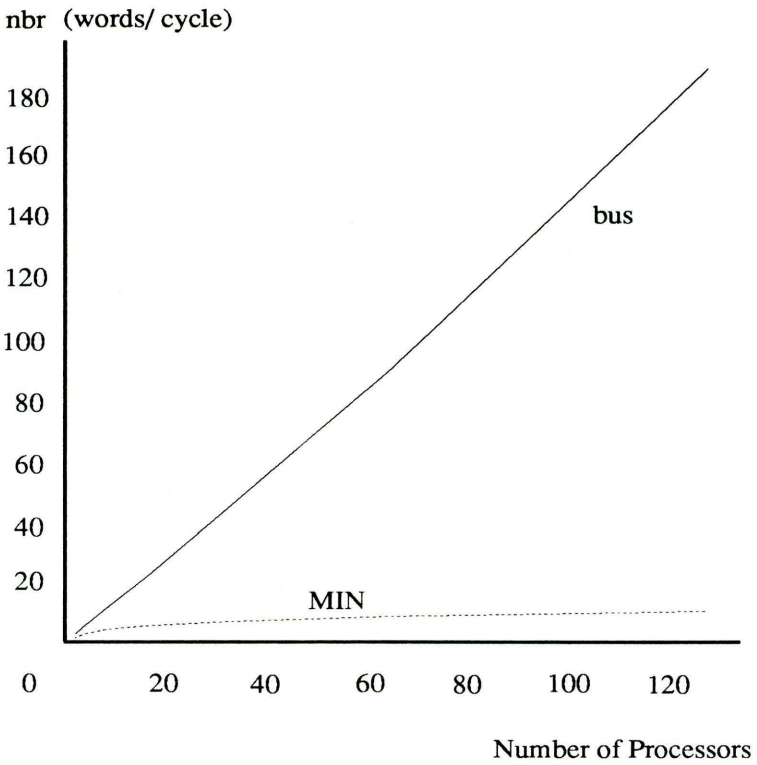


Figure 4.9: *nbr* vs. the number of processors.

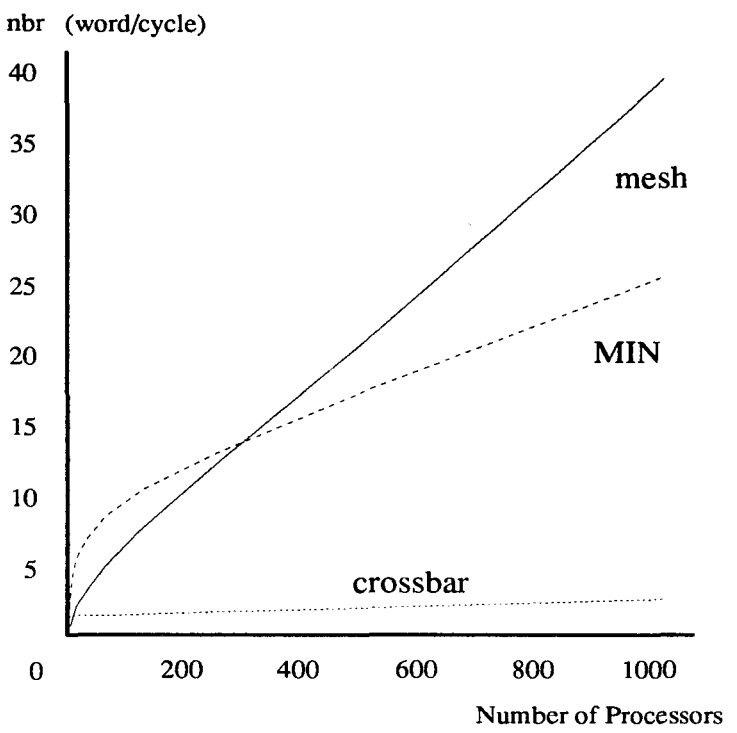


Figure 4.10: *nbr* vs. the number of processors.

cycle, the interconnecting links should have a cycle of $20 \times nbr$ MHz. Obviously, it is impossible to operate interconnecting links at some of the speeds indicated given current technological constraints. Recognizing this, certain configurations may be eliminated from further consideration and simulation can be used to further investigate the remaining candidate networks. The simulation is to help verify and refine (if necessary) the *nbr* values. It may also be used to investigate other aspects of multiprocessor performance, such as memory interleaving.

From analyzing Figures 4.9 and 4.10 a designer can make some observations of what the best interconnection network is for his design. These figures graph the *nbr* results against the number of processors. As can be seen, the bus is the least capable interconnection strategy, but it is still a feasible choice for very small numbers of processors. The mesh is a better performer than the MIN until about 300 processors, after which, the MIN is better. This is due to the average path length through the interconnection networks. The average path length of the MIN increases with $log_2(N)$ while the mesh increases with $\sqrt{N}$. This may not reflect real world cases since there exist methods of controlling the average path length of a mesh through locality of reference [29, 30]. The difference in performance between the mesh and MIN is not substantial. As expected the crossbar performs the best. However, performance is not the only factor to consider in choosing an interconnection network.

Cost is another major factor in deciding which interconnection network should be used. It is difficult to make a truly accurate cost model without getting very specific with system specifications. To develop a truly accurate cost formula many different variables must be

known, for example: the memory size must be known, and the fabrication process used to manufacture the VLSI circuits must be known. The analysis presented here is intended for use in the initial phases of multiprocessor design, hence, most of these variables may be unknown. However, the overall cost will be proportional to the number of processors. Table 4.1 shows the basic cost functions of each network type analyzed based on previous work [10, 3, 24]. Cost is assumed to be proportional to the switch and wire costs. For

Table 4.1: Basic cost functions of the various interconnection networks where n is the number of processors.

Network type	Cost function (in units)
bus	$C_{bus}(n) = O(n)$
Mesh	$C_{mesh}(n) = O(n)$
MIN	$C_{MIN}(n) = O(nlog_2 n)$
Crossbar	$C_{crossbar}(n) = O(n^2)$

the performance value, the inverse of the *nbr* will be used. The inverse of the *nbr* is used because it is the frequency at which an interconnection link operates in order to sustain the processor performance under ideal conditions. If the link speed is fixed, performance will degrade by an amount proportional to the *nbr*. Figures 4.11, and 4.12 are graphs of the cost/performance results. Remember. the lower the cost/performance ratio, the better. Please note that this simple cost/performance analysis is intended to heighten the reader's awareness that performance is not the only issue to consider in multiprocessor design, it is not intended to be a detailed cost/performance model. From the graphs, we can see that the mesh has the best cost performance ratio. The cost/performance ratio of the crossbar and MIN are similar for small numbers of processors, but as the number of processors increases, the MIN is much more economical. The bus interconnection has

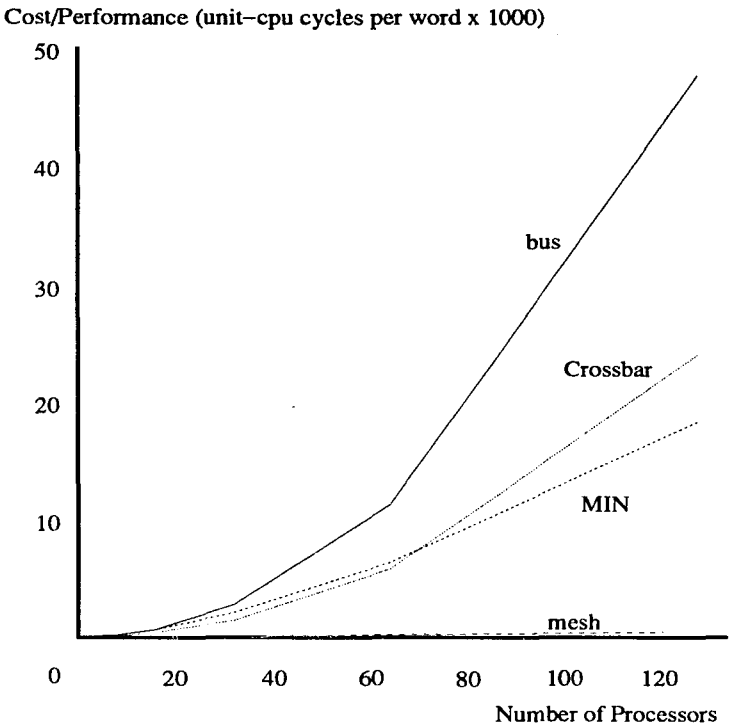


Figure 4.11: Cost-Performance ratio for each network type with small scale parallelism.

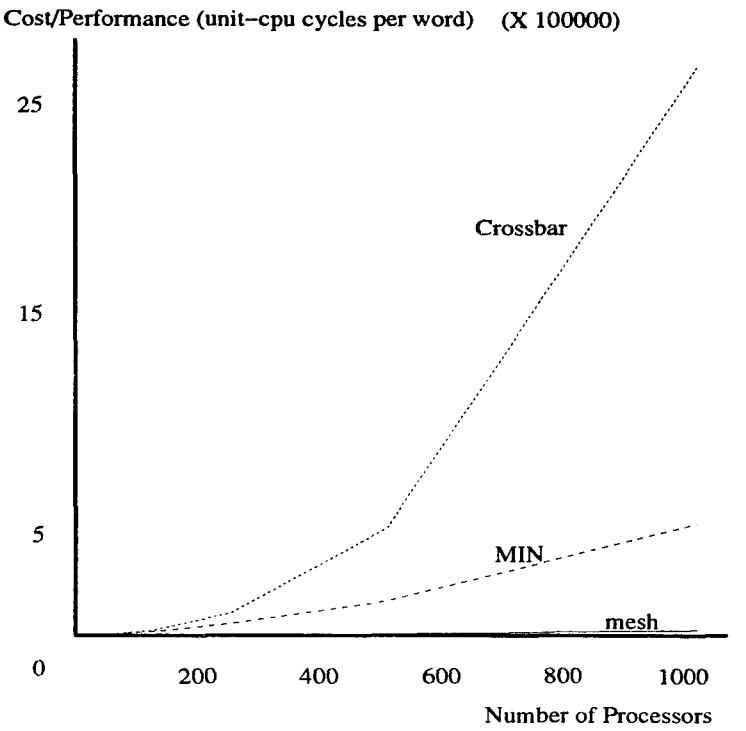


Figure 4.12: Cost-Performance ratio for each network type with medium and large scale parallelism.

the worst cost/performance ratio of the networks evaluated. It is important to realize the reason behind the cost/performance ratios for each network. The bus, while an economical interconnection strategy, has extremely poor performance characteristics for increasing numbers of processors, this causes a poor cost/performance ratio. For the crossbar, the performance is excellent, however, the cost is astronomical. The MIN has moderate cost and performance characteristics which is reflected in its cost/performance ratio. The mesh has excellent cost and performance characteristics, and therefore, has the best cost/performance ratio. An interconnection network choice must be made considering all possible factors. The cost advantage of any network over the crossbar is obvious, but the level of performance attainable by the crossbar is unrivaled. A designer must weigh the importance of each factor with the goals of the design to make the best decision.

CHAPTER 5

SAN MODEL OPERATION AND DESCRIPTION OF THE MULTIPROCESSOR MODELS

The purpose of the simulation is to explore the implications the *nbr* has on multiprocessor performance. With the SAN models, various multiprocessor characteristics may be explored and the effectiveness of the *nbr* value can be accessed. As a reminder only single level multiprocessors are simulated. In this stage adjustments may be made to the *nbr* value to provide the best speed estimate possible. The analysis provided by this simulation is steady state. All processors are assumed to be running in a steady state where each processor is generating a memory transaction every fourth instruction executed. Requests are also implicitly queued when contention for an interconnecting link arises. In this chapter the operation of SAN models in *UltraSAN* [11] is discussed. After an introduction to SAN operation, the operation of each interconnection network model is discussed. Finally, some inconsistencies inherent in the models are discussed. While these inconsistencies are present, their effect on the results is minimal.

5.1 SAN MODEL OPERATION

To verify the results of the analytical phase, the tool *UltraSAN*[11] was used. *UltraSAN* is a modeling and simulation tool based on stochastic activity networks, or SANs [11, 12, 13]. SANs were chosen for the simulation due to their probabilistic nature. To clarify the SAN models and their operation, a brief introduction to the definitions and rules governing SANs is presented. Figure 5.1 is a fairly simple SAN model which contains all of the

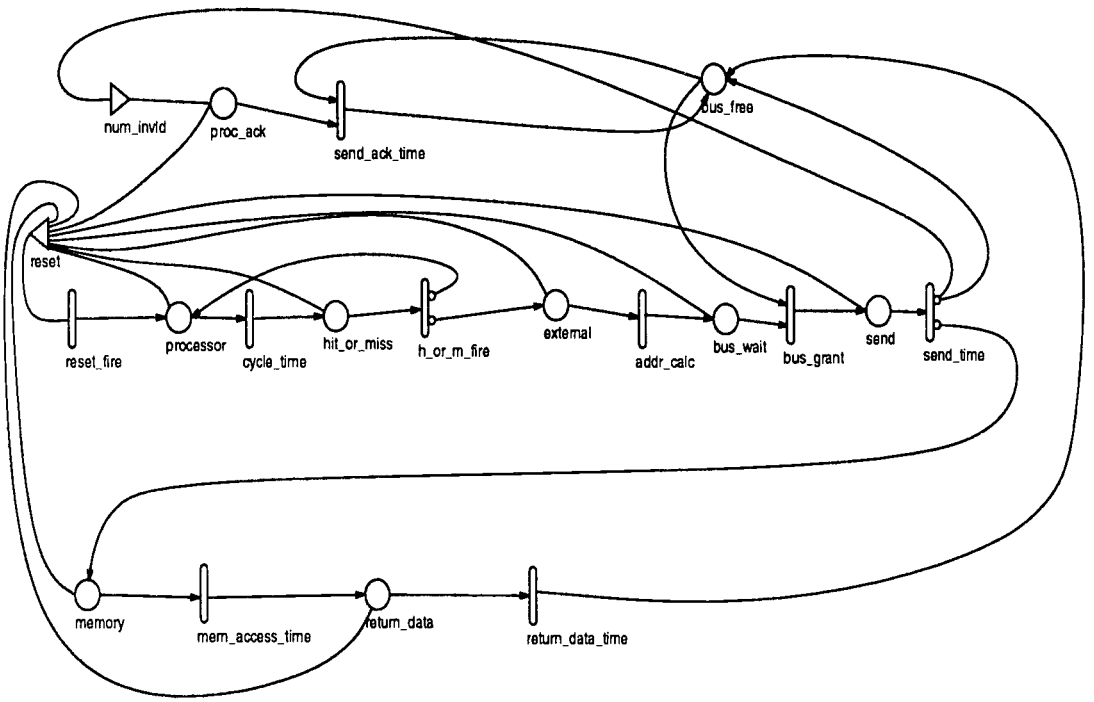


Figure 5.1: An example SAN model to demonstrate the functionality of the various components.

elements used in the models of this thesis. These elements are *places*, *timed activities*, *input gates*, and *output gates*.

Places are represented by circles in Figure 5.1. For example, *processor* is a place. Each place holds a certain number of tokens, which represents the state of the simulation. In this model, a *token* is just a representation of control. Places are connected to either *input gates*, *output gates*, or the input or output of *activities*. The marking of a place is just the number of tokens it contains. This is represented in *UltraSAN* by the function $MARK(place\_name)$.

Activities are the method by which control is transferred. This transfer of control is accomplished through the movement of tokens. *Activities* may be instantaneous or timed. This thesis uses no instantaneous activities, and so *activities* is synonymous with *timed activities*. Timed activities are depicted as ovals, and take a certain amount of time to complete. Completion of activities is discussed later. Activities can also be used to make decisions, through case probabilities. Activity *h_or_m_fire* is an example of an activity with cases. When the activity completes, one of the possible paths is chosen based on their probability values.

Input gates have two portions, a predicate, and a function, they are depicted graphically by a left pointing triangle. A predicate is a statement of the condition that needs to be satisfied for the gate to enable its activity. A function is simply a statement or sequence of statements that typically alter the marking of any place(s) connected to the gate. They are defined in C code with the MARK extension. The function of the input gate is executed when its activity completes. In Figure 5.1, *reset* is an input gate. Its predicate is:

$(MARK(proc\_ack) == 0)$ && $(MARK(processor) == 0)$ &&

$(MARK(hit\_or\_miss) == 0)$ && $(MARK(external) == 0)$ &&

$(MARK(bus\_wait) == 0)$ && $(MARK(send) == 0)$ &&

$(MARK(memory) == 0)$ && $(MARK(return\_data) == 0)$ &&

This means that when *bus_free* is the only place that contains a token, input gate *reset* enables activity *reset_fire*. The function for *reset* is identity, meaning no action.

Output gates consist only of functions, they are graphically depicted by right pointing triangles. The completion of an activity triggers the output gate function to execute. The function of output gate *num_invld* is MARK(*proc_ack*) = x, this places x tokens into place *proc_ack*. The number x is based on the number of processors in the system and S, the percentage of processors sharing data.

Activities are enabled when all input places directly connected to the activity contain at least one token, and the predicates of all connected input gates are true. Once enabled, an activity may "fire". The activity time may have many different distributions, and must be enabled for the entire time to fire. After firing, an activity is said to have completed. For example, in Figure 5.1, activity *cycle_time* is enabled when place *processor* contains at least *one* token. Activity *bus_grant* is enabled only when both *bus_wait* and *bus_free* contain at least *one* token each. Finally, the activity *reset_fire* is enabled when the predicate of the input gate named *reset* is true. The action of completion removes a token from each of the input places, or in the case of an input gate, executes its function, and then places a token in each of its output places, or in the case of an output gate, executes its function. If the activity has different cases, a specific case is chosen and then the token is passed, or function executed.

This is the basic functionality of the SAN elements, however, for model construction a composed model must be developed. The composed model for Figure 5.1 is shown in Figure 5.2. Composed models have three elements, of which only two are shown in

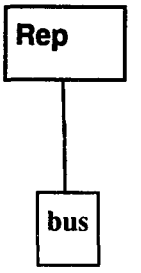


Figure 5.2: Example composed model for the bus multiprocessor. The figure illustrates the subnet *bus* being replicated.

Figure 5.2. The elements are *subnets*, *replications* (REP), and *joins* (JOIN). Figure 5.1 is an example of a subnet and is represented in Figure 5.2 as the bus node. Replications reproduce subnets, and can as in this example do so with places in common. Common places are shared by each replication. In the example, the subnet *bus* is replicated n times with the place *bus_free* in common. The number n is determined by the size of the multiprocessor being simulated. This causes the entire subnet with the exception of *bus_free* to be replicated, this one copy of *bus_free* is shared by all the replications. The final construct is the *join*. A join is a means by which you can connect different submodels together. Here too, common places may be specified, but they must have identical names in each subnet.

UltraSAN can be used to obtain both analytic and simulation results. Due to the complexity of the models, only simulation was done. The simulation in this thesis uses

deterministic activity times. With this brief introduction a description of the model functionality may be made.

5.2 MULTIPROCESSOR MODEL DESCRIPTIONS

Each model contains processing components, interconnection network components and memory components. Processor and memory models are based loosely on actual systems [37, 38, 39, 40]. Interconnection network models are also based on previous work [41, 42, 43, 44]. The processor and memory models are similar for each interconnection network simulated. The basic processor model consists of a processor place, a cache, and an address calculation for global memory. Rather than assuming a processor issues a memory transaction every cycle as in the analytical analysis, it is assumed the processor issues a memory transaction every fourth cycle, a more realistic value. Each memory module consists of a memory place, an activity denoting service time, and a mechanism through which data is returned to the requesting processor. For these simulations it is assumed that the memory is capable of retrieving one word per cpu cycle. Therefore, the service time for a memory transaction is the cache line size L times $1 word/cycle$, in this case, $four$. No data is returned until the entire line is retrieved from memory. While this assumption is not optimal, it greatly simplifies the model. The degree of memory interleaving is equal to the number of processors. A two processor multiprocessor has 2-way interleaved memory, and so on. Another similarity between all the models is that no transactions are lost or discarded. If a resource is contended for, the first request to arrive is serviced, and the rest are queued. Each queued element is serviced in time. Requests were queued to reduce the model complexity. As mentioned above the model complexity is a very important issue due to the limits of our computing resources. Overly complex models,

or models of extreme size require enormous amounts of computing resources and time to run.

5.2.1 BUS MODEL DESCRIPTION

The multiprocessor bus structure is shown in Figure 5.3, and its SAN model representation is shown in Figure 5.4. In the model's initial state, places *processor* and

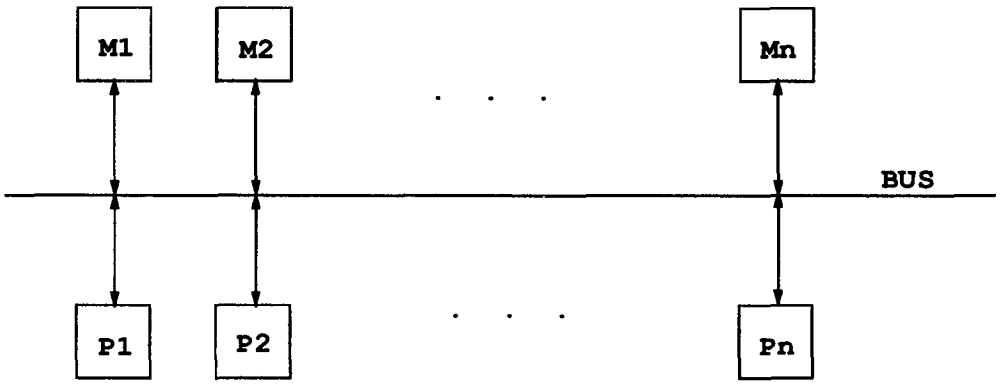


Figure 5.3: Structure assumed for bus multiprocessors.

bus_free contain a single token each, all other places are empty. The token in *processor* enables the activity *cycle_time*. On the fourth cpu cycle, *cycle_time* "fires", removing the token from the processor and placing it in the place labeled *hit_or_miss*. Activity *h_or_m_fire* is now enabled. After a short delay, time to determine if the requested item is contained in the cache, *h_or_m_fire* completes. When this occurs one of two paths are chosen based on the probability of a *miss*. A *miss* occurs with probability

$$M = (1 - P_r) + (1 - h_i) + h_i \times P_s \times P_d$$

For this particular case, a *miss* refers to the need for an external (to processor cache pair) transaction. This occurs when a write transaction is initiated requiring invalidations, a

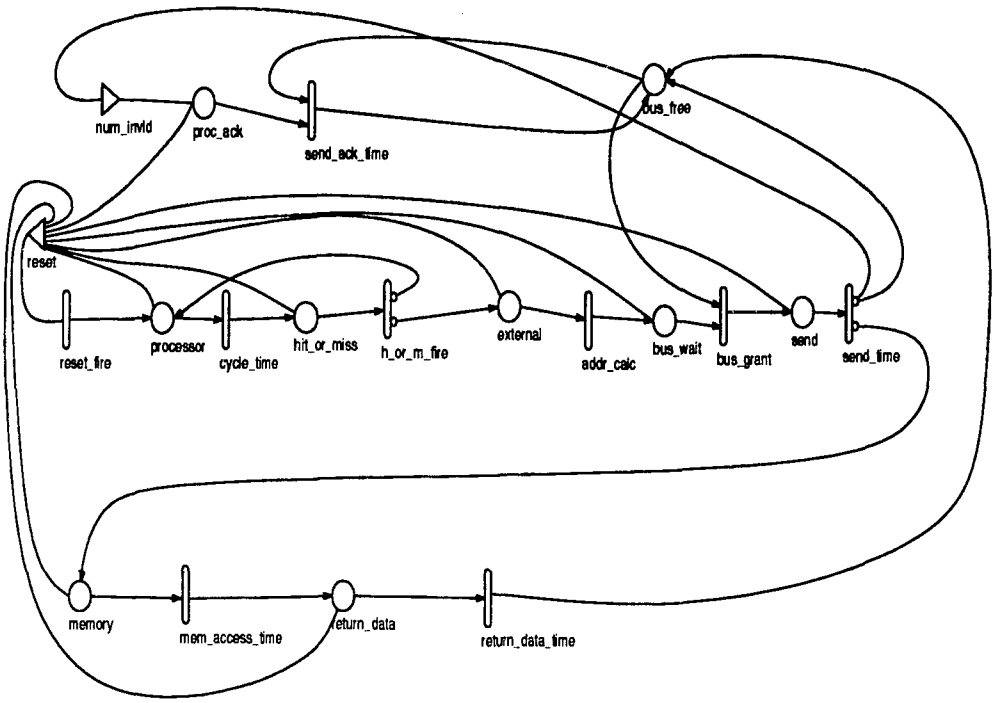


Figure 5.4: SAN sub-model for the bus multiprocessor.

cache miss, or on a hit to dirty shared data. A *hit* is simply $1 - M$. A *hit* reinables the processor by replacing the token. On a *miss* the token is placed into *external*, which enables activity *addr_calc*. *Addr_calc* represents any delay necessary to prepare a request for transmission. On completion *addr_calc* places a token into *bus_wait*. At ths juncture, the activity *bus_grant* requires both *bus_wait* and *bus_free* to contain a token. The place *bus_free* initially contains a single token, which represents the bus being idle. When the bus is free, and a processor is requesting the bus, *bus_grant* is enabled. The time required for *bus_grant* to complete is based on the *nbr* results. After completing, a token is placed into *send* enabling *send_time*. *Send_time* represents the time to transmit the transaction to its destination and is based on the *nbr* results. As a consequence of this action, the

token in *bus_free* has been removed, and the bus is unavailable for use by any other processor. Activity *send_time* is enabled, and upon completion one of the possible paths is chosen based on P_r. When a write transaction occurs invalidations are sent. When this path is taken, a certain number of tokens ($S \times N$) representing the number of invalidations necessary is put into *proc_ack*. The bus must be free for these invalidations to be sent. The time for *send_ack_time* to complete is based on the *nbr*. Two things should be noted here about the invalidation processing. First, this is not a very efficient means of implementation. A snoopy protocol [25, 26, 27] would require less traffic in a bus multiprocessor than the distributed protocol implemented since the invalidations are put on the bus only once as opposed to ($S \times N$) times. The distributed protocol was kept to maintain consistency with the *nbr* calculation. It is also assumed that no arbitration is needed to obtain the bus for these transactions. If the request had been designated as a read, a token would be placed in *memory*. It is important to notice that the bus is released (token put in *bus_free*) after *send_time* completes. This allows for interleaving of bus transactions and should increase the performance obtained. The request has now reached memory, and *mem_access_time* is enabled. After fetching the cache line containing the requested data a token is placed in *return_data*. Again, the time for *return_data* to complete is based on the *nbr* results. Memory is also able to accept new requests. After obtaining the bus the data is returned to the processor. This operation is not done directly though. First a token is placed in *bus_free* to release the bus. At this time, all places excluding *bus_free* will contain no tokens. When this situation arises activity *reset_fire* is enabled. Upon completion, the processor is reinabled and the process begins again.

Figure 5.4 only represents one processor with a bus and memory. In order to represent a multiprocessor, it is necessary to *replicate* this model. Figure 5.5 shows the composed model for the bus multiprocessor. The replication is done with the place *bus_free*

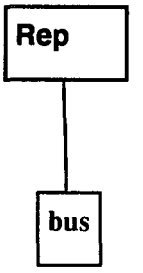


Figure 5.5: SAN sub-model for the bus multiprocessor.

common to each of the replicated models. This shares the bus. It is important to note that this model strictly enforces a uniform distribution of requests. Therefore, while bus contention is represented, contention for memory modules is not.

5.2.2 CROSSBAR MODEL DESCRIPTION

The crossbar multiprocessor structure is shown in Figure 5.6. Unlike the SAN representation for the bus multiprocessor, the crossbar model contains three parts. The processor is shown in Figure 5.7, the memory in Figure 5.8, and the crossbar switch in Figure 5.9. The processor portion functions in the same fashion as it did in the bus multiprocessor model, however, there are some minor differences. The transaction type is not determined in the processor model. This was done to help simplify the construction of the crossbar switch. The place *track* in the processor is a mechanism to keep the processor from being reset before the request completes. The processor request completes when a token representing the cache line arrives in either place $R1$ or place $Invr1$. It

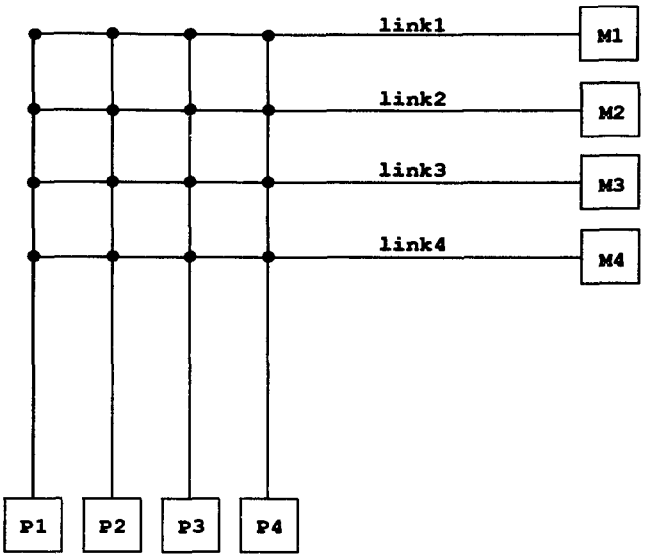


Figure 5.6: An actual 4 x 4 crossbar interconnection network.

should be noted that the processor performs "optimistic" processing. This means that when a transaction is determined to be an invalidation, the processor is reset, it does not need to wait until all invalidations are sent and acknowledgments received.

The three submodels shown in Figures 5.7, 5.8, and 5.9 must be connected in some fashion. The connection scheme is depicted in Figure 5.10. Places with the same names are joined together so that information may be passed between the submodels. This construction strictly enforces uniform distribution of requests as before, and limits the memory interleaving to be equal to the number of processors. With this construction, all traffic generated by one specific processor arrives at one and only one specific memory. This inconsistency will be discussed in a later section of this chapter. An alternative connection shown in Figure 5.11 can be used to explore the effects of reducing (or increasing) the degree of memory interleaving. We can now discuss the functionality of the rest of the model.

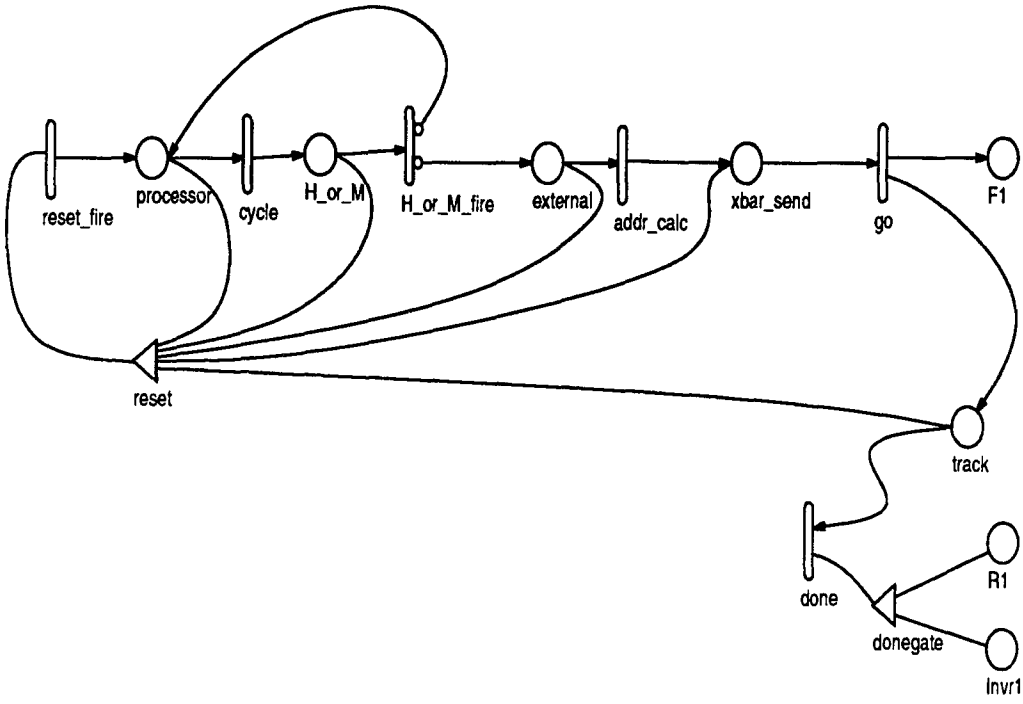


Figure 5.7: SAN sub-model representing the processor used in the crossbar interconnection network.

When a processor has generated a request that must be serviced externally to the processor-cache pair, a token is in place $F1$. Since this place is common to both the processor and switch models the activity $F1in$ is enabled, provided *limit* contains a token. Initially *limit* does contain *one* token. This limits the access to the link to one transaction at a time, thereby modeling the link contention. After $F1in$ fires, a token is placed in $F1q$ which enables *Fsend*. This represents the time required to transmit the request across the crossbar link and it is based on the *nbr* values. After *Fsend* completes the request it is forwarded to memory place $M1$.

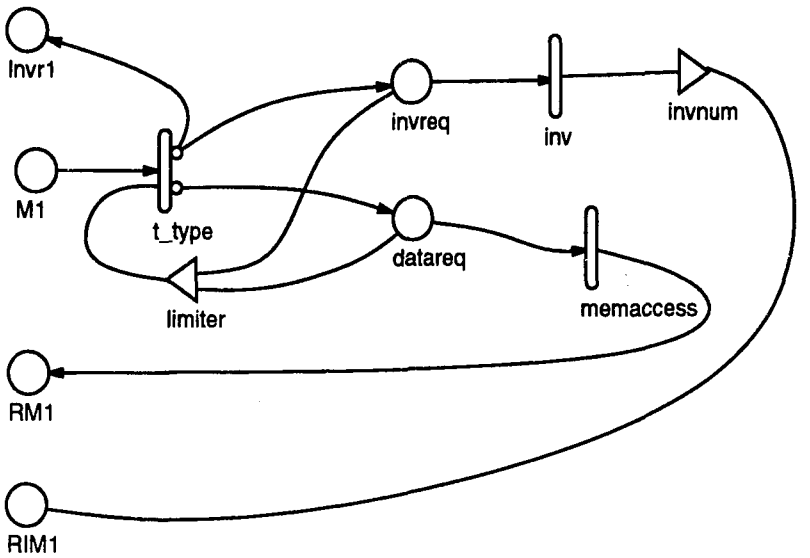


Figure 5.8: SAN sub-model representing the memory for the crossbar multiprocessor.

At this juncture there is a choice of paths. Again, invalidations are determined by the probability of a write transaction. If the transaction is determined to be an invalidation by the activity *t_type* a token is placed in both *invreq* and $Invr1$. The token in $Invr1$ allows the processor to be reinabled as described earlier. The token in *invreq* enables activity *inv*. After completing the output gate *invnum* places the number of transactions needed to represent invalidations and their acknowledgments into place $RIM1$. The input gate named *limiter* restricts access to the memory to one request at a time by disabling *t_type* if either *invreq* or *datareq* contains a token. Data requests are passed to *datareq* which enables activity *memaccess*. Upon completion of *memaccess* a token is placed into $RM1$. Both places $RM1$ and $RIM1$ are common to the memory and the switch.

Once a token arrives at $RM1$ or $RIM1$ the link must be free, *limit* has a token, to be accepted for transmission. On acceptance, they are sent across the link. The activities

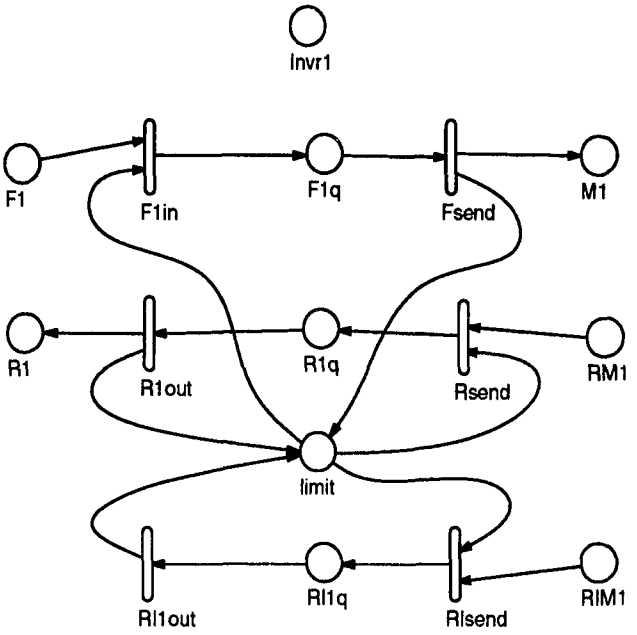


Figure 5.9: SAN sub-model representing the switch used in the crossbar interconnection network.

Rsend and *Rlsend* represent transmission time and are based on the *nbr* results. The data being returned is placed into $R1$ which allows the processor to be reinabled thereby restarting the process.

5.2.3 MIN MODEL DEVELOPMENT

The MIN multiprocessor structure for four processors is shown in Figure 5.12, and the SAN representation for the MIN is shown in figures 5.13, 5.14, 5.15. As in the crossbar representation, the MIN uses multiple parts, processor, memory and switching elements. The switching element is essentially a 2 by 2 crossbar. The processor used in the MIN model is identical in operation to the one used in the crossbar and requires no further discussion. However, the switching element is quite different. The state of the switch

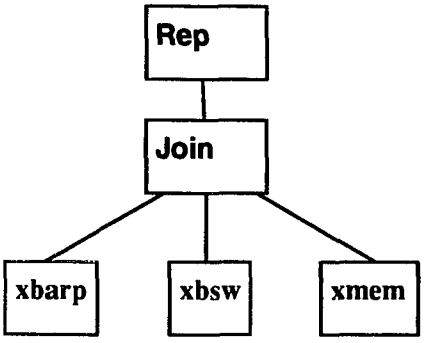


Figure 5.10: Composed model depicting the construction of the crossbar multiprocessor.

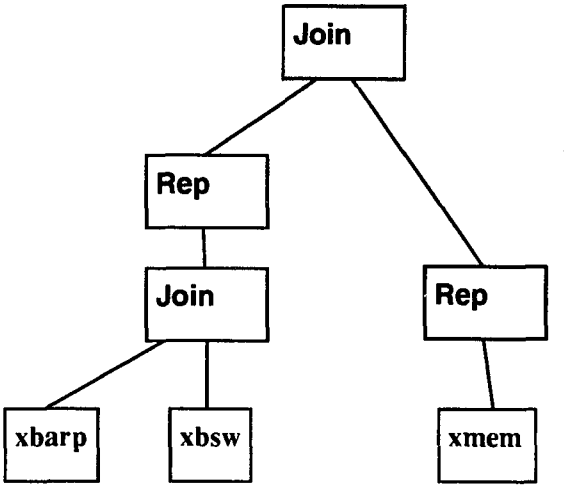


Figure 5.11: Alternative composed model for the crossbar multiprocessor that allows experimentation with memory interleaving.

(crossed or straight) is saved so that contention can be simulated. The MIN represented is a generic interconnection that requires $log_2 N$ stages, where N is the number of processors. It does not represent any specific type like shuffle exchange or omega interconnection schemes. Each specific MIN implementation has its benefits and drawbacks, and the selection of a specific multistage interconnection strategy is left up to the designer. The analysis presented is applicable to any MIN strategy which employs $log_2 N$ stages. The

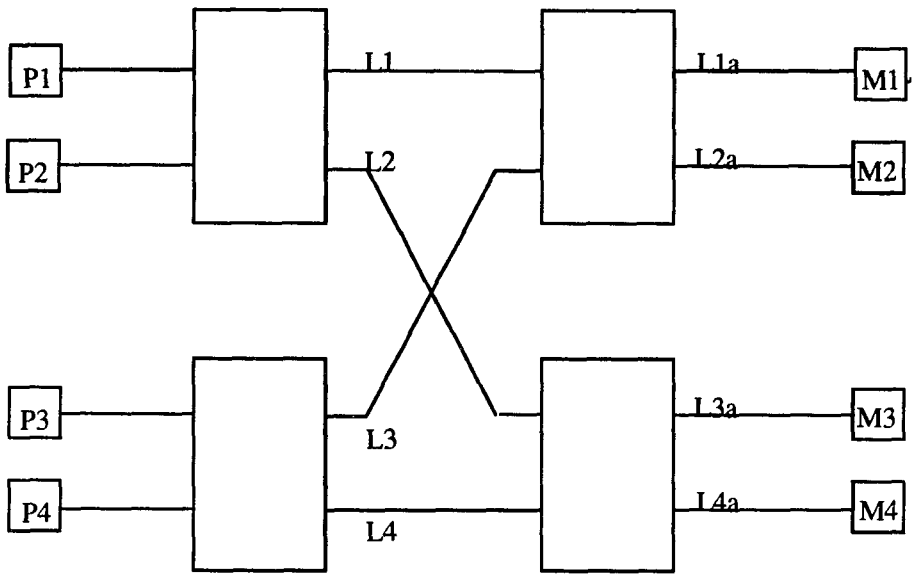


Figure 5.12: An actual 4 x 4 multistage interconnection network.

memory used in the MIN representation is also identical to that which is used in the crossbar.

The switching element used for the MIN representation is somewhat more complex than the crossbar model's switch. This is to better represent contention. The model shown in Figure 5.15 represents one path through the 2 by 2 switch. The entire switch is obtained through replication. This can be seen in the composed model shown in Figure 5.16. The three components, processor, switching elements, and memory are connected together by joining places with the same names. The places for the interface between the processor and switch are *din*, *dback*, *invback*, and *invrelay*. Between the switch and memory are $N1$, RIN, RN, and *invr*. For the connections between the switching elements the same places are used, however, they have different names. As mentioned the switch is fabricated through replication, this is the next step for the model construction. The

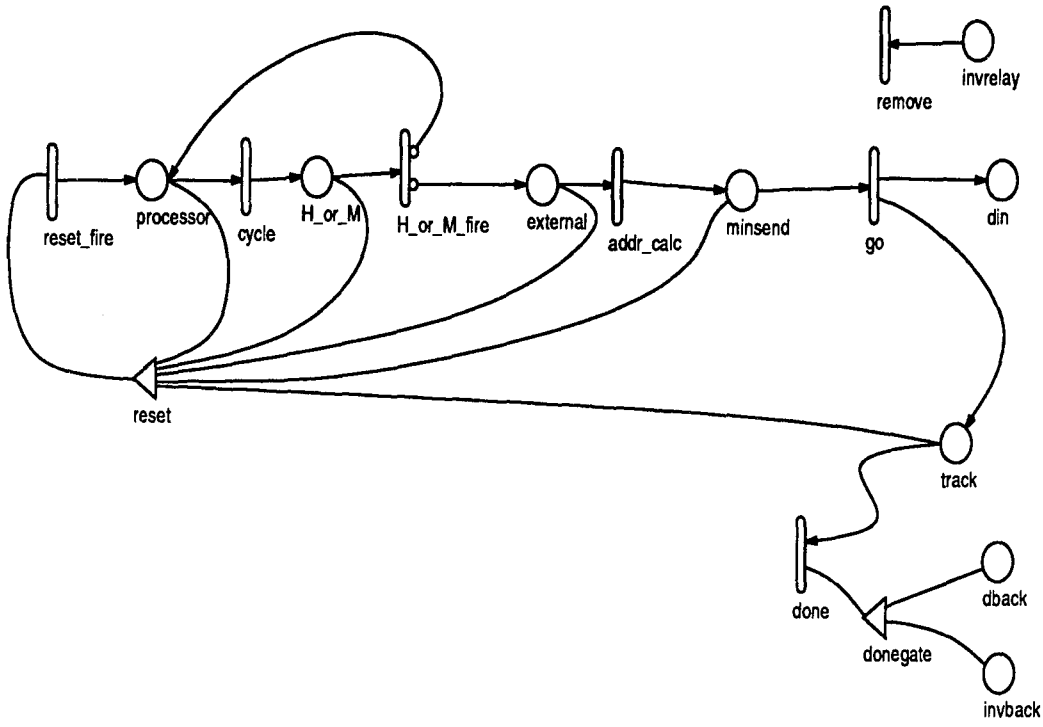


Figure 5.13: SAN sub-model for the processors used in the MIN multiprocessor.

replication box above the join in Figure 5.16 constructs all the switches by replicating the joined model twice with the places corresponding to *straight*, *cross*, and *limit* in common. These places have different names in each stage's switching element. This subsystem is than replicated $n/2$ times to obtain the full MIN model. This leads to the same inconsistency described in the crossbar, except that each processor will send their transactions to two different memory modules as opposed to one specific module. This inconsistency is discussed in a later section of this chapter. Specific functionality of the MIN switch is discussed now.

Requests enter the switch via *din* enabling activity *switch*. *Switch* completes and decides if the request requires the switch to be set straight or crossed. The probability

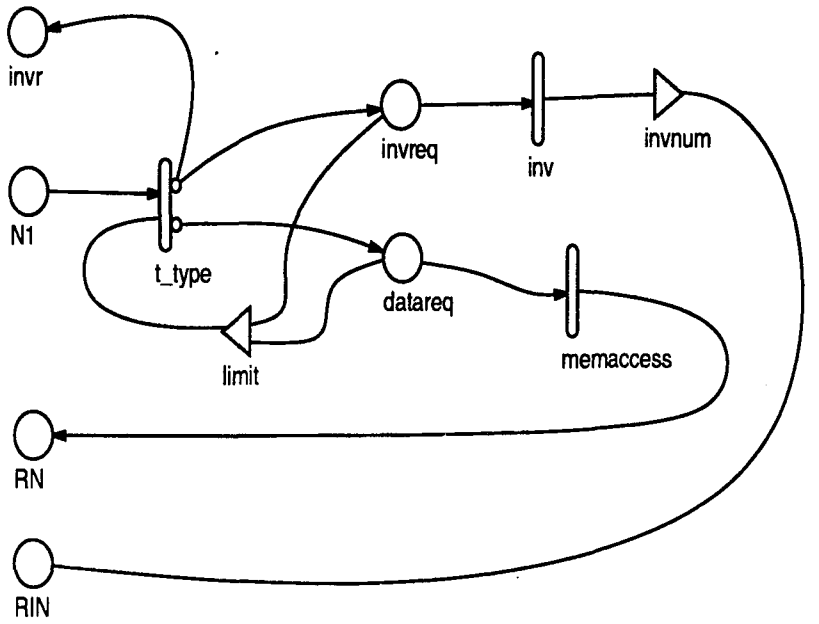


Figure 5.14: SAN sub-model for the memory used in the MIN multiprocessor.

of going straight or cross is 1/2. For the case of going straight, a token is placed in *S*. This is essentially a queue for this switching element. As mentioned earlier no requests are discarded, they are queued when necessary. The activity *Sok* is enabled when limit contains at least *one* token, initially *two*, a request is trying to go straight through the switch, and the switch is not in the *crossed* state. The *crossed* state is represented by a single token in *cross*. The input gate *sokin* trys to enable *Sok* when there are no tokens in *cross*. When *Sok* is enabled and completes, the MARK of *straight* is set to be 1, and a token is placed in *queue*. The same processing occurs for requests attempting to "cross" the switch. The token in *queue* enables activity *send* which completes after a delay based on the *nbr* results. Notice that the switch is capable of only passing one other request.

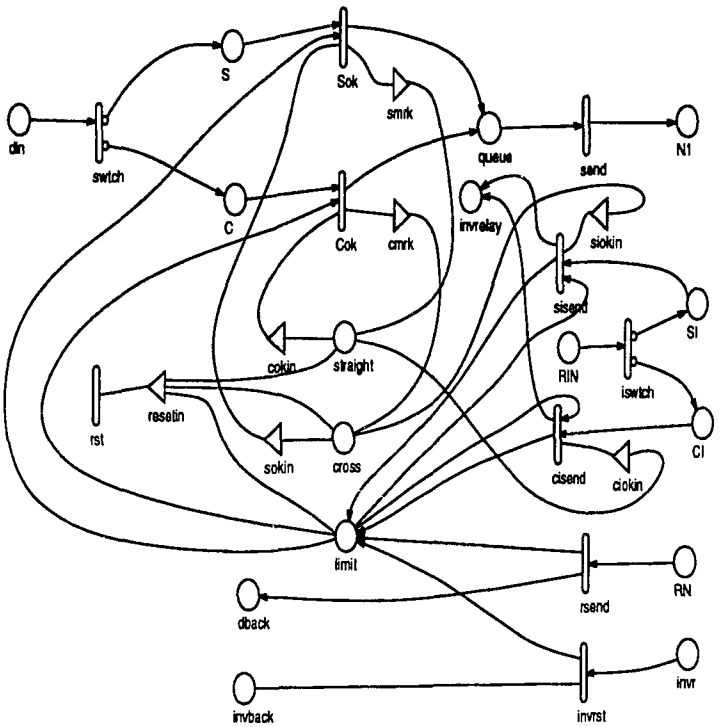


Figure 5.15: SAN sub-model for the MIN switching element.

This is because a token was removed from *limit* and was never replaced. This is to keep the switch from resetting. The token is replaced when the reply is received from memory.

Three transactions can return from memory, a signal that allows the processor and all the switching elements to reset, *invr*, the line of data, and the invalidations and acknowledgements. The line of data returns through RN. A token in RN represents data needing to pass through the switch, and enables activity *rsend*. Upon completion *rsend* replaces one token in *limit*, and passes another to the processor, or the next switching element as the case may be. When an invalidation or acknowledgement arrives, token in RIN, activity *iswtch* is enabled. Activity *iswtch* decides the path the message is required to take. The path, straight or cross is again decided with probability 1/2. The activities

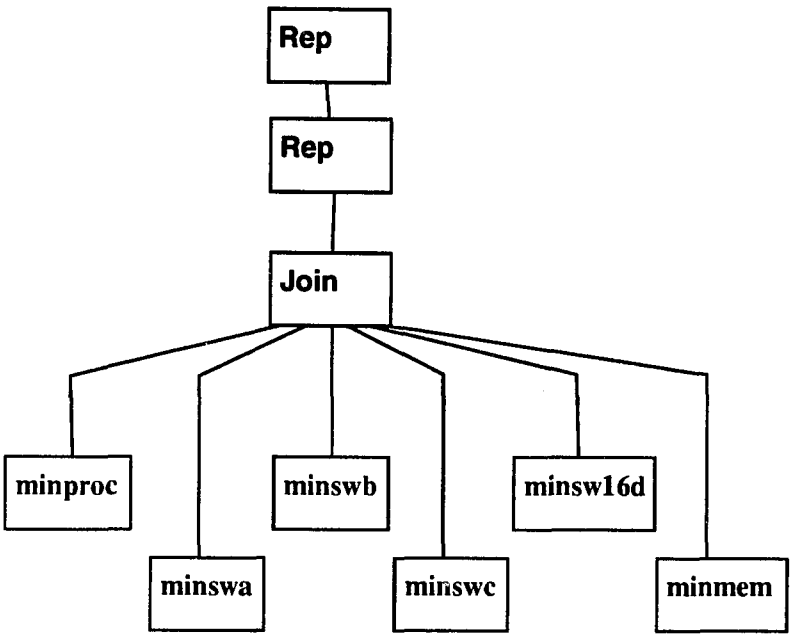


Figure 5.16: Composed model for a 16 processor multistage interconnection network.

sisend and *cisend* are enabled in the some fashion as were *Sok* and *Cok*. Completion places a token into *limit* and *invrelay*. Place *invrelay* is the interface to the next element in the path to the processor. The switch will remain in the crossed or straight state until the MARK of *limit* is 2. When this occurs the switch is unused, and therefore has no state. Eventually all transactions initiated by a processor will be satisfied, and it will be reinabled to begin the cycle again.

5.2.4 MESH MODEL DEVELOPMENT

The mesh multiprocessor structure for four processors is shown in Figure 5.17. Other than the processor model the mesh is very different than the other models. Figures 5.18 and 5.19 show the model representation of the processor, and mesh node respectively.

As indicated the processor model is the same as that in the crossbar and will not

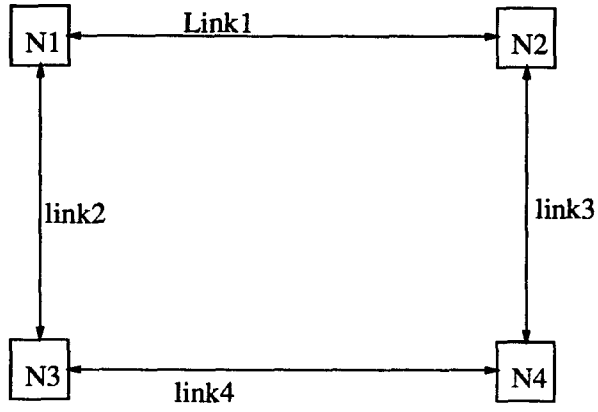


Figure 5.17: Simplified version of the mesh SAN model representation.

be discussed further. The mesh model implements a regular mesh with no wrap around connections. Therefore, edge nodes are different from internal nodes. A node is comprised of a processor, memory, and routing logic. The composed model for the 2 x 2 mesh is shown in Figure 5.20. A sample node used in the construction is shown in Figure 5.21. Requests from the processor enter the node through place *preq*. A restriction of *one* request being serviced is enforced for the routing logic through place *nodelimit* which has an initial *marking* of *one*. Activity *ttype* is enabled when $MARK(preq)$ and $MARK(nodelimit)$ is *one*. The firing of *ttype* determines the transaction type, either invalidation, or data request. The decision is based on the probability of a read instruction. If the transaction is determined to be an invalidation, output gate *inum* places a certain number of tokens corresponding to the number of invalidations and acknowledgements into place *ireq* enabling *ipath*. Activity *ipath* determines which neighboring node is the destination and places a token into the proper send queue, *dir* or *rir*. Both of these paths are symmetric with regards to actions taken, however, the destination node is different. The path selection is made with equal probability. That is to say for this example a token

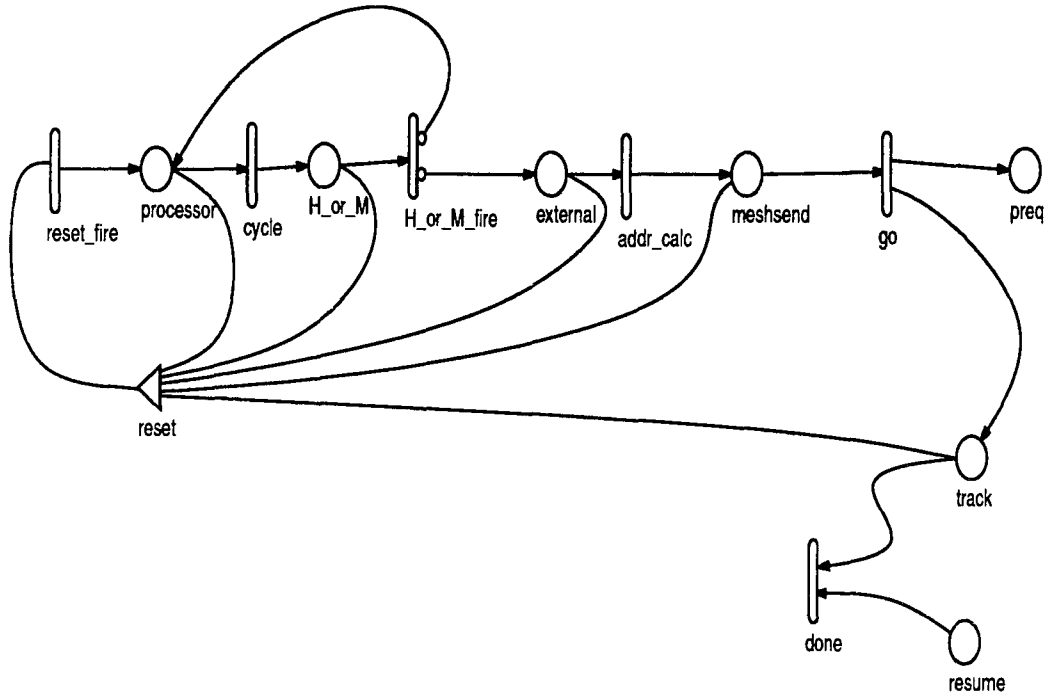


Figure 5.18: SAN sub-model representing the processor used in the mesh interconnection network.

is placed in *dir* with probability 1/2, with the same being true for *rir*. For nodes with *three*, or *four* neighbors, the choice is made with probability 1/3, or 1/4 respectively. Activity *isd*, or *isr* is then enabled when the required link becomes available, denoted by a token in *lim*13, or *lim*34. These places initially contain *one* token each. The time required for completion of activities *isd* and *isr* are based on the *nbr* values. After completion these activities place a token into the input place of their neighboring node.

If the token in *preq* had been designated a data request a token would be placed into *dreq*. As in the invalidation messages a choice of paths is made, and the request is sent

to a neighboring node. As you can see, it is assumed that the data is not contained in the requesting processor's node as was the assumption in the analytical analysis.

These data and invalidation/acknowledgment messages enter a node through places corresponding to $dr3$, and $ir3$. A decision is then made, by *reqpath*, or *irpath* as to whether or not this node is the destination node. For invalidation/acknowledgements, the tokens are consumed if it is the destination, if not they are passed to a neighboring node via the same mechanism as before. For data requests, if it is the destination node the request is sent to memory which after satisfying the request sends the data to one of the neighboring nodes, otherwise the request itself is forwarded.

Data returned in this fashion arrives in places corresponding to $dback3$. The mechanism for determining whether or not the data has reached its destination is a little more complex than before. First a token is placed in either *dest* or *memdone* with a probability derived from the mesh size. This will be derived shortly. If the token was placed into *dest* it is checked to see if the processor is waiting for data to be returned. If it is, then the processor is reset to resume processing. If it is not waiting for data, the token is passed to a neighboring node via *memdone*. All transmission times are based on the *nbr* values previously determined.

To determine the probability that a transaction has reached its destination is no trivial calculation. For the case of a 2×2 mesh shown in Figure 5.17 the probability that a token arriving at $N1$ has reached its destination is determined as follows. To simplify the calculation assume that all traffic is towards $N1$, traffic generated or forwarded by $N1$ is not considered. Consider all cases of data reaching $N1$. Define $\mid link1 \mid$ as the amount

of traffic on $link1$, $\mid N1 \mid$ as the amount of traffic generated by $N1$, and $\mid \overline{N1} \mid$ as the amount of traffic entering $N1$. P_f is the probability that a request is forwarded.

$\mid link4 \mid = 1/2 \mid N4 \mid$

$\mid link3 \mid = 1/2 \mid N4 \mid$

$\mid link1 \mid = 1/2 \mid N4 \mid \times P_f + \mid N2 \mid$ since all data travels to $N1$.

$\mid link2 \mid = 1/2 \mid N4 \mid \times P_f + \mid N3 \mid$

This results in:

$\mid \overline{N1} \mid = \mid N2 \mid + \mid N3 \mid + \mid N4 \mid \times P_f$

One third of these transactions are destined for $N1$, therefore, the probability that a transaction has reached its destination, P_{dest}, is

$P_{dest} = 1/3(2 + P_f)$.

Where $P_{dest} \equiv 1 - P_f$. Solving for P_f, we obtained $P_f = .25$, and so $P_d = .75$. In the 3×3 mesh,

$P_{dest} = 1/8(2 + 3 \times P_f + 2 \times P_f^2 + P_f^3)$

Since

$\mid \overline{N1} \mid = \mid N4 \mid + \mid N2 \mid + P_f \times \mid N7 \mid + \mid N5 \mid + \mid N3 \mid + P_f^2 \times \mid N8 \mid + \mid N6 \mid + P_f^3 \times \mid N9 \mid$

In general,

$P_{dest} = \frac{1}{N-1} \times (2 + 3P_f + 4P_f + \ldots + \sqrt{N} \times P_f^{\sqrt{N}-2} + (\sqrt{N} - 1) \times P_f^{\sqrt{N}-1} + (\sqrt{N} - 2) \times P_f^{\sqrt{N}} + \ldots + P_f^{(\sqrt{N}-1)\times 2-1})$.

N is the number of processors in the system.

The construction of the model shown in 5.20 is not very efficient. Due to size constraints of the system being used model construction becomes intractable even for small models

since $2 \times N$ submodels are required for a mesh with N nodes. For this reason simplifications became necessary. These simplifications are similar to those made for the MIN and crossbar and are presented in the following section.

5.3 MODEL INCONSISTENCIES

As mentioned in the crossbar, MIN and mesh sections, there are inconsistencies in the model representations. The simulation model used for the crossbar and MIN is represented in Figure 5.22. The boxes $P1$ to $P4$ represents the processors in either a 4×4 crossbar or MIN, $M1$ to $M4$ represent the memories, and the boxes labeled D are arbitrary delays. In a MIN D represents the switching elements, while in a crossbar they do not really exist. An actual 4×4 crossbar and MIN are shown in Figures 5.6 and 5.12. While at first the Figure 5.22 looks nothing like an actual crossbar or MIN, they are equivalent under certain conditions, namely a uniform distribution of requests.

For the crossbar shown in Figure 5.6, given a uniform distribution of requests, $P1$ contributes an equal amount of traffic to each of the links, $link1$ to $link4$. Defining $\mid P1 \mid$ as the amount of traffic processor $P1$ generates, it is clear that the amount of traffic on $link1$, $\mid link1 \mid$ due to $P1$ is given by

$\mid link1 \mid_{P1} = 1/4 \mid P1 \mid$.

Therefore, the total traffic on link1 is

$\mid link1 \mid = 1/4 \mid P1 \mid + 1/4 \mid P2 \mid + 1/4 \mid P3 \mid + 1/4 \mid P4 \mid$.

Realizing that all the processors in the network are identical, than,

$\mid P1 \mid = \mid P2 \mid = \mid P3 \mid = \mid P4 \mid = \mid P \mid$

in the steady state. Simplifying | $link1$ | we get | $link1$ |=| P |. So under these conditions the Figures 5.22 and 5.6 are equivalent.

The case of the MIN is slightly more complicated, but still holds true provided the probability of passing straight through a switch is equal to the probability of passing through crossed in addition to a uniform distribution of requests. As noted in the section on the MIN model this is definitely the case. Working backwards from link $L1a$,

$| L1a |= 1/2 | L1 | +1/2 | L3 |$

$| L1 |= 1/2 | P1 | +1/2 | P2 |$

$| L3 |= 1/2 | P3 | +1/2 | P4 |$

$| L1a |= 1/4 | P1 | +1/4 | P2 | +1/4 | P3 | +1/4 | P4 |$

but since $| P1 |=| P2 |=| P3 |=| P4 |=| P |$

$| L1a |=| P |$.

So the Figures 5.22 and 5.12 are also equivalent, but under more restrictive assumptions.

Since the model of Figure 5.22 has no connections a further reduction can be made. This is shown in Figure 5.23.

The composed model for the 2×2 mesh shown in Figure 5.20 represents the mesh shown in Figure 5.17. As mentioned earlier, this method of construction will only suffice for very small numbers of processors. In order to simulate larger meshes, it became necessary to simplify the model. Several steps were taken and verified using simulation to obtain the final simplified model. First, the approach taken for the MIN and crossbar was taken, separating the rows as in Figure 5.24. From Figure 5.17 that the traffic on each link is | P |. Realizing this, the model may be broken down into the separate parts shown

in Figure 5.24. However, the simplification can be taken much further than eliminating the replicated nodes. The traffic on each link is still equal to $| P |$, so the simplification shown in Figure 5.25 can be made. Figure 5.25 shows the simplification for the an interior node that is connected to *four* interconnecting links. The edge nodes require a different model. A model for the edge nodes was not developed because their is an imbalance of transactions leaving and entering edge nodes. This imbalance is discussed later.

This simplification is necessary to simulate models with large number of processors. The results of this simplification will be discussed in the next chapter.

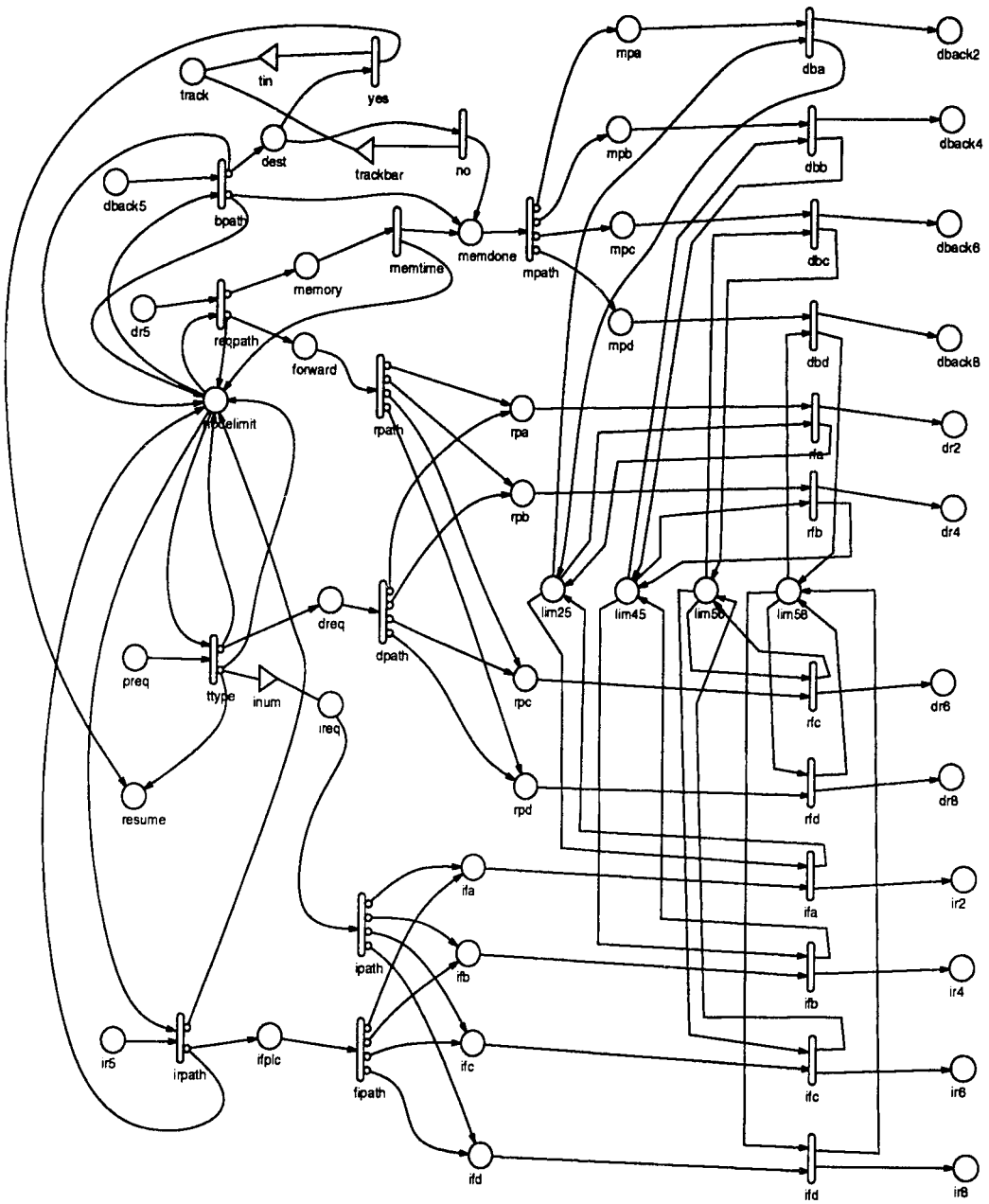


Figure 5.19: SAN sub-model representing a mesh interconnection network node.

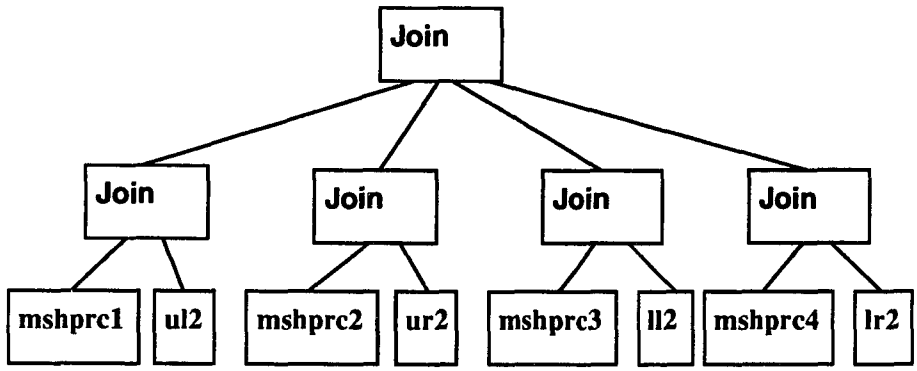


Figure 5.20: Composed model of the 2 x 2 mesh SAN model.

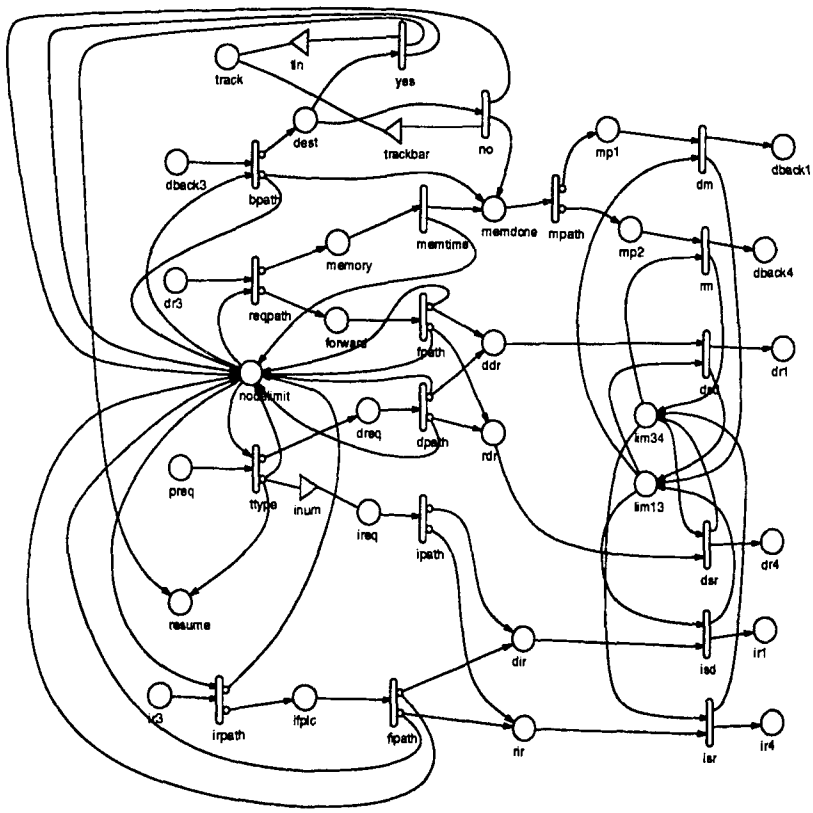


Figure 5.21: Sample SAN node for the 2 x 2 a mesh interconnection network.

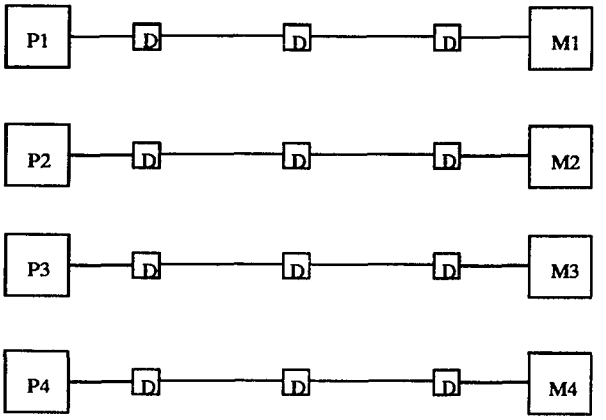


Figure 5.22: Actual SAN model representation of the crossbar and MIN.

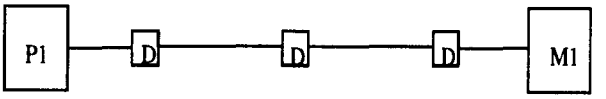
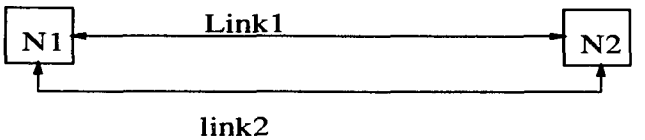


Figure 5.23: Simplified version of the SAN model representation.



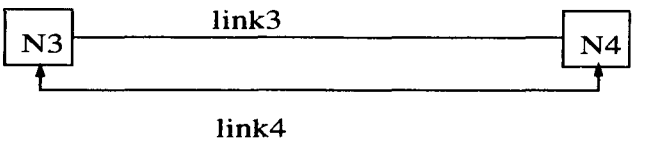


Figure 5.24: Simplified version of the mesh model representation.

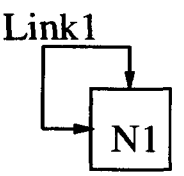


Figure 5.25: Simplification for the internal mesh nodes.

CHAPTER 6

RESULTS OF THE SIMULATION USING THE NBR CALCULATIONS

A single processor bus was used as a gauge for measuring the performance of the multiprocessors. The interconnecting link speed for the single processor bus model evaluation was assumed to be equal to the processor speed. This uniprocessor system achieved a 66% processor utilization, a 12% memory utilization and a 3% link utilization. The utilization figures are calculated through performance variables during simulation. The calculations are based on the amount of time the places corresponding to the processors, memories, and interconnecting links contained a token. A token in a place indicates that the place is being utilized. All graphs with processor performance data include a light line as a reference to this gauge. Furthermore, all results are graphed with their corresponding error bars. Some error bars may not be visible, indicating that the simulation results are very accurate.

6.1 SIMULATION RESULTS FOR THE BUS MULTIPROCESSOR MODEL

Using the *nbr* results from the analytical phase, very favorable simulation results were obtained for bus multiprocessors. Figure 6.1 shows the performance values obtained for several bus multiprocessor sizes. As the figures indicate the *nbr* values are a good

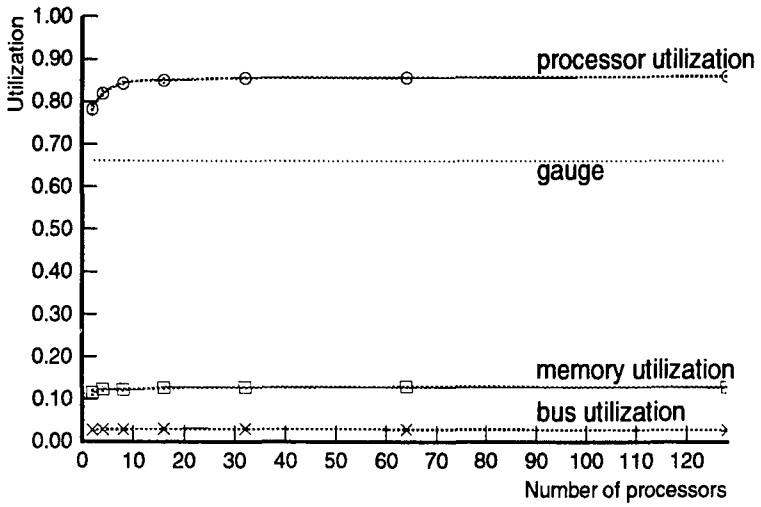


Figure 6.1: Simulation results for the bus multiprocessor with the capability of interleaving bus transactions.

predictor of link speed. The excess performance obtained in the bus multiprocessor is due to the ability to interleave bus requests which was not considered in the *nbr* calculation. This is one method of obtaining more processing capabilities above the uniprocessor bus. Also, the bus and memory utilization remain fairly constant. As an investigation into the effects of contention, the capability of interleaving requests was removed. These results are shown in Figure 6.2. As can be seen, contention for the bus rises rapidly, but it is not until the bus utilization reaches approximately 90% that the processor utilization

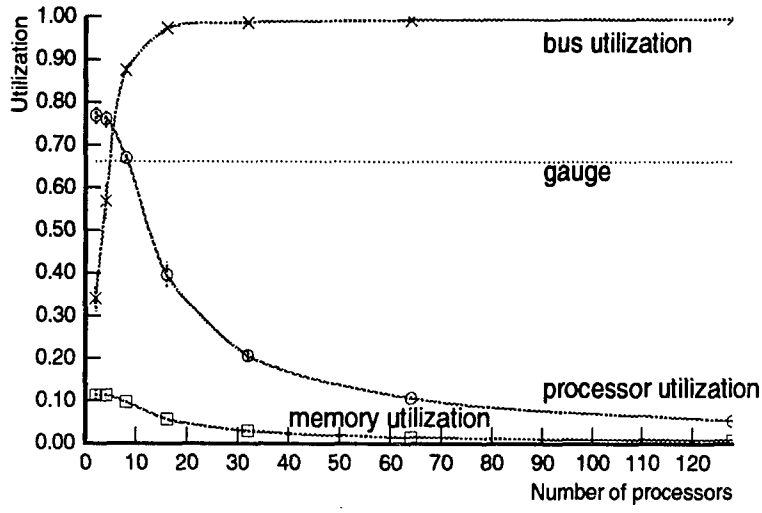


Figure 6.2: Simulation results for the bus multiprocessor without the capability of interleaving bus transactions.

falls below the gauge. Due to the bus being held throughout the memory access time, the contention rises enough to invalidate the assumption about contention made in developing the *nbr* equations. The memory access time becomes the bottleneck, and will remain so regardless of how fast the bus is.

6.2 SIMULATION RESULTS FOR THE CROSSBAR MULTIPROCESSOR MODEL

The simulation results for the crossbar *nbr* values are shown in Figures 6.3, and 6.4, full model and simplified model results, respectively. Both figures demonstrate the accuracy of the *nbr* estimates for the crossbar. As noted earlier the memory utilization remains essentially constant. The link utilization however, increases as the number of processors increase. This is due to the additional transactions used for invalidations and

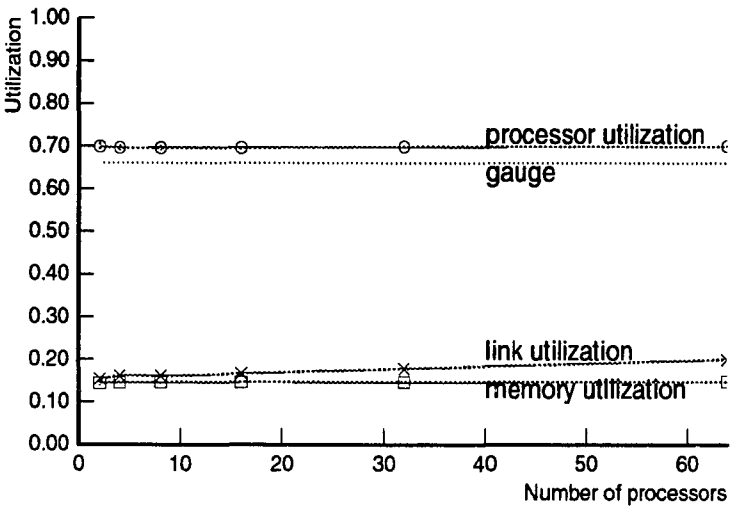


Figure 6.3: Simulation results for the full crossbar multiprocessor model.

acknowledgements This does not appear in the bus, MIN, or mesh model. This is a result of the performance capabilities of the crossbar. The crossbar is inherently an excellent performer, and does not require large increases in the link speed (*nbr*) to maintain high performance levels. It is also more tolerant of contention, and can maintain higher levels of contention and still achieve the desired level of performance. The full crossbar model was only able to run up to 64 processors due to the complexity of the larger models. The 64 processor crossbar required nearly 11.5 hours of cpu time to run. This large amount of time required for simulation was the driving force behind developing simplifications. Figure 6.5 shows the processor utilization resulting from the full model, and the simplified model for comparison. The simplification is not as accurate as the full model, but it is within tolerable limits. The maximum distance between the full and simplified models for the given data is approximately 0.7%. This small sacrifice in accuracy has substantial

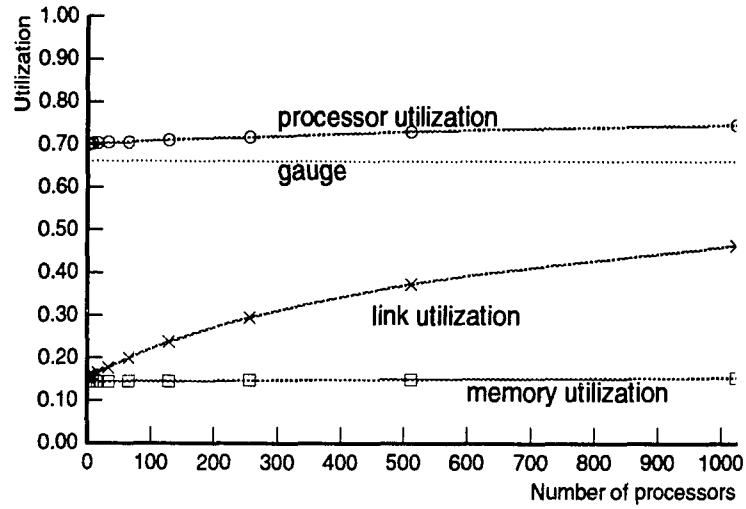


Figure 6.4: Simulation results for the simplified crossbar multiprocessor model.

gains in execution time. The simplification of the 1024 processor crossbar model required little more than 2 minutes of cpu time to complete.

6.3 SIMULATION RESULTS FOR THE MIN MULTIPROCESSOR MODEL

The resulting processor utilizations using the MIN model described in the previous section and the *nbr* results are graphed in Figures 6.6 and 6.7. These are respectively the full model results, and the simplified model results. The results of the full model simulation, Figure 6.6, oscillate around the gauge, however, the gauge does fall within the calculated error of the simulation. While the results of the simplified model are not as accurate as those of the full model they are very close to the uniprocessor gauge. A comparison of the simplified and full model results are shown in Figure 6.8. The results

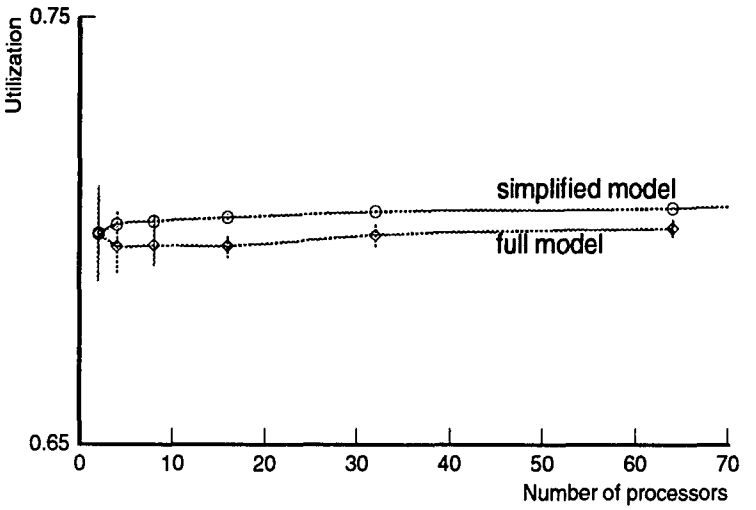


Figure 6.5: Comparison of the simulation results for the simplified and full crossbar multiprocessor models.

of the two are well correlated and the simplification is a very acceptable alternative. The time required to run the full model simulation of a 128 processor MIN is nearly 120 times longer than the time required for the simplified model of the same size.

As before, the memory utilization remains essentially constant, as does the average link utilization. Figure 6.9 shows the utilization for the links of a 128 and 1024 processor MIN. As each stage is traversed on the way to memory the link utilization drops. This is expected for two reasons. First, messages have an increased chance of blocking at each successive stage. Secondly, since the links are held until the request is satisfied, the links closer to the processor will be held longer than those closer to the memory. Another feature of Figure 6.9 is that the slopes of these lines are different, with the smaller multiprocessor having a larger slope. This is not significant because the link utilization oscillates. This oscillation is opposite to that which is seen in the processor

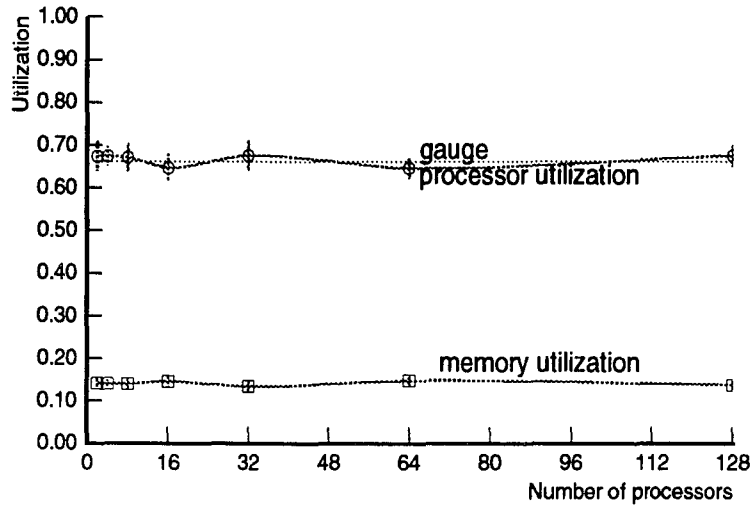


Figure 6.6: Simulation results for the full MIN multiprocessor model.

utilization, lower link utilizations correspond to higher processor utilizations. This result is not surprising since with lower link utilizations memory requests can be serviced faster.

6.4 SIMULATION RESULTS FOR THE MESH MULTIPROCESSOR MODEL

The mesh model is by far the most complex. Only the 4 processor and 9 processor mesh were modeled completely. The results for the full models are shown in tables 6.1 and 6.2. There is a loss of accuracy associated with small mesh interconnection networks, namely for obtaining accurate utilizations for the edge and corner nodes. As the mesh grows in size, this inaccuracy becomes insignificant since the number of interior nodes (nodes connected to *four* links) grows as $(\sqrt{N} - 1)^2$, where N is the number of processors,

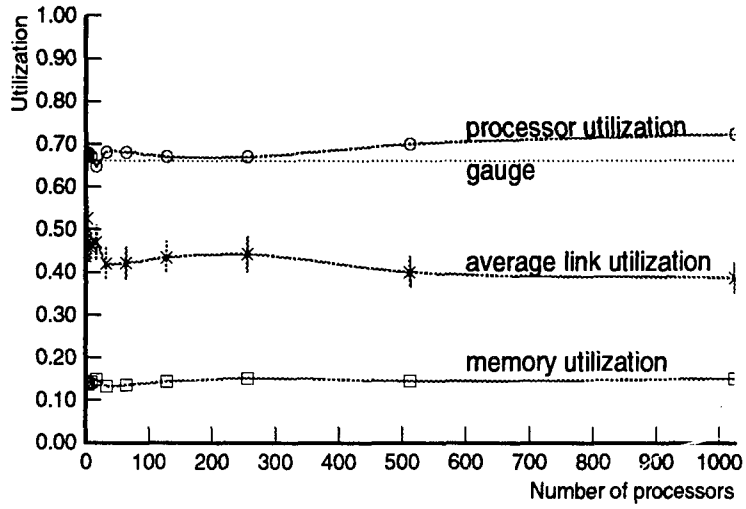


Figure 6.7: Simulation results for the simplified MIN multiprocessor model.

and the edge/corner nodes grows as $4 \times (\sqrt{N} - 1)$. Clearly interior nodes dominate the performance characteristics of large scale mesh interconnection networks.

Table 6.1: Simulation Results of the 4 processor mesh multiprocessor.

Node number	processor utilization	memory utilization	link utilization
1	48 +/- 2%	10 +/- 1%	36 +/- 1%
2	48 +/- 2%	10 +/- 1%	35 +/- 1%
3	49 +/- 2%	10 +/- 1%	35 +/- 1%
4	47 +/- 2%	10 +/- 1%	35 +/- 1%

The inaccuracy is further illustrated in both the full and the simplified model results. The results for processor utilization show a significant performance drop from interior nodes to edge and corner nodes. The inaccuracy of the *nbr* for small meshes, and performance drop for edge nodes are caused by both the model construction and the *nbr* calculation. The inaccuracy of the *nbr* stems from the assumption that each node is connected to 4 links, which is obviously untrue for the edge and corner nodes. This could be

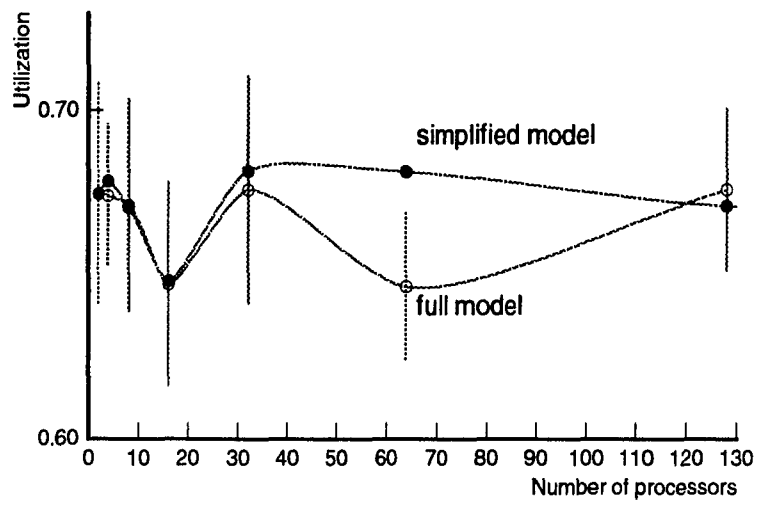


Figure 6.8: Comparison of the simulation results of the simplified and full MIN multiprocessor models.

rectified by enhancing the N_L calculation to better reflect the influence of these nodes, or to perform separate calculations for the edge and corner nodes. Furthermore, the uniform distribution of requests assumption for the model causes a net flow of transactions out of the edge nodes. In any time period, 1/3 of the transactions in the edge nodes are transmitted to the interior nodes while only 1/4 of the transactions in interior nodes (that are connected to edge nodes) are transmitted to an edge node, therefore, the majority of traffic remains in the interior portion of the mesh. This inaccuracy diminishes as the ratio of edge nodes to interior nodes decreases. The corner nodes suffer from this effect the most and exhibit the lowest utilizations in the model. The results for the simplified model are shown in Figure 6.10. Memory utilization again, remains essentially constant, as does the total link utilization. It is important to realize that this loss in accuracy pertains

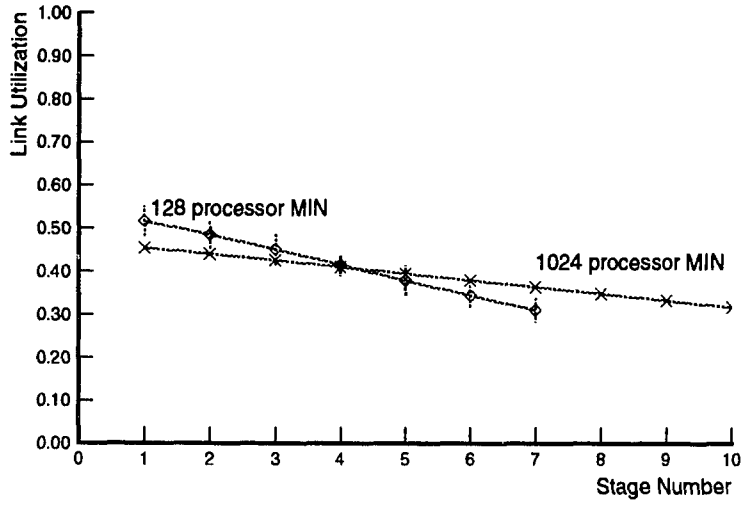


Figure 6.9: Simulation results of the link utilization for 128 and 1024 processor MIN multiprocessor models.

only to very small meshes, 4 and 9 processor meshes, and even in this case the model still exhibits a fair amount of accuracy.

Table 6.2: Simulation Results of the 9 processor mesh multiprocessor.

Node number	processor utilization	memory utilization	link utilization
1	43 +/- 2%	8 +/- .5%	25 +/- 1%
2	53 +/- 2%	12 +/- .7%	36 +/- 1%
3	41 +/- 2%	7 +/- .5%	21 +/- 1%
4	53 +/- 2%	12 +/- .6%	36 +/- 1%
5	61 +/- 2%	15 +/- .7%	46 +/- 1%
6	53 +/- 2%	12 +/- .6%	35 +/- 1%
7	42 +/- 2%	7 +/- .5%	25 +/- 1%
8	53 +/- 2%	12 +/- .5%	37 +/- 1%
9	43 +/- 2%	8 +/- .4%	23 +/- 1%

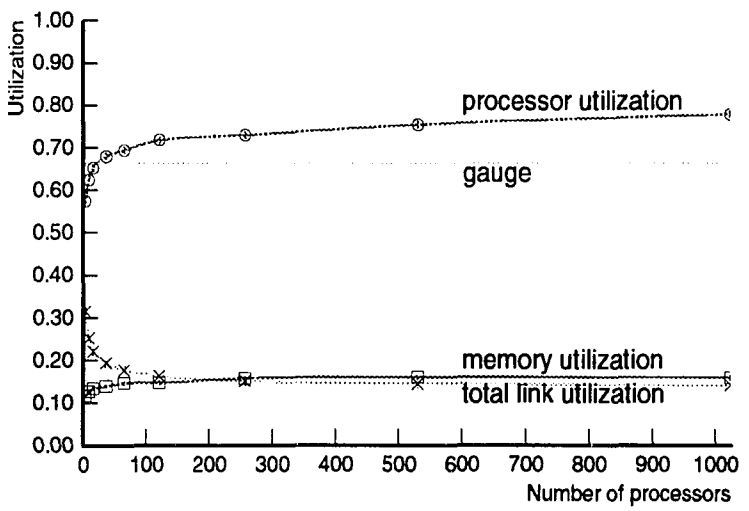


Figure 6.10: Simulation results of the simplified mesh multiprocessor model.

CHAPTER 7

CONCLUSIONS

This thesis presents a two-phase analysis that is a useful aid in the design of multiprocessor computers. The analysis is both simple to use and generates accurate results quickly. Various interconnection networks can be easily investigated and the network bandwidth requirements determined without investing a great deal of time or cost. The *nbr* values may be used to determine the feasibility of a particular multiprocessor interconnection strategy. This helps to quickly eliminate interconnection strategies that will not be capable of supporting the processing load without exceeding the limits of current technology.

The *nbr* calculations are shown to be accurate and flexible. Processors with different characteristics may be considered, easily helping the designer to determine the best possible mating of processor to interconnection network strategy to obtain maximum performance within the current technological constraints. Using this analysis, it can be determined if a given choice of processor will cause the *nbr* of the interconnection network to exceed technology constraints. It can also be determined if the *nbr* of an interconnection network exceeds what is required by a given processing load. This knowledge can be used to reduce the cost in a system, it does not make sense to select a processor whose

performance is so high that an interconnection strategy can not be implemented within current technological constraints, and vice versa. The analysis is also flexible enough to consider hierarchical interconnect strategies, although they are not evaluated here. The accuracy of the *nbr* calculations is evidenced in the simulation results.

The simulation models developed provide a good method for verifying the *nbr* results from the analytical phase of the analysis, as well as providing a vehicle for testing the effects contention and various memory configurations have on the multiprocessor system performance. Due to performance constraints of our system, the model constructions were simplified greatly. This simplification greatly reduced the simulation time and virtually eliminated the constraints on model size. Some accuracy is sacrificed in this model simplification, however, as is evident in the simulation results, the loss of accuracy is marginal. Furthermore, with the simplifications, arbitrarily large multiprocessor interconnection networks may be evaluated with little or no increase in disk space requirements.

Using the analytical and simulation phases in concert offers an excellent aid to designers for the selection of an interconnecting network and the determination of the interconnecting link speed requirements of a multiprocessing computer. The ease of use, the accuracy of the results, and the small time investment required to obtain the results makes this two-phase analysis an ideal starting block for multiprocessor design.

APPENDIX A PROGRAM FOR CALCULATING THE NBR

```
/*********************************************************************/
/* PROGRAM NAME:  THESIS.C                                           */
/* WRITTEN BY:  Earl E. Hokens III                                   */
/* DATE: JULY 5, 1992                                                */
/* PROGRAM DESCRIPTION:  This program codifies my analytical equations*/
/* for evaluating an interconnection network to determine if it is   */
/* capable of supporting a given processor.  It returns numbers that */
/* represent the bandwidth required of each link within the inter-   */
/* connect to maximally support the processors in the system.        */
/*********************************************************************/

/******************* SUBROUTINE DEFINITIONS **************************/

void process_exit(void);
float calc_nbr(float nbr[], float ohinv[], float oh[],
   unsigned long apl[], float num_per_link[], int ut[],
   unsigned long C[], int i, float D[]);
float calc_ohinv(unsigned long total_num_proc, int level,
unsigned long C[]);
float calc_oh (float hit);
float calc_D(float hit);
void set_apl(char net[], unsigned long n[], unsigned long apl[],
   float num_per_link[], unsigned long C[], int i);
void chg_probs(int value, int *next);
unsigned long llog2 (unsigned long n);


/*********************  GLOBAL CONSTANTS  ****************************/

/*********************  GLOBAL VARIABLES  ****************************/

float prob_variables[4];
float *p_read;
float *p_dirty;
float *p_share;
float *percent_share;

/*********************  INCLUDE FILES  *******************************/

#include<stdio.h>           /* ansi C */
#include<stdlib.h>          /* ansi C */
#include<math.h>            /* ansi C */
```

```
/*********************  INPUT FILES  ******************************/

/*********************  OUTPUT FILES  *****************************/

FILE *output1;
char out_line[81];

/*****************************************************************/
/*                          MAIN                                 */
/*                                                               */
/* DESCRIPTION:  THIS SECTION OF THE PROGRAM CONTAINS THE        */
/*               MAIN CONTROLLING LOGIC.                         */
/*                                                               */
/*                                                               */
/*****************************************************************/

void main()
{       /*  MAIN  */
/*****************************************************************/
/*                          HOUSKEEPING                          */
/*                                                               */
/* DESCRIPTION:  THIS SECTION OF THE PROGRAM CONTAINS VARIABLE   */
/*               DEFINITIONS, CONSTANT DEFINITIONS, FILE DEFINITIONS */
/*               AND INCLUDE FILES.                              */
/*                                                               */
/*****************************************************************/

int val_varied = 0;
char net[11];
char inbuf[81];
int level = 0;
unsigned long total_num_proc = 0;
unsigned long n[11];
float num_per_link[11];
float h[11];
float oh[11];
float D[11];
unsigned long C[11];
int ut[11];
float ohinv[11];
float nbr[11];
int i = 0;
unsigned long apl[11];
```

```
char finish = ' ';
int dummy = 0;
int *next = 0;
int lcv = 0;

   p_read = &prob_variables[0];
   p_dirty = &prob_variables[1];
   p_share = &prob_variables[2];
   percent_share = &prob_variables[3];
   next = &dummy;

/*****************************************************************/
/* This section of housekeeping opens files for input and output */
/* The open is then checked to veryfy success of the open        */
/*****************************************************************/

if ((output1 = fopen("thesis.out", "w")) == NULL) {
printf("Error opening file for writing\n");
exit(0);}

/*****************************************************************/

/* This DO WHILE loop is the main loop for the entire program
   the program will continue processing as long as DONE equals 0 */

do
{

   for (lcv = 0; lcv <= 11; lcv++)
   {
   num_per_link[lcv] = 0.0;
   net[lcv] = '0';
   h[lcv] = 0.0;
   oh[lcv] = 0.0;
   ut[lcv] = 0;
   ohinv[lcv] = 0.0;
   nbr[lcv] = 0.0;
   apl[lcv] = 0;
   n[lcv] = 0;
   D[lcv] = 0.0;
   C[lcv] = 1;
   }

   prob_variables[0] = .8;
```

```
   prob_variables[1] = .1;
   prob_variables[2] = .05;
   prob_variables[3] = .06;

   printf("\nWelcome. This program is used to determine the
bandwidth \n");
   printf("requirements for the interconnecting links of a
multiprocessor \n");
   printf("interconnection network for a particular
processor. \n");
   printf("Among other uses, the results may be used to
determine if a \n");
   printf("type of network can sustain the processor
capabilities.\n\n\n");

/* Find the number of heirarchies in the network.  This determines
   the number of iterations needed to calculate the Network
   Bandwidth Requirement (NBR) for the entire network.   */

  printf("How many heirarchical levels are in the interconnection
network\?\n");
  gets(inbuf);
  sscanf(inbuf,"%d",&level);

/* The total number of processors is used to determine,
   probabilistically the amount of overhead required in a
   heirarchical interconnection */

  printf("This theoretical system uses one type of RISC \n");
  printf("microprocessor.  What is the maximum issue rate \n");
  printf("of load/stores per cycle\? \n");
  gets(inbuf);
  sscanf(inbuf,"%f",&nbr[0]);

  oh[0] = 1;
  D[0] = 1;

  printf("\nWhat is the total number of processors in the system\?\n");
  gets(inbuf);
  sscanf(inbuf,"%ld",&total_num_proc);

  for (i = 1; i < (level + 1); i++)
    {  /* for level loop */
```

```
do    /* do loop for getting network type */
{
printf("The type of interconnection used at level %d ",(i - 1));
printf("is:\n\n");
printf("\t1.  Mesh\n");
printf("\t2.  Multistage Interconnection Network\n");
printf("\t3.  Bus\n");
printf("\t4.  Crossbar\n");

printf("\n");

gets(inbuf);
sscanf(inbuf,"%c",&net[i]);

switch (net[i])
{
case '1':  break;
case '2':  break;
case '3':  break;
case '4':  break;
default :  printf("Invalid choice. Try again.\n");
net[i] = '5'; break;
}  /* end switch */

} while (net[i] == '5');       /* end net type do loop */

if (level > 1)
{
printf("How many elements are interconnected in");
printf(" this level of the network\? \n");
gets(inbuf);
sscanf(inbuf,"%d",&n[i]);
}
else
n[i] = total_num_proc;

set_apl(net, n, apl, num_per_link, C, i);

printf("\nWhat is the hit ratio of the cache at this level? ");
gets(inbuf);
sscanf(inbuf,"%f",&h[i]);

        printf("\nWhat is the unit of transfer, or cache line
   size(in words)\n");
```

```
printf("to the cache at this level? ");
gets(inbuf);
printf("\n");
sscanf(inbuf,"%d",&ut[i]);

} /* end for level */

/* set up a loop that will be used to vary certain constants used
in the calculations.  The variations will generate results that
will be used to determine their performance implifications. */

for (val_varied = 0; val_varied < 4; val_varied++)
{     /* for variable */

do    /* do modify, this loop determines when the next variable
should be modified */
{

*next = 0;

for (i = 1; i < (level + 1); i++)
{  /* for level loop */
ohinv[i] = calc_ohinv(total_num_proc, i, C);
oh[i] = calc_oh(h[i]);
D[i] = calc_D(h[i]);
nbr[i] = calc_nbr(nbr, ohinv, oh, apl, num_per_link, ut, C, i, D);

/* output the stuff to the file, then call increment.  increment is
  a routine that varies the proper probability by the proper amount */

/***************************** OUTPUT **********************************/

fprintf(output1, "%s%d\n", "\nTotal levels:  ", level);
fprintf(output1, "%s%d\n", "The level is:  ",i);
fprintf(output1, "%s%c\n", "The network type is:  ", net[i]);
fprintf(output1, "%s%ld\n", "Total number of \
 processors:  ",total_num_proc);
if (level > 1)
 fprintf(output1, "%s%ld\n", "Units covered at this level: ", C[i]);
fprintf(output1, "%s%d\n", "APL at this level  ", apl[i]);
fprintf(output1, "%s%d\n", "Unit of transfer:  ", ut[i]);
fprintf(output1, "%s%f\n", "\nRead probability is:  ",*p_read);
fprintf(output1, "%s%f\n", "Probability of sharing is:  ", *p_share);
fprintf(output1, "%s%f\n", "probability that data \
```

```
is dirty is:  ", *p_dirty);
fprintf(output1, "%s%f\n", "Percentage of processors \
sharing is:  ", *percent_share);
fprintf(output1, "\n%s%f %s", "The NBR is ", nbr[i], "words \
per cycle.\n");
fprintf(output1, "\n%s\n", "***********************************");

/*********************************************************************/

} /* end for level */

chg_probs(val_varied, next);

} while (*next != 1); /*  do modify */
}   /*  end for variable  */

fprintf(output1, "\n%s\n", "-----------------------------------");

printf("\nDo you wish to evaluate another network? (Y or N) ");
gets(inbuf);
sscanf(inbuf,"%c",&finish);
finish = (char) toupper(finish);

} while (finish == 'Y');   /* end of main program loop */

process_exit();

   }   /*  MAIN  */



/*********************************************************************/
/*                          PROCESS EXIT                             */
/*                                                                   */
/* DESCRIPTION:  THIS SECTION OF THE PROGRAM CLOSES FILES AND        */
/*               PERFORMS OTHER TASKS REQUIRED TO TERMINATE THE      */
/*               PROGRAM.                                            */
/*                                                                   */
/*********************************************************************/

   void process_exit(void)
   {

/* Close the files */
```

```
fclose(output1);

      }

/*********************************************************************/
/*           CALCULATE NETWORK BANDWIDTH REQUIREMENT                 */
/*                                                                   */
/* DESCRIPTION:  THIS PROCEEDURE CALCULATES THE NBR FOR THE CURRENT  */
/*               LEVEL OF THE INTERCONNECTION NETWORK.  THE RETURNED */
/*               VALUE HAS UNITS OF WORDS PER CYCLE.                 */
/*                                                                   */
/*********************************************************************/
   float calc_nbr(float nbr[], float ohinv[], float oh[],
unsigned long apl[], float num_per_link[], int ut[],
unsigned long C[], int i, float D[])
   {
   float nbr_ret;

   nbr_ret = (D[i] * ut[i] + oh[i] + ohinv[i]) * nbr[0] * apl[i]
 * num_per_link[i] * C[i];

   return(nbr_ret);
   }

/*********************************************************************/
/*              CALCULATE OVERHEAD (BANDWIDTH)                       */
/*                                                                   */
/* DESCRIPTION:  THIS ROUTINE CALCULATES THE OVERHEAD USED IN THE    */
/*               CALCULATE NBR ROUTINE.  THE VALUE RETURNED HAS THE  */
/*               UNITS OF WORDS PER CYCLE.                           */
/*                                                                   */
/*********************************************************************/
   float calc_ohinv (unsigned long total_num_proc, int level,
unsigned long C[])
   {
   float oh_ret;
   float c_float;
   float total_num_proc_float;

   c_float = C[level] * 1.0;
   total_num_proc_float = total_num_proc * 1.0;

   oh_ret = (1 - *p_read) * (*p_share) * (2) *
(1 + *percent_share * total_num_proc) *
```

```
(1 - (c_float/total_num_proc_float));

   return(oh_ret);
   }

/*************************************************************************/
/*          CALCULATE THE NUMBER OF TRANSACTIONS INITIATED               */
/*                                                                       */
/* DESCRIPTION:  THIS ROUTINE CALCULATES THE NTI.  THIS VALUE HAS        */
/*               UNITS OF TRANSACTIONS PER CYCLE.  THIS VALUE IS         */
/*               USED TO CALCULATE THE NBR.  THE DIFFERENCE BETWEEN      */
/*               NBR AND NTI IS THAT NBR CONSIDER CACHE LINE SIZES.      */
/*               SINCE CACHE LINE SIZES WILL NOT BE ONE WORD, MULTIPLE*/
/*               TRANSACTIONS WILL BE REQUIRED TO COMPLETE EACH REQUEST*/
/*                                                                       */
/*************************************************************************/

   float calc_D(float hit)
   {
   float d_ret;

      d_ret = 2 * ((1 - hit) + hit * (*p_share) * (*p_dirty));

   return(d_ret);
   }

/*************************************************************************/
/*                CALCULATE OVERHEAD (TRANSACTION)                       */
/*                                                                       */
/* DESCRIPTION:  THIS ROUTINE CALCULATES THE NUMBER OF OVERHEAD          */
/*               TRANSACTIONS REQUIRED TO MAINTAIN OPERATION.            */
/*               REMEMBER THAT SOME OF THE CONSISTENCY TRANSACTIONS      */
/*               MAY REQUIRE MORE THAN ONE ACTUAL TRANSACTION.           */
/*                                                                       */
/*************************************************************************/

   float calc_oh (float hit)
   {
   float oh;

   oh = *p_read * (*p_share) * (*p_dirty) * 2 +
*p_read * (*p_share) * (1 - *p_dirty) * (1 - hit) +
*p_read * (1 - *p_share) * (1 - hit) +
(1 - *p_read) * (1 - *p_share) *((1 - hit) + 2 * hit);
```

```
   return(oh);
   }

/*************************************************************************/
/*            SET THE AVERAGE PATH LENGTH OF THIS LEVEL                  */
/*                                                                       */
/* DESCRIPTION:  THIS ROUTINE DETERMINES THE AVERAGE PATH LENGTH         */
/*               THAT EACH TRANSACTION MUST TRAVEL WITHIN THE NETWORK.*/
/*                                                                       */
/*************************************************************************/
void set_apl(char net[], unsigned long n[], unsigned long apl[],
float num_per_link[], unsigned long C[], int i)
   {
   int j;

   if (net[i] == '1')
{
for(j=1; j < i+1; j++)
 apl[j] += (unsigned long) ceil(((double)sqrt((double) n[i]) - 1));
num_per_link[i] = .5;
}
   else
if (net[i] == '2')
{
num_per_link[i] = 1;
for(j=1; j < i+1; j++)
apl[j] += llog2(n[i]); /* log base 2 */;
}
else
if (net[i] == '3')
{
num_per_link[i] = n[i];
for(j=1; j < i+1; j++)
apl[j] += 1;
}
else
{
for(j=1; j < i+1; j++)
apl[j] += 1;
num_per_link[i] = 1;
}
for(j=i; j > 1; j--)
C[i] *= n[j-1];
```

```
   }  /* end subroutine */

/*************************************************************************/
/*                      CHANGE PROBABILITIES                             */
/*                                                                       */
/* DESCRIPTION:  THIS ROUTINE VARIES EACH PARTICULAR PROBABILISTIC       */
/*               VALUE USED IN CALCULATING THE OVERHEAD.                 */
/*                                                                       */
/*************************************************************************/

   void chg_probs(int value, int *next)
   {
   int rread;
   int dirty;
   int share;
   int percent;

   rread = (int) ceil((double) *p_read * 100.0);
   dirty = (int) ceil((double) *p_dirty * 100.0);
   share = (int) ceil((double) *p_share * 100.0);
   percent = (int) ceil((double) *percent_share * 100.0);

   if ((value == 0) && (rread >= 40))
   {
if (rread == 40)
{
*p_read = .8;
*next = 1;
}
else
*p_read = *p_read - .1;
}


   if ((value == 1) && (dirty <= 50))
   {
if (dirty == 50)
{
*p_dirty = .1;        /* reset *p_dirty */
*next = 1;
}
else
*p_dirty = *p_dirty + .1;
```

```
}

   if ((value == 2) && (share <= 25))
   {
if (share == 25) {
*p_share = .05;        /* reset *p_share */
*next = 1;
}
else
*p_share = *p_share + .05;
}


   if ((value == 3) && (percent <= 10))
   {
if (percent == 10)
{
*percent_share = .06;        /* reset *percent_share */
*next = 1;
}
else
*percent_share = *percent_share + .01;
}


   }

/*************************************************************************/
/*                  CALCULATE THE LOGARITHM BASE TWO                     */
/*                                                                       */
/* DESCRIPTION:  THIS ROUTINE CALCULATES THE CEILING OF THE BASE         */
/*               TWO LOGARITHM OF A NUMBER.                              */
/*                                                                       */
/*************************************************************************/

   unsigned long llog2 (unsigned long n)
   {

   unsigned long result;
   unsigned long logarithm;

   result = 1;
   logarithm = 0;
```

```
do
{
result = result * 2;
logarithm++;
} while (result < n);

return(logarithm);
}
```

REFERENCES

[1] L. N. Bhuyan, Q. Yang, and D. Agrawal, "Performance of Multiprocessor Interconnection Networks," *IEEE Computer Magazine*, pp. 25–37, February 1989.

[2] K. Hwang and D. Degroot, *Parallel Processing for Supercomputers and Artificial Intelligence.* New York, New York: McGraw-Hill, 1988.

[3] G. Almasi and A. Gottlieb, *Highly Parallel Computing.* New York, New York: Addison Wesley, 1989.

[4] A. L. DeCegama, *The Technology of Parallel Processing.* Englewood Cliffs, New Jersey: Prentice Hall, 1989.

[5] L. Bhuyan, "An Analysis of Processor-Memory Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, pp. 279–283, March 1985.

[6] A. Agrawal, "Limits on Interconnection Network Performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 398–412, October 1991.

[7] C. lin Wu and M. Lee, "Performance Analysis of Multistage Interconnection Network Configurations and Operations," *IEEE Transactions on Computers*, vol. 41, pp. 18–27, January 1992.

[8] J. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers*, vol. C-30, pp. 771–780, October 1981.

[9] D. P. Agrawal and V. K. Janakiram, "Evaluating the Performance of Multicomputer Configurations," *IEEE Computer Magazine*, pp. 9–27, June 1985.

[10] L. Bhuyan and D. P. Agrawal, "Design and Performance of Generalized Interconnection Networks," *IEEE Transactions on Computers*, vol. C-32, pp. 1081–1090, December 1991.

[11] J. A. Couvillion, R. Freire, and et al., "Performability Modeling with UltraSAN," *IEEE Software Magazine*, pp. 69–80, Sept 1991.

[12] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic Activity Networks: Structure, Behavior, and Application," in *Proc. Int'l Workshop Timed Petri Nets*, pp. 106–115, 1985.

[13] W. H. Sanders, "Construction and Solution of Performability Evaluation Tool Based on Stochastic Activity Networks," tech. rep., Computing Research Lab U of Michigan, Ann Arbor, Mich, 1988.

[14] R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer Magazine*, pp. 5–16, February 1990.

[15] R. Piepho and W. Wu, "Comparison of RISC Architectures," *IEEE Micro*, pp. 51–62, August 1989.

[16] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *IEEE Computer Magazine*, pp. 49–58, June 1990.

[17] S. Vaughan-Nichols, "Catch as Cache Can," *Byte Magazine*, pp. 209–215, June 1991.

[18] S. A. Przybylski, *Cache and Memory Hierarchy Design.* San Mateo, California: Morgan Kaufmann Publishers, INC., 1991.

[19] H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers*, vol. C-20, pp. 153–161, February 1971.

[20] C. lin Wu and T. yun Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-29, pp. 694–702, August 1980.

[21] J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," in *In Proc. 6th Int. Symposium on Computer Architecture*, pp. 168–177, 1979.

[22] B. Makrucki and T. Mudge, "Probabilistic Analysis of a Crossbar Switch," tech. rep., University of Michigan, 1981.

[23] C. Kruskal and M. Snir, "Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Transactions on Computers*, vol. C-32, pp. 1091–1098, December 1983.

[24] K. Hwang, *Advanced Computer Architecture Parallelsm Scalability Programmability.* New York: McGraw-Hill, Inc., 1993.

[25] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer Magazine*, pp. 12–24, June 1990.

[26] A. Agarwal, R. Simoni, and et al., "An Evaluation of Schemes for Cache Coherence," in *In Proc. 15th Int. Symposium on Computer Architecture*, pp. 280–289, 1988.

[27] M. Tomašević and V. Milutinović, "A Simulation Study of Snoopy Cache Coherence Protocols," in *Hawaii International Conference on System Sciences*, pp. 427–436, 1992.

[28] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224–234, 1991.

[29] D. Lenoski, J. Laudon, and et al., "The DASH Prototype: Implementation and Performance," in *In Proc. 19th Int. Symposium on Computer Architecture*, pp. 92–103, 1992.

[30] D. Lenoski, "The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor," Ph.D. dissertation, Stanford University, 1991.

[31] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar," tech. rep., Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.

[32] D. Cheriton, H. Goosen, and P. Boyle, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE Computer Magazine*, pp. 33–46, February 1991.

[33] Z. Vranesic, M. Stumm, and et al, "Hector: A Hierarchically Structured Shared-Memory Multiprocessor," *IEEE Computer Magazine*, pp. 72–78, January 1991.

[34] J. Archibald and J.-L. Baer, "An Economical Solution to the Cache Coherence Problem," in *In Proc. 11th Int. Symposium on Computer Architecture*, pp. 355–362, 1984.

[35] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, pp. 273–298, November 1986.

[36] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing.* New York: McGraw-Hill, Inc., 1990.

[37] intel, *Multimedia and Supercomputing Processors*, 1992.

[38] Motorola, "Technical Summary 32-Bit RISC Microprocessor–MC88100," tech. rep., Motorola inc., 1991.

[39] Motorola, "Technical Summary MC88110 RISC Microprocessor," tech. rep., Motorola inc., 1991.

[40] Motorola, "Application Note DRAM Interface to the MC88200 M Bus," tech. rep., Motorola inc., 1991.

[41] M. A. Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems.* Cambridge, MA: MIT Press, 1987.

[42] H. Jiang and L. N. Bhuyan, "MVAMIN: Mean Value Analysis Algorithms for Multistage Interconnection Networks," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 189–201, July 1991.

[43] P. G. Harrison and N. M. Patel, "The Representation of Multistage Interconnection Networks in Queuing Models of Parallel Systems," *Journal of the acm*, vol. 37, pp. 863–898, October 1990.

[44] S. Caselli and G. Conte, "GSPN Models of Concurrent Architectures with Mesh Topology," in *IEEE Petri Nets and Performance Models - 1991*, pp. 280–289, 1991.