

1994

Performance Analysis of Multiprocessor Disk Array Systems Using Colored Petri Nets

Kurt R. Almquist
University of Rhode Island

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Terms of Use

All rights reserved under copyright.

Recommended Citation

Almquist, Kurt R., "Performance Analysis of Multiprocessor Disk Array Systems Using Colored Petri Nets" (1994). *Open Access Master's Theses*. Paper 1108.
<https://digitalcommons.uri.edu/theses/1108>

This Thesis is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

**PERFORMANCE ANALYSIS OF
MULTIPROCESSOR DISK ARRAY SYSTEMS
USING COLORED PETRI NETS**

**BY
KURT R. ALMQUIST**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING**

UNIVERSITY OF RHODE ISLAND

1994

MASTER OF SCIENCE THESIS


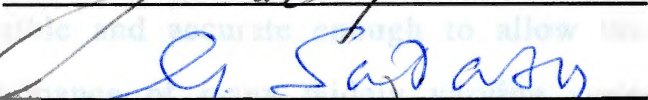
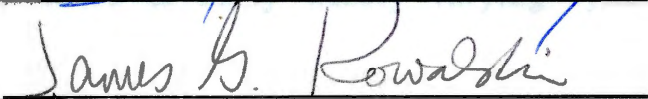
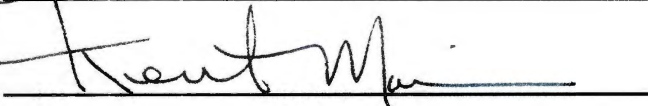
OF

KURT R. ALMQUIST

APPROVED:

Thesis Committee

Major Professor

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1994

Abstract

Due to the increasing gap between the performance of the processors and secondary storage systems, the design of the storage systems has become increasingly important. Arrays of interleaved disks are a popular method of increasing the performance of secondary storage systems. In order to optimize the performance and configuration of the disk arrays, performance evaluations are required. This paper presents a Colored Petri Net simulation model which can represent various configurations of systems containing multiple processors connected to a disk array system across a single stage interconnection network. This flexible model allows many system parameters to be varied such as number of processors, buses and disks in the array and the delay distributions associated with each. The performance estimates produced by this model are validated in this paper against those found in other models and found to be in good agreement. This paper shows that the CPN model presented here is flexible and accurate enough to allow the model to estimate the performance of many widely varying system configurations.

List of Figures

Introduction

1	System Configuration	6
2	Resource Contention Points in the System	7
3	Simplified Version of the CPN Model	11
4	Array Response Time vs Number of Disks	21
5	Disk Array Response Time vs Array Request Size	21
6	Disk Array Response Time vs Number of Buses	27
7	Utilization vs Number of Buses	28
8	Disk Array Response Time vs Proportion of Small Accesses	34
A.1	Processor Page of the CPN Model	41
A.2	Interconnection Network Page of the CPN Model	42
A.3	Disk Array Page of the CPN Model	43
A.4	Declaration Page of the CPN Model	44 - 45
B.1	A Simple Resource Contention CPN Model	50
D.1	Typical Results from a CPN Simulation Run	65

Chapter 1:

Introduction

The performance of processors and semiconductor memories is increasing at a much greater rate than I/O systems such as magnetic memories. Therefore, the performance of the I/O systems is impacting increasingly upon the total system's performance to the point where it can become the source of a performance bottleneck in the system. The throughput of the I/O system can be increased by replacing a single disk I/O system with a disk array in which data may be placed on different disks so it can be accessed concurrently. [1,2,3,4].

Many different organizations of disk arrays have been proposed in the current literature [2,3,8]. In order to understand the benefits and costs of each disk array configuration, it is important to have a method for the estimation of the whole system's performance. This will allow the system designer to understand the effects of various system elements upon the system's performance.

There are two types of models that are generally used for the performance analysis of systems. The first is an analytical model, which reduces the system's functionality to a set of equations. The equations are then used to estimate the system's performance. The second is a simulation model, which generally encapsulates the system's functionality into a model in a more direct manner. The

simulation model is then executed to emulate the system's performance.

Several analytical models have been developed which are based upon many simplifying assumptions to allow the system to be described by a usable set of equations. While these equations allow the quick generation of results, they can also describe only a limited or unrealistic set of system configurations. One such example is in a paper by Lee and Katz where an analytical model is developed which assumed that each processor issues a new request for a block of data whenever any of the subblock data requests from the previous request are finished.[3] This assumption implies that all the subblock data requests generated from a request for a block of data finish their disk accesses at the same time and that each processor spends no time processing the data which it has just received. This is not a realistic assumption because in a real system each disk request may have a different service time because of the starting position of the head on each disk, or a different number of requests present at each disk.

In a paper by Yang, Hu and Yang, a more realistic set of assumptions about the disk array and how it processes requests is presented. However, this model can neither address the relationships associated with the interconnection network (IN) which connect the processors and the I/O system nor can it handle different size data accesses within the same run.[1]

As shown above, a common problem associated with existing models is that the assumptions which are made to enable the system to be characterized by a set of equations also limit the model's ability to handle all the different parameters which are important in a system.

The assumptions are as follows:

1. Each processor generates a request for a piece of data stored on the disk array. The request is a result of some action by the program which requests it. The data request is captured by a disk controller, called disk requests. The disk requests are then transferred to the appropriate disk where the data is stored. The access time for the data is the time required to transfer the data to the disk.
2. The array requests are, which is the number of data requests which a single array request, are always dependent upon the number of the disk array which is the access time, the processor, the processor group size, and the request size. Therefore, the array requests can be generated in any order only. Hence, all of the data.
3. The individual disk request of an array request can be of different sizes, due to the fact that the amount of data requested is not the same for all the disks and the different disk sizes are used. This is the reason for the random grouping of data. Hence, the data is not the same for all the disks.
4. It cannot be guaranteed that a data request is always captured upon the completion of a data request. This request upon the completion of the data request and the processor is within the processor processor request.

Chapter 2:

Guiding Assumptions and System Description

The model presented in this paper tries to more accurately describe a real system by expanding upon the system assumptions described in reference [1]. The assumptions are as follows:

1. Each processor generates a request for a block of data stored on in the disk system. The request for a block of data, called a logical disk request or an array request, is replaced by several subblock requests, called disk requests. The disk requests are then transferred to the appropriate disk where the subblock is stored. The separate disks can then service the disk requests in parallel.
2. The array request size, which is the number of disks accessed by a single array request, can change depending upon various attributes of the disk array such as the subblock size, the parity scheme, the parity group size, and the request type. Therefore, the array requests cannot be guaranteed to access either only one or all of the disks.
3. The individual disk requests of an array request may finish at different times due to both the interference between disk requests at each of the disks, and the different seek times on each disk due to the random starting position of each disk's head.
4. It cannot be guaranteed that a new array request is always issued upon the completion of a disk request. This depends upon the workload of the I/O system and the frequency at which the processor generates requests.

5. Each processor is capable of multiprocessing. Therefore, more than one array request generated by the same processor may exist at the same time.
6. The size of the traffic transferred across the interconnection network, either the data requests or the data blocks, should be allowed to be variable within a single simulation run. It cannot always be assumed that each data block is the same size for all processors in the system.
7. The interconnection network is made of one or more buses which connect the processors to the disks in the disk array. The number of buses in the system cannot always be assumed to be enough to support the workload of the system.

These assumptions accurately describe a real system containing a disk array I/O system. In the following a model based on the above assumptions about a system containing a disk array is presented. The model is a simulation model which was created using Colored Petri Nets (CPN). CPNs, as most simulation modeling tools do, allow the user the flexibility to model in detail whatever area is deemed of interest in the system.

The model consists of several independent processors connected to a single disk array I/O system via an interconnection network (IN) as shown in Figure 1. Figure 2 shows the points of resource contention which will be described in the following paragraphs.

A disk array is an I/O system which replaces a single disk with a collection of disks. In a single disk I/O system a block of data is stored usually together on the disk. In contrast, in a disk array

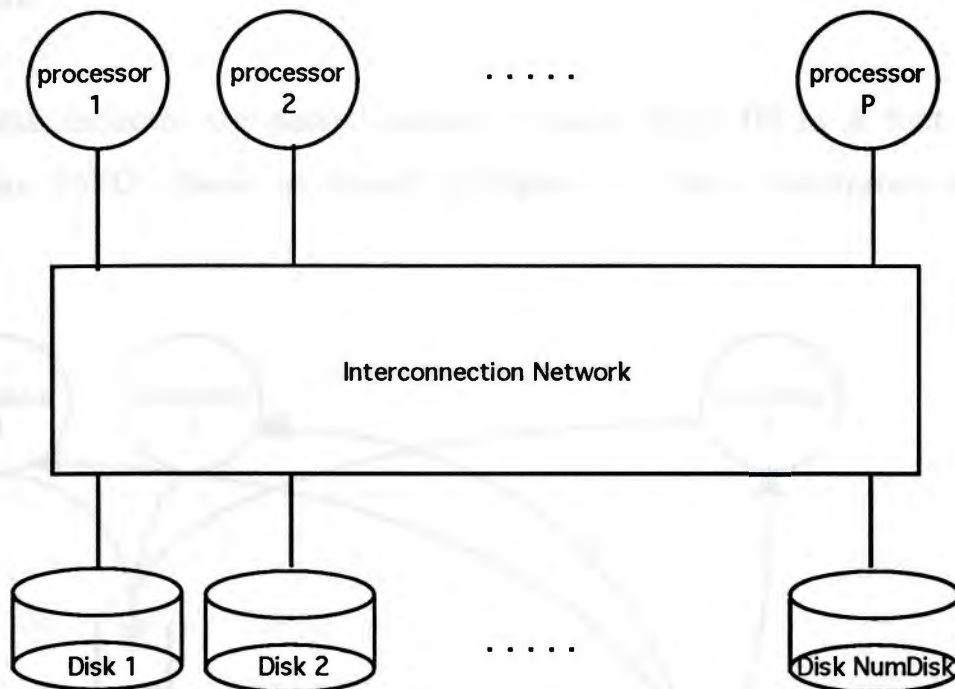


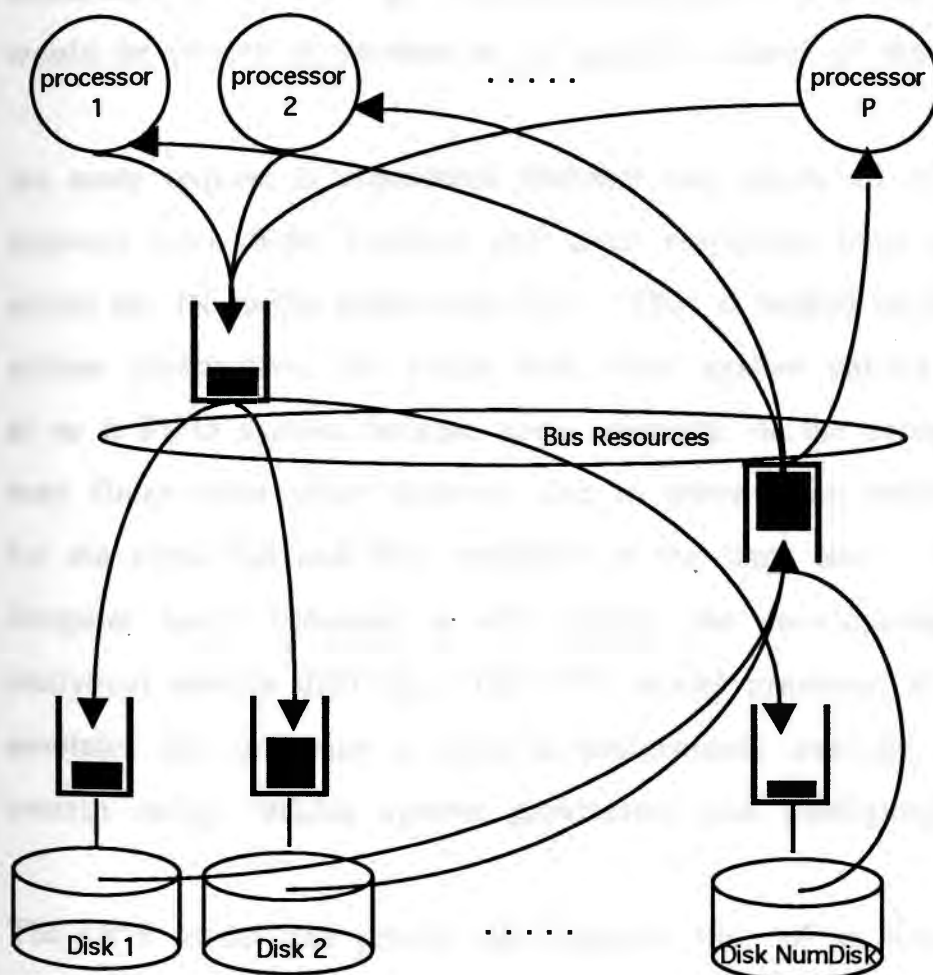
Figure 1 System Configuration

system this block of data can be broken into one or more subblocks which are then stored on separate disks. Because each of the disks in the disk array can be accessed concurrently, the block of data can be accessed more rapidly than on a single disk system.

Each processor can generate a logical disk request, hereafter called an array request, for a block of data from the disk array which in turn is broken into several disk requests. The number of disk requests per array request varies depending upon several

parameters such as the size of the data requested, the amount of interleaving between the disks and the parity scheme of the disks. Thus one or more disks can be accessed by a single logical disk request.

The disk requests are passed across a single stage IN in a first-in, first-out (FIFO) queue as shown in figure 2. Once transferred to the



Resource Contention Points in the System

Figure 2

disk, the disk requests are distributed to the assigned disk. Each disk handles its requests in a FIFO queue fashion.

The results of the individual disk requests are then transferred back to the CPU via the IN using a FIFO queue like the one used to transfer the request to the disks. As figure 2 shows, the IN queues leading to and departing from the disks share the same IN resources. If both IN queues are vying for a bus resource then one would be chosen at random to be granted control of the bus.

An array request is considered finished only when all of its disk requests have been handled and their responses have returned across the IN to the originating CPU. Thus if looked on from a system perspective, the whole disk array system cannot be looked at as a FIFO system because some elements of the array request may finish after other requests due to other array requests vying for the same bus and disk resources at the same time. This irregular queue behavior is what makes the development of analytical models difficult. The CPN model presented in this paper emulates this behavior to allow a performance analysis of this system using various system parameters and configurations.

The CPN model can predict the response time of an array request, and analyze the disk, interconnection network and processor utilization under various system configurations and workloads. This model is validated through a series of measurements and

compared with the findings presented in [1]. This model is used to perform a quantitative evaluation of the disk array's performance for different IN and disk data integrity configurations. The model presented is fairly general and could be used by disk array or system designers to study the effects of various system parameters and configurations.

3.1 A Functional Description of the Disk Array Model

The following is a functional description of how a disk request is generated and handled in the system, which is executed. The function and derivation of the model's variables, which are explained, are described in section 3.2.

Figure 3 shows a simplified version of the Control Unit that model which will be used for illustrative purposes. The main CPU model is included in Appendix A.

There are P independent processes that generate array requests. The processes are represented by CPU threads which reside in the CPU Processing Table (see of figure 3). One of the attributes associated with each entry is the time t_i it is available for use. When the simulated time reaches the time t_i , which is processed when it is enabled, that entry moves to the Device Time Request table where a set of N disk access requests (DAR) requests are issued. The set of disk requests generated at the same time represents an array request.

Chapter 3:

The Colored Petri Net Disk Array Model

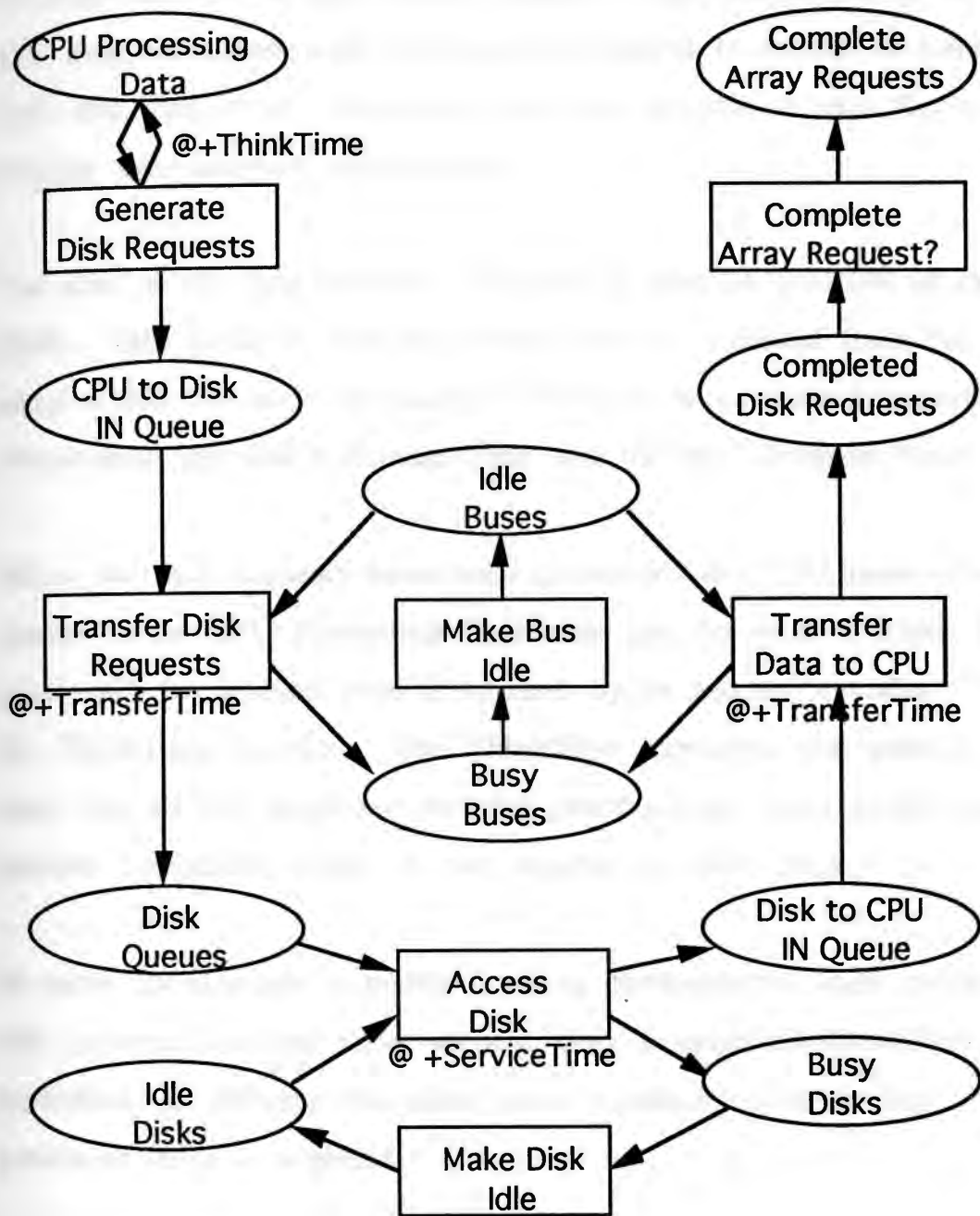
This chapter describes the Colored Petri Net model of a system which contains a disk array I/O subsystem. The chapter is broken up into two parts, the first describes the functionality of the system and the second describes the parameters used in the model.

3.1 A Functional Description of the Disk Array Model

The following is a functional description of how a disk request is generated and handled in the system which is modeled. The limits and derivation of the model's variables, which are capitalized, are described in section 3.2.

Figure 3 shows a simplified version of the Colored Petri Net model which will be used for discussion purposes. The actual CPN model is included in Appendix A.

There are P independent processors that generate array requests. The processors are represented by CPU tokens which reside in the CPU Processing Data node of figure 3. One of the attributes associated with each token is the time it is available for use. When the simulated time reaches the time at which a processor token is enabled, that token moves to the Generate Disk Requests token where a set of N disk access request (DAR) tokens are made. The set of disk requests generated at the same time represents an array request.



Simplified Version of the CPN Model

Figure 3

Another attribute of each array request is the assignment of disks. The disk associated with the first disk request is chosen at random from the disk array. Hereafter, the disks associated with the array request are assigned sequentially.

The size of the data subblock requested is also an attribute of the DAR. Thus different size data blocks may be accessed from the disk array within the same simulation. The size of data block accessed affects both the disk's Service Time and the bus' Transfer Time.

When the disk requests have been generated, the CPU token then returns to the CPU Processing Data node and the time at which the token will be enabled next is updated by an amount calculated by the ThinkTime function. The ThinkTime represents the amount of time that all the processes for that processor are busy performing internal operations which do not require the disk array.

In order to simulate a multiprocessing environment, each processor will generate another array request after a specified ThinkTime, regardless of whether the other array requests made by that processor have completed.

The disk requests then enter into CPU-to-Disk interconnection network (IN) queue to be sent across the IN to the disk array. The single stage IN will have B buses. A Disk-to-CPU IN queue exists to handle the traffic from the disks to the processor. The elements within each queue are handled in a first-in first-out (FIFO) fashion

but both queues contend for the same bus resources. If both IN queues have an token contending for the same bus resource, then one of the tokens is granted the bus resource at random. Therefore, due to the possible contention between the two IN queues, the data flowing through the IN cannot be considered to be FIFO as a whole. The bus resource will remain busy for an amount of time, called the Transfer Time, which is related to the size of the data being transferred and the data transfer rate of the bus. The other queue will wait until there is a bus resource available before proceeding.

Once the disk request token passes across the IN, it enters the disk array. There are NumDisk disks in the disk array. The disk request token will wait until the disk resource token it requires is available. When the required disk is available, the disk request is granted access to the disk. The disk is then unable to process another request until this access is complete. The amount of time the access takes is called the disk's Service Time which is a function of the disk's SeekTime, the Rotational Latency and the Disk Access Time. The disk request token is replaced by a data token which can be a different size than the disk request. The data token is also not available until the disk access is completed. If two DARs are waiting for the same disk, then one is chosen at random to be serviced. The other DAR must wait for the disk to become available again before it can be serviced. Disk accesses to different disks can be performed in parallel.

After the disk access has completed, the data enters the Disk-to-CPU IN queue to be transferred to the processor. As for the disk request, this queue is served internally in a FIFO fashion and externally in contention for bus resources with the CPU-to-Disk IN queue.

Once across the IN, the data subblock waits at the processor for all other data subblocks in its array request to arrive. Once all arrive, the array access is complete. Therefore, in contrast with reference [2] the array request processing does not complete when one of the disk requests is finished. In addition, like reference [1] the array request processing as a whole is not completed in a FIFO fashion due to the handling of the various disk requests at each disk.

Although it may appear that this model only simulates reads from a disk, it also accurately describes the case where a write to a disk is performed in which the write has a completion handshake that is the same size as a read disk request. This is true because, in a system which has handshaking, the amount of time the IN and the disk array are busy would be the same whether the piece of data is passing to or from the disk.

The main disadvantage of a CPN is that if the modeler is not careful the model can get too complex to be analyzed. This is due to the direct relationship between the CPN model's complexity and the size of the state matrix related to the model. In addition, as the state matrix gets larger the simulation model executes more slowly. It

was found that performing the simulations on a higher performance platforms with more RAM available resulted the ability to simulated more complex models, and the current models can be simulated more quickly. Therefore, the modeler must balance the amount of detail in the model and the host computer's ability to handle the complexity contained in the model.

In order to extract data from the model's outputs, a C program was written which extracts the CPU, Bus and Disk utilization data from the raw data produced in the simulation. This program is shown in Appendix C. If different information were required by the modeler, the program could easily be altered to extract it.

The system modeled has several irregular queue characteristics which would make the development of analytical queuing models difficult. The CPN model developed emulates this behavior to allow a performance analysis of this system to be performed using various system parameters and configurations.

3.2 Description of System Parameters

This section describes the formulas and limits of the parameters which were referenced in the previous sections.

- The ThinkTime function is user definable and for this model has been set to an independent, exponentially distributed random variable with mean Z as it was in [1].
- N is the number of disk requests in an array request. Its value is determined by several factors such as the amount of

declustering between disks and the parity scheme used. The value of N can range from 1 to the number of disks in the disk array. The number of disk requests generated by each processor, N, can either be constant for all processors or variable based upon the system being studied.

- A DAR is a disk access request. There are N DARs generated to represent each array request. The information stored in a DAR for this model is: The originating processor, which element of the array request it is, the disk to be accessed, the number of elements to be accessed, and the size of the data block request. The assignment of disks to the different disk requests of an array request is done sequentially. This means that the second DAR accesses the disk $((\text{Disk} + 1) \bmod \text{NumDisk})$ and so on until the N-th DAR accesses disk $((\text{Disk} + N - 1) \bmod \text{NumDisk})$. The term NumDisk indicates the number of disks in the disk array. Thus the 'mod NumDisk' term prevents accesses to disk numbers greater than number of disks in the disk array.

- The bus's Transfer Time T_t

$$= \frac{\text{(size of transferred request or data (in bytes))}}{\text{(transfer rate of bus (in bytes per sec))}}$$

- The disk's Service Time

$$= (\text{Seek Time} + \text{Rotational Latency} + \text{Data Access Time})$$

as defined in references [1, 2, 3 and 4].

- Seek Time (T_s) = time to get the head to the correct track of the disk

$$T_s = T_a * X + T_b * X + T_c$$

As defined in reference [4]

Chapter Ta = (10*minSeek + 15*avgSeek - 5*maxSeek)/(3*numCyl)

Experie Tb = (7*minSeek - 15*avgSeek + 8*maxSeek) /(3*numCyl)

Tc = minSeek

The nei where minSeek, avgSeek, maxSeek and numCyl are
previous parameters of the disk drives used.

those and as defined in reference [1]

In partic X = |(t1) - (t2)| of the analytical model developed in

reference where t1 and t2 are random numbers from between 0 and
values the number of tracks on a disk, T. This makes the model
reference more realistic by giving X a mean distribution of (T/3).

dis - Rotational Latency (Tr) = time to get head to correct data block
or sector with-in the track)

The As defined in [1,4] reference [1] was the case of three

listed in Tr = random (0.. time for a full disk rotation)

to - Data Access Time (Ta) = time to read or write data to disk

adequ As defined in [4] system's level. To comply with this in the

CPU Ta = (time for a full disk rotation) * (# of bytes accessed)
(number of bytes in a track)

Another simplifying assumption made in reference [1] was that all
array requests were of a similar magnitude with the same size.

This means that the size of the data requests per track had the same
request size. To get better control over the distribution for all

array requests made in a distributed system configuration. This was
not difficult to comply with as the CPU model was designed to allow

these parameters to either be constant or varied. Finally, in

reference [1] the statistical data requests of each array request

were assumed to be independent of each other. To comply with

Chapter 4:

Experimental Validation of the CPN Model Results

The method used to validate the CPN model described in the previous chapter was to compare the results of the CPN model to those found in models of similar systems presented in other papers. In particular, the results of the analytical model developed in reference [1] were compared to those of the CPN model for same values of system parameters. The analytical model presented in reference [1] was chosen because the assumptions made in developing that model were very similar to those of the CPN model.

The assumptions made in reference [1] were the same as those listed in Chapter 1 for the CPN model with the following exceptions: In reference [1] it was assumed that the number of buses is always adequate to support the system's load. To comply with this in the CPN model, the number of buses in the IN was specified to be large enough that the IN imposed no limitations on the rest of the model. Another simplifying assumption made in reference [1] was that all array requests made in a particular simulation were the same size. This means that the size of the data accesses per disk and the array request size N are both constant across all the processors for all array requests made in a particular system configuration. This was not difficult to comply with as the CPN model was designed to allow these parameters to either be constant or varied. Finally, in reference [1] the individual disk requests of each array request were assumed to be independent of each other. To comply with

this would require the method of disk request generation to be altered in the CPN model. Because this was determined to be a weak relationship in reference [1], the method of disk request generation in the CPN model was not altered. Thus, in the CPN model the disk requests which originate from the same array request will access disks sequentially from some arbitrary first disk as described in Chapter 3. Therefore, it was possible to satisfy all the assumptions made in reference [1], with the exception of the independence of disk requests accessed by the same array request which was considered a weak assumption.

Because of the complexity of the systems modeled in reference [1], the length of time required for each simulation run using the CPN model was quite long. Therefore, the length of the simulation runs had to be limited. On average, for a system which was of the complexity of the ones presented in this section, the amount of time to perform a simulation run for a range of values would be around 24 hours. Limiting the length of the simulation runs can lead to significant errors when the data varies a great deal such as at high system load. Therefore, it cannot be guaranteed that the simulation results produced are within the guidelines normally used for determining when to end a simulation run. However, the simulation runs were extended as long as time and RAM allowed in order to minimize these errors.

As defined in reference [1], the utilization of the disk array system is a function of the rate at which requests arrive at the disk array

and the service rate of the disk array. The service rate of the disk array is generally constant and independent of the arrival rate. Thus variations to the disk array's utilization are induced mainly by variations the disk array's arrival rate, called Lambda. Lambda is defined as follows:

$$\text{Lambda} = (N * P) / (\text{NumDisks} * Z)$$

where N, P, NumDisks and Z are defined in section 3.2

In order to exercise the CPN model at disk utilizations over its range, lambda will be varied in two different simulation runs. In the first run the number of disks in the array is varied and in the second run the number of elements in an array request is varied. As in reference [1], other system parameters, as defined in section 3.2, were set to typical values as follows: $T_a = 0.4632\text{ms}$, $T_b = 0.0092\text{ms}$, $T_c = 2\text{ms}$, $\text{NumCyl} = 949$, size of data accessed from each disk, the subblock size, = 4 kbytes and the average transfer rate was 0.6023 msec/kbyte across the IN. As in reference [1], the disk's Data Access Time, which is the amount of time to actually read the data from the disk, was not included in the disk's Service Time calculation. In addition, number of processors P was set to 10 and the mean think time Z was set to 100 msec.

In the first comparison, which is shown in figure 4, the disk utilization was varied by altering the number of disks from 30 to 100. The value of N was set to 10. It can be observed from this

figure that the CPN model's average disk array processing time closely matches those found the analytical model in reference [1].

Array Response Time vs Number of Disks

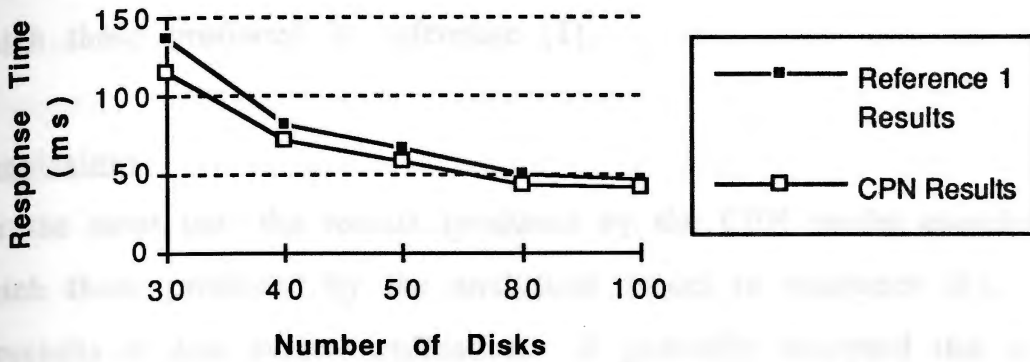


Figure 4

The main reasons for the discrepancies which do exist are discussed in the conclusion portion of this section.

Disk Array Response Time vs N

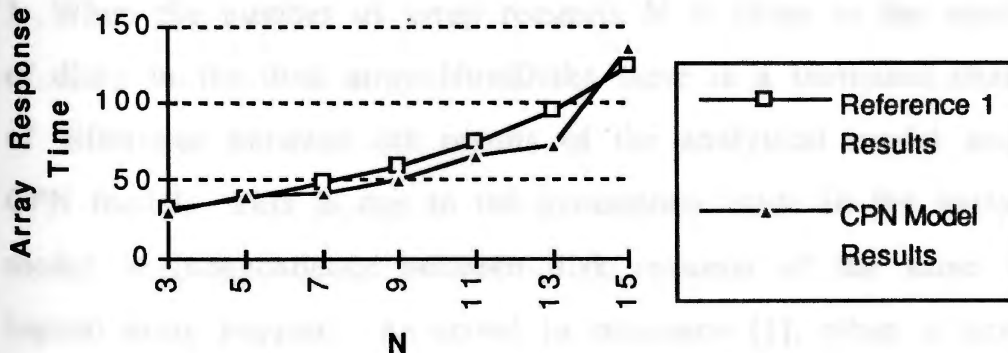


Figure 5

In the second comparison, shown in figure 5, the size of the disk array request size N was varied from 3 to 15 as was done in reference [1]. The number of disks in the disk array was set to 50. As in the previous figure, the results of the CPN model closely match those produced in reference [1].

Conclusions:

For the most part the results produced by the CPN model closely match those produced by the analytical model in reference [1], especially at low system utilization. It is generally accepted that a model should estimate response times at low to medium loads within 15% of the actual system. In both figures, the CPN model's results for low to medium load were within 15% of the analytical model's results. The main areas of difference occur during the higher utilization of the IN and/or disk array subsystems. In particular, the areas of high load are in figure 4 when there are few disks and in figure 5 when N is large. The discrepancies are due to the following:

1. When the number of array requests N is close to the number of disks in the disk array $NumDisks$ there is an increased chance of difference between the results of the analytical model and the CPN model. This is due to the assumption made in the analytical model of independence between disk requests of the same logical array request. As stated in reference [1], when a large proportion of the disks is being accessed by the same array request then there is more parallelism within each array request. This parallelism makes the individual disk requests of

the same array request more dependent upon each other because they are less likely to collide with each other than if all disk requests are assigned randomly as in the model reference [1]. As disk requests collide, their array response time can increase greatly as one of disk requests must wait until the other request completes before it can access the disk. While the effects of this are minimal at low to medium system loads where few disk collisions occur, at high system loads the analytical model will have many more collisions than the CPN model. This problem was noted in reference [1].

2. When one or more of the subsystems is highly utilized there is more chance of error in the CPN model's results. When one of the subsystems becomes a bottleneck, it can cause the array response time to vary greatly from one array request to the next. In the CPN model it would take significant simulation time for these varied response times to average out to a consistent value. Since the amount of time for simulation was limited, the areas of high system utilization will have a greater amount of error in the CPN model results than when the system utilization is low. This is most apparent in figure 5 when N is greater than 11. At this point the array response times do not have a smooth curve shape as desired. Therefore this portion of the CPN data is most suspect to error.

Together these are the reasons for differences between the results of the CPN model and those of the analytical model in reference [1].

Overall, the CPN model appears to adequately model the operation of the system of interest, especially at low to medium load.

The last chapter shows that the CPN model accurately estimates the response time of a disk array to various system loads. In this chapter, some of the assumptions made in reference [1] will be investigated and a performance evaluation will be done using the CPN model.

3.1: A Study of the Effects of Varying the Number of Buses in the Interconnection Network

In the other disk array model's studied, the effects of the interconnection network (IN) on the overall system performance were ignored. Therefore, the first assumption investigated will be to vary the number of buses in a single stage IN to determine how this affects the system's performance. The second assumption investigated will be to vary the size of data accessed by each disk request. This will be used to study the effects of various methods of ensuring data integrity in disk array upon the system's performance.

To make the CPN model consistent with those used in reference [8] the following assumptions were made: -

1. The disk's service time now included the Data Access time as defined in reference [9, 1 and 4]. The Service Time calculation was defined as in Section 2.2 of this paper.

Chapter 5:

Analysis Using the CPN Model

The last chapter shows that the CPN model accurately estimates the response time of a disk array to various system loads. In this chapter, some of the assumptions made in reference [1] will be investigated and a performance evaluation will be done using the CPN model.

5.1 A Study of the Effects of Varying the Number of Buses in the Interconnection Network

In the other disk array model's studied, the effects of the interconnection network (IN) on the overall system performance were ignored. Therefore, the first assumption investigated will be to vary the number of buses in a single stage IN to determine how this affects the system's performance. The second assumption investigated will be to vary the size of data accessed by each disk request. This will be used to study the effects of various methods of ensuring data integrity in disk array upon the system's performance.

To make the CPN model consistent with those used in reference [4] the following assumptions were made:

1. The disk's Service Time now included the Data Access time as defined in references [2, 3 and 4]. The Service Time calculation was defined as in Section 3.2 of this paper.

2. To study the delay effects of the IN on the array response time, the bus's Transfer Time T_t was associated with the TRANSFER DAR TO MEMORY and TRANSFER DATA TO CPU transitions in the Bus page of the CPN model. Therefore, T_t was not included in the ServiceTime calculation as it was in the last chapter. The data size assumed was 1 byte for each disk request and 4 kbytes for each subblock of data transferred to the processors. The bus transfer rate was assumed to be 0.6023 ms/kbyte. The delay assigned to each transfer was calculated as in section 3.2. Therefore, the delay associated with the TRANSFER DAR TO MEMORY transition was 0.0006023 ms and the delay associated with the TRANSFER DATA TO CPU transition was 2.4092 ms.

Note that while this model simulates the processing of a read disk access only, it also accurately maps the functionality for a system which performs a 4kbyte write with a 1byte acknowledge.

3. The system configuration is as follows:

$P = 10$ processors

$N = 10$ disk requests per array request

NumDisks = 50 disks

4. The number of buses was varied from 1 to 30 in order to study the buses' impact upon the system's performance. It was originally intended to simulate through a full cross-bar

configuration where there is a bus present to connect each processor to each disk, which requires 500 buses, but simulation showed that the array response time was stable when the number of buses was greater than 10. Therefore, the addition of more buses would not bring any value to the study.

Disk Array Response Time vs Number of Buses

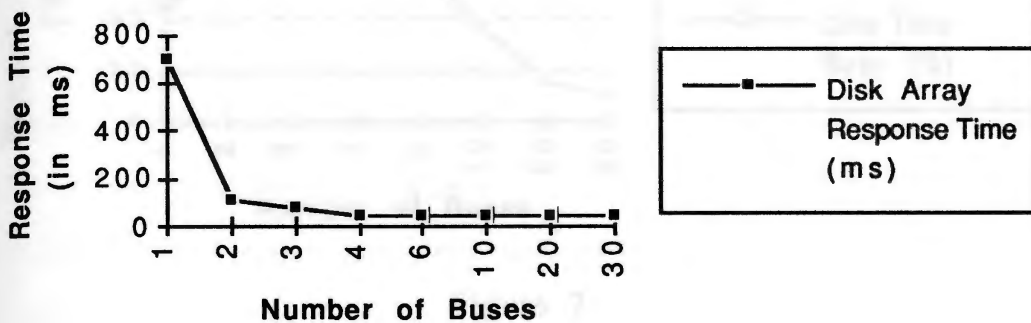


Figure 6

Figure 6 shows the array response time versus the number of buses in the system. As is shown in this figure, the disk array response time increases dramatically when the number of buses is two or less. This could be due to either the loading on the IN or the disk array. To determine which subsystem is the bottleneck, figure 7 shows the bus utilization versus number of buses and it also shows the disk utilization versus the number of buses. These figures show that for the cases where number of buses is less than three, the bus utilization is large and the disk utilization decreases. Over the rest of the range, the disk utilization is fairly constant. This indicates that when the IN utilization is very large, the IN can delay

communication to and from the disk array enough to cause the disk utilization to drop. Because the area where the IN utilization is high in figure 7 coincides with the area where the array response time is large in figure 6, the limiting factor for this case is the IN.

Utilization vs Number of Buses

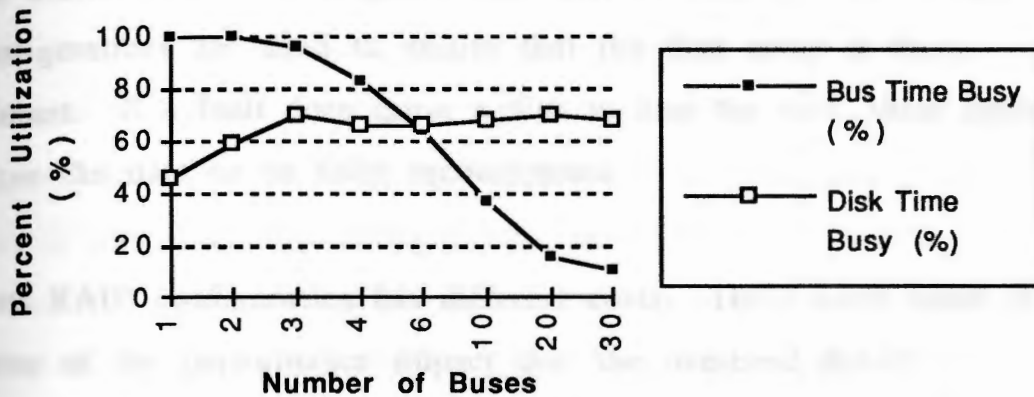


Figure 7

In conclusion, the bus system can severely limit the performance of the disk array when the number of buses is small. In contrast, once the number of buses reaches a certain point, more than 4 in this case, adding more buses does not significantly alter the I/O subsystem's performance. Therefore, a system designer must ensure that there are enough buses to prevent the IN from limiting the system performance while not including too many buses in order to minimize the cost of the system.

5.2 A Study of the Effects of Various Disk Data Integrity Methods and Subblock Size on System Performance

In the second analysis problem, the effects of the overhead induced by various Redundant Array of Inexpensive Disks (RAID) data integrity schemes and a new method proposed in reference [4] on the overall disk array response time will be studied. The RAID configurations are used to ensure that the disk array is fault tolerant. If a fault does cause a disk to lose the data, these methods allow the data to be fully reconstructed.

Each RAID configuration has different costs. These costs come in terms of the performance impact that the overhead RAID processing incurs upon the total disk array performance. The costs are also monetary as each RAID configuration requires additional disks in order to perform the specific RAID algorithm. Thus the goal of the RAID disk array designer is to minimize both the costs while maintaining the disk array's fault tolerance.

In reference [4], it was noted that the overhead caused by the RAID configurations has the most impact when the disk accesses are for small sized data. This is because for small accesses the amount of time used to transfer the data across the IN is much less than the amount of time required to access the data on the disk. This imbalance results in a bottleneck in the disk array. Because the disk array is already much slower than the rest of the system, the impact of this bottleneck can be great.

The case which best exemplifies the overhead induced by small disk accesses is the one in which the small accesses are Read-Modify-Write accesses. This type of access requires more accesses between the disk and disk controller than a simple Read or Write access. The transfers between the disk controller and the disk do not use the IN, but instead are handled by a bus which is inside the disk array. It is assumed that there is only one bus between the disk controller and the disks. Therefore, each of the transfers between the disk controller and a disk must occur sequentially. This is the worst case scenario possible because the service time for a Read-Modify-Write request will be the sum of the service times for each of the several accesses required between the disk controller and the disks. This assumption is consistent with reference [4].

The overhead incurred is different for each RAID method because each method causes a different amount of additional Read and Writes between the disk controller and the disk to perform the actions to ensure the data consistency. In reference [4], four different data integrity configurations were presented. They are non-redundant disk array, RAID Level 1, RAID Level 5 and a new scheme called Parity Logging. The details of each configuration will be discussed in the following paragraphs.

The first configuration was the standard, non-redundant disk array configuration where no data backup occurs. Each Read-Modify-Write operation requires a Read from the disk, the data is updated

by the disk controller and then the new data is written back to the disk. There is no additional overhead associated with Read-Modify-Write operations. It was included to provide a point of reference to be used for comparison with the other disk array configurations.

The second method was RAID Level 1 in which a second disk array was added which contains a copy of all the data sent to the first array. This method is often called "disk mirroring". For each Read-Modify-Write operation the data is read from the primary disk, updated, and then written to both the primary disk and its mirror disk. Therefore the performance overhead incurred is an additional write to the second disk. Because the performance overhead is not great, the main disadvantage to this method is the cost of a complete second disk array.

The next configuration is a RAID Level 5. In this method, a single disk is added to the primary disk array. This disk maintains parity information about the data on the primary array to ensure that data can be reconstructed. For each Read-Modify-Write access, the data must be read from and written to the disk array and in addition the matching data on the parity disk is read from and written to the parity disk. Thus the overhead incurred is an additional read and write for the updating of the parity disk. In the case of a small access to a disk array the overhead for a RAID Level 5 system can impose a significant system impact.

The method proposed in reference [4] is called the Parity Logging method. The method proposed is similar to the RAID level 5 scheme except that a parity and a logging disk are added to the disk array. Instead of directly writing the parity data to the parity disk, it has a buffer in RAM which holds the amount of parity information that can be stored in a disk track. When this buffer is full of parity information then this buffer is written to a track on the logging disk. This continues until the logging disk is full of parity information, at which time all the data on the parity disk and logging disk is read into memory, the parity data is updated and then the new disk full of parity data is written back to the parity disk. Therefore, if each data transfer is the one block and there are X data blocks per track and Y cylinders per disk then once every X accesses there is an additional track access and every $X*Y$ accesses there are 3 full disk accesses. Depending upon the block, track and disk sizes, the overhead induced by this method can be quite small compared to the RAID configurations while only adding two disks to the disk array.

In reference [4], it is stated that the impact of small accesses is greatest on Read-Modify-Write accesses to the disk and then proceeds by presenting the worst case scenario where all the accesses are small. To do performance analysis of a system it would be helpful to see the overhead caused by each data integrity method for more than one data access size. In this study it was assumed that the size of the data accessed will have two possible types: a small access which performs a Read-Modify-Write on a

single data block and a large access which reads a whole track from a single disk. The proportion of the large to small accesses will be altered to study the effects of the large and small accesses on the system performance.

The disk array configuration was the same as those used in references [3] and [4] for an IBM Lightning drive as follows:

numCyl = 949 cylinders per disk, 14 tracks per cylinder, 48 data blocks per track, 512 bytes per block, 13.9ms for time of full disk rotation, cylinder seek time = 2ms, avgSeek (block) = 12.6ms, minSeek (block) = 2ms, maxSeek (block) = 25ms, and there were 22 disks in the disk array

- The SeekTime is calculated as in section 3.2:
- Because the amount of time which is required to actually access the data on the disk varies significantly for small and large disk accesses and to be consistent with reference [4], the Data Access time is entered into the model to help accurately portray the disk array's performance. The Data Access Time (T_a) was defined as in section 3.2. Thus:

$$T_{ab} = \text{Block access time} = 0.289853333 \text{ ms}$$

$$T_{at} = \text{Track access time} = 13.9 \text{ ms}$$

$$T_{ad} = \text{Disk Access time} = 13191.1 \text{ ms}$$

In addition, the CPN model had the following system parameters:

10 processors; the processor's Think Time was exponentially distributed with mean of 100 ms; and 1 disk request per array request (the size of the array request is not an important parameter in this particular study).

It was assumed that there was enough buses in the system so that the IN is not a bottleneck for the I/O subsystem. In addition,

the bus's Transfer Time T_t is different for large and small accesses and was calculated as in section 3.2 which produced:

$$\text{Track Transfer Time} = T_{tt} = 13.65 \text{ ms}$$

$$\text{Block Transfer Time} = T_{tb} = 0.284 \text{ ms}$$

Because the IN was assumed that there were enough buses in the system, the bus's transfer time was included into the service time calculation in the Access Data in Disk transition.

Disk Array Response Time vs Proportion of Small Accesses (%)

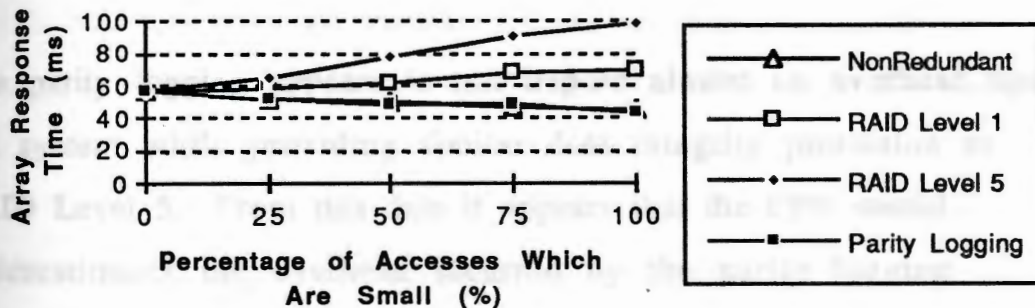


Figure 8

The disk array response times of each of the data integrity configurations are shown in figure 8. As the figure shows, the effects of the small write problem talked about in reference [4] is more prevalent when there are more small accesses than large accesses. When most of the accesses are small, the overhead effects are greatest. As the percentage of large accesses increases, the

difference between the disk array configurations is reduced. This continues until the case where all the accesses are large when there is no difference between the various disk array configurations.

As expected the RAID Level 5 response imposes the largest amount of overhead. In addition, its impact is greatest in the case where there is all small writes. At this point it more than doubles the disk array response time of the non-redundant disk array.

The RAID Level 1 imposes much less overhead than RAID Level 5 while still maintaining complete data integrity. The main problem with RAID Level 1 is that two complete disk arrays are required which can be costly.

The parity logging appears to not impose almost no overhead upon the system while providing similar data integrity protection as RAID Level 5. From this data it appears that the CPN model underestimates the overhead incurred by the parity logging methodology. As stated in reference [4], the expected overhead was to be around 25% of the disk response time. It did not appear from the simulation data that the parity full disk transfers occurred. The parity disk updates account for a large portion of the overhead in this scheme. Therefore, unless this transfer occurs the CPN model will underestimate the response time for this model. Because the model appears to be correct, the way to increase the likelihood of getting the disk accesses to occur is to run the simulation for longer periods.

The complexity of these system configurations was less than the complexity of the models presented earlier in this paper. This was mainly due to reducing the number of disks and setting the array size N to 1. Because the model is simpler, the simulation could proceed much more quickly than earlier models. Therefore, the simulations for this particular performance analysis was run for twice as much simulation time as the simulation runs in the validation section of this paper. This leads to more simulation data which produces more reliable results. This can be observed in figure 8 as the data series for each disk data integrity configuration appears to be nearly linear as expected. However the fact that the Parity Logging results are less than expected indicates that there still is some error in the results. Therefore, the longer the simulation run and the less complex the model is, the more accurate the results of the simulation.

Small writes are prevalent in many applications. Small accesses can impose a severe performance penalty for certain disk array data integrity configurations, in particular RAID Level 5. Therefore a system designer must balance the performance degradation brought on by the data integrity configuration, the proportion of small accesses to large ones, the system's data integrity needs and the cost constraints of the system.

Chapter 6:

Conclusions

This paper presents a Colored Petri Net simulation model which emulates a system comprised of a multiprocessor subsystem connected across an interconnection network to a disk array I/O subsystem. The following is a listing of the assumptions which governed the model, grouped by subsystem:

Processors:

- the number of processors can vary
- the think time of the processors can vary
- the number of disk requests in the array request can vary
- the size of data requested for each disk request can vary
- the disks generate array requests which can fork into several disk requests.
- once all the disk requests which belong to the same array have all been handled, they all join back together to complete the array request cycle.

IN:

- the size of the IN can vary
- the configuration of the IN is a single stage
- the delay across the IN can vary with the size of the data crossing it
- The data crossing the IN can be processed in a non-FIFO fashion due to contention for resources between the buses entering and the buses leaving the processors.

Disks:

- the service time of the disks can vary
- the disks in the array can service
- the number of disks can vary

The Colored Petri Net model presented is very flexible and allows many of the system parameters to be altered. For example if a constant think time for the processors was desired, only a minimal change to the model would be required. This allows the CPN model to overcome most of the limitations of analytical models which are brought on by the simplifying assumptions required to develop the state equations of the analytical model. In addition, this allows the user to model in detail only the portions of the model which are pertinent to the study. For example, the whole interconnection network page of the model could be eliminated if it was not pertinent.

This model can be used to do performance analysis's of systems which conform to the basic system architecture and can be characterized in a functional or procedural fashion. It can be used to validate analytical models such as ones presented in references [1] and [4].

While this model can estimate system performance on systems which have much larger state spaces than generally is possible with Petri Nets, its main limitation is still the complexity of the state space. If the model is very complex, then if the model can be

simulated at all, it must be performed on a high performance computer platform. To ensure that the model will run, the model complexity must be minimized.

In addition, the amount of time it takes to perform a simulation is a function of the complexity of the system's state space. To ensure accurate simulation results, the simulation time must be maximized.

In conclusion, the model developed emulates the system described and is flexible enough to emulate many different system configurations. This system can produce data about the service time of an array request, the utilization of the interconnection network and the disk array. The outputs of this model have been satisfactorily validated against the results produced in other studies of similar system over a range of all workloads. This model can be used to characterize a multiprocessing, disk array system at most levels of detail required and in the areas of interest specified in the assumptions above. The price of this flexibility is increased modeling and simulation time over analytical models. Thus, the user of this model or this modeling tool must carefully balance the amount of detail in the model required to produce useful results against the complexity of that model.

Appendix A:

The CPN model contains 4 CPN model pages: One for the processors and the generation of the disk requests, one for the interconnection network (IN), one for the disk array and one which contains the color, variable and function descriptions. These pages are logically connected and therefore act as though the model is on one page. The model was separated into these pages so that the model would be more understandable.

The separation of the model into pages also gives the user the flexibility to remove a whole subsystem's functionality from the model to simplify the model when the subsystem is not needed. In this model the interconnection network page of the model could be removed but only if it is assumed that the interconnection network is large and fast enough not to impose any limitation on the rest of the system. This simplification of the model would lead to faster simulation runs due to the smaller state space of the model.

```

input cpustate;
output (mar,newcpustate);

action
let
val disk = ran'DISK();
val limit = 0.75;
val process = 4; (* disk Type*)
val proc = GenSize(limit,process);
val dar={
  CPUId =#Cpu cpustate,
  DARId = #Id cpustate,
  Element = 0,
  Disk = disk,
  N = 0,
  Process= proc};
val newcpustate={
  Cpu=#Cpu cpustate,
  Id = #Id cpustate +1,
  StartIdle = time ()};
in
(dar,newcpustate)

```

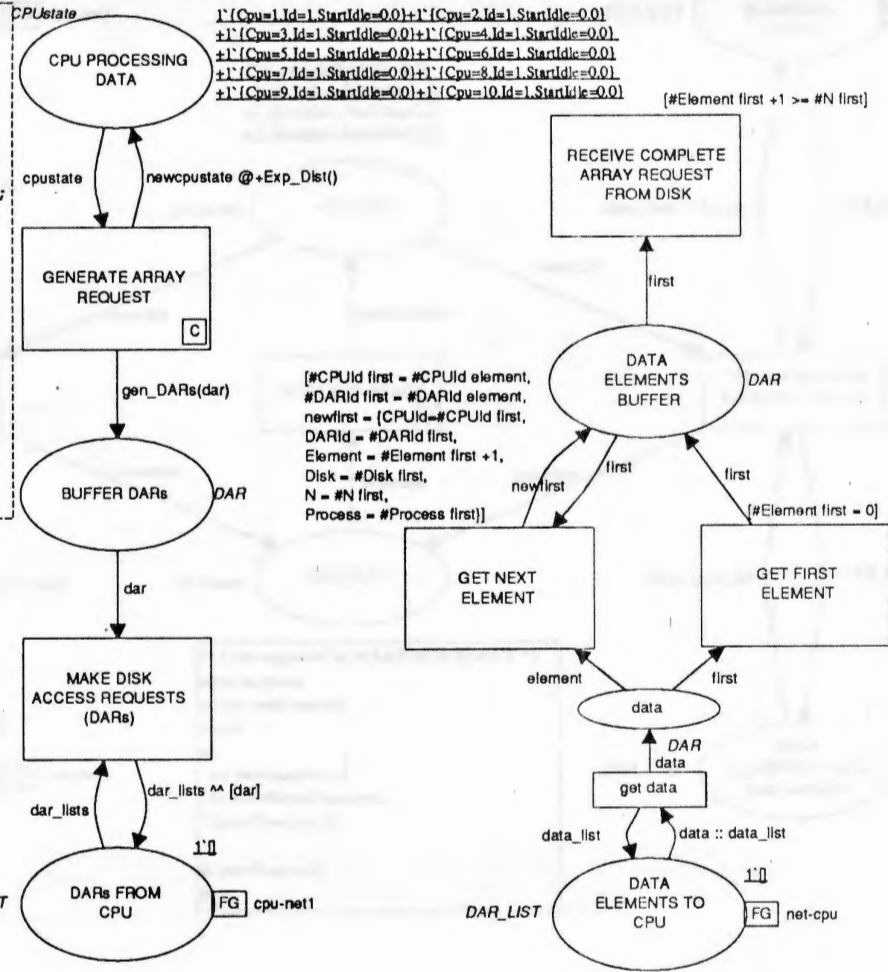


Figure A.1 Processor page of CPN Model

Figure A.2 Interconnection Network page of CPN Model

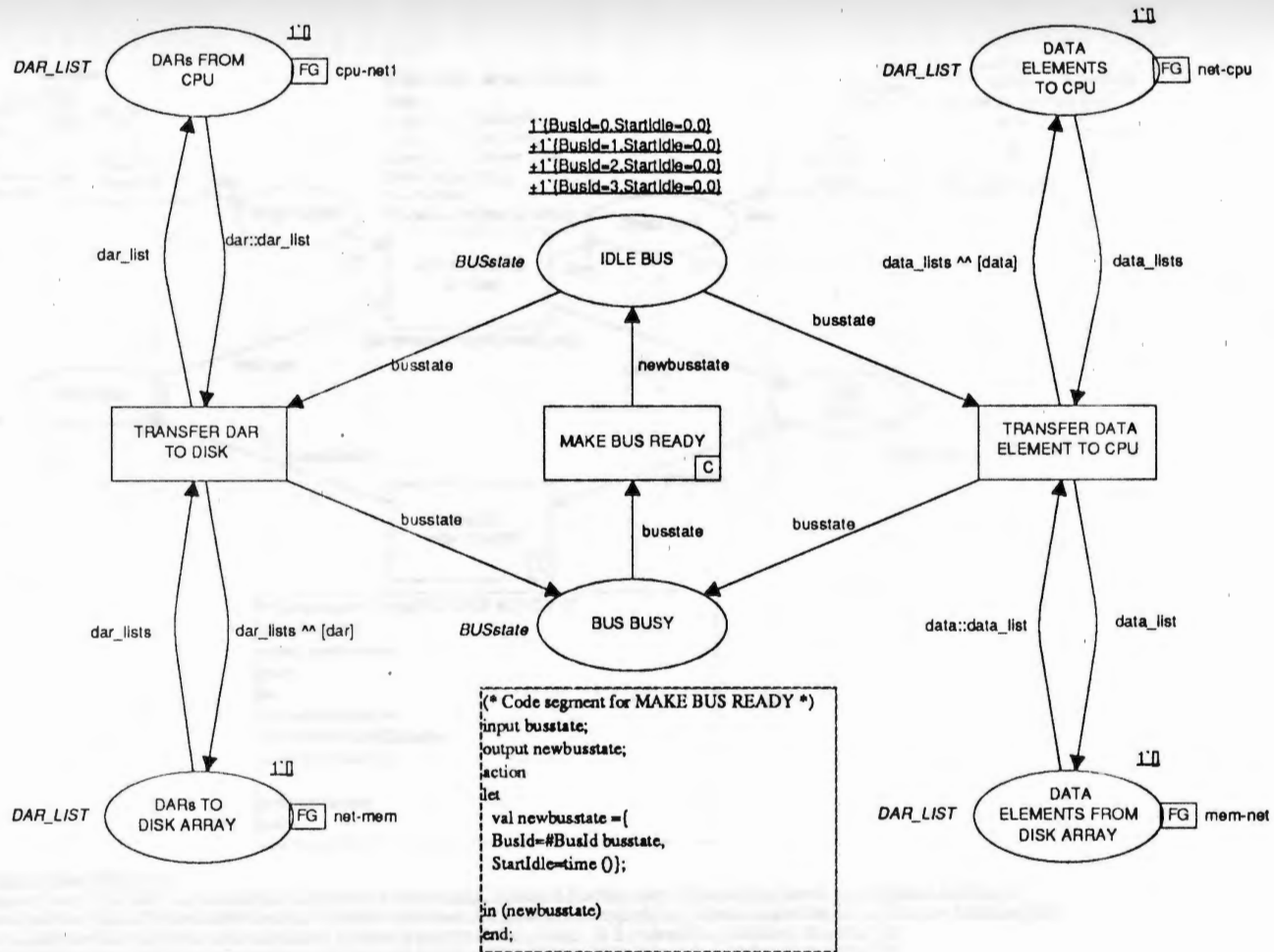
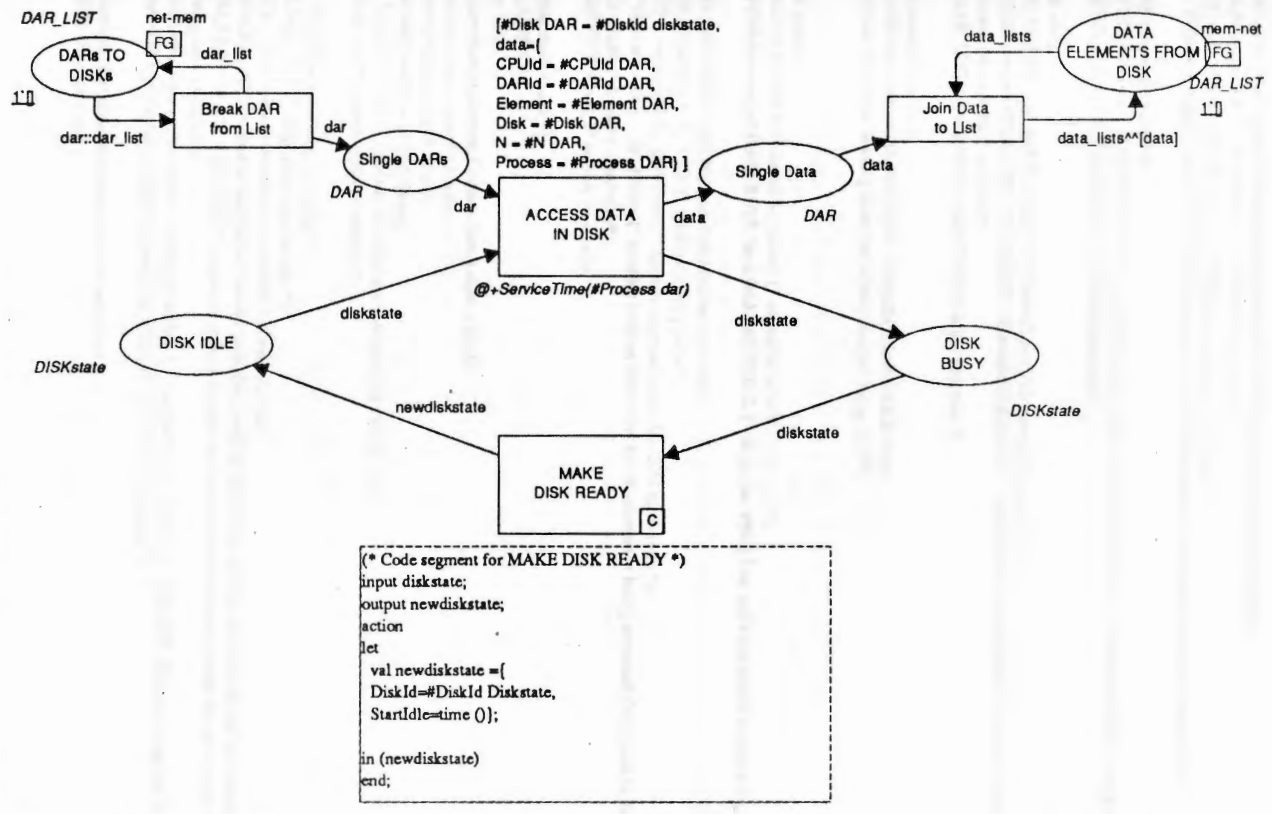


Figure A.3 Disk Array page of CPN Model



(* Initial State for Place DISK IDLE *)
 1*(DiskId=0.StartIdle=0.0)+1*(DiskId=1.StartIdle=0.0)+1*(DiskId=2.StartIdle=0.0)+1*(DiskId=3.StartIdle=0.0)+1*(DiskId=4.StartIdle=0.0)+1*(DiskId=5.StartIdle=0.0)
 +1*(DiskId=6.StartIdle=0.0)+1*(DiskId=7.StartIdle=0.0)+1*(DiskId=8.StartIdle=0.0)+1*(DiskId=9.StartIdle=0.0)+1*(DiskId=10.StartIdle=0.0)+1*(DiskId=11.StartIdle=0.0)
 +1*(DiskId=12.StartIdle=0.0)+1*(DiskId=13.StartIdle=0.0)+1*(DiskId=14.StartIdle=0.0)+1*(DiskId=15.StartIdle=0.0)+1*(DiskId=16.StartIdle=0.0)
 +1*(DiskId=17.StartIdle=0.0)+1*(DiskId=18.StartIdle=0.0)+1*(DiskId=19.StartIdle=0.0)+1*(DiskId=20.StartIdle=0.0)+1*(DiskId=21.StartIdle=0.0)


```

val Time = time; (* allows the current time to be accessed from CPN regions *)

(*constant declarations *)
val NumCPU = 10; (* allows for 128 different CPUs *)
val NumDAR = 10000; (* 10000 DARs allowed per CPU *)
val Length = 49; (* this is the length of a vector, *)
val NumDISK = 22; (* allows 50 disk *)
val NumBUS = 50; (* number of buses possible *)

(*CPU state colors *)
color STARTIDLE = real; (* holds the time that a CPU,BUS or disk starts being unused *)
color CPU = int with 1 .. NumCPU; (* CPU identifier*)
color ID = int with 1 .. NumDAR; (* disk access request identifier*)
color CPUstate = record Cpu:CPU * Id:ID * StartIdle:STARTIDLE timed; (* represents the state of the CPU *)

(* DAR generation colors *)
color PROCESS = int; (* holds which what type of process caused this DAR to be sent: maps to the RAID configuration in this model *)
color DISK = int with 0..NumDISK-1; (* I/O disk identifier *)

(*DAR colors *)
color ELEMENT = int with 0..Length-1; (* identifies which vector element it is. *)
color DAR = record CPUId:CPU * DARId:ID * Element:ELEMENT * Disk:DISK * N:ELEMENT * Process:PROCESS timed;
(*represents information in a DAR *)
color DAR_LIST = list DAR; (* holds DARs in a list structure *)

(* Disk state colors *)
color DISKstate = record DiskId:DISK * StartIdle:STARTIDLE timed;
(* Identifies which disk is being used and when it started being idle *)

(* BUS colors *)
color BUS = int with 0..NumBUS-1 timed; (* identifies which bus is used *)
color BUSstate = record BusId:BUS * StartIdle:STARTIDLE; (* Identifies which bus and what time it started being idle*)

color X = real with 0.0 .. 949.0; (* random number holder *)
color TR = real with 0.0 .. 14.0; (* rotation latency color*)
color XD = real with 0.0 .. 0.999999; (* random number holder for CPU think time *)
color TVD = int with 1 .. 23322624; (* random number holder for which element is being accessed. Only used in Parity logging model*)
color TT = real; (* transfer time color*)
color SERVICE = real; (* service time color*)

(* variable declarations *)
var n:ELEMENT;
var disk: DISK;
var dar,newdar,data,element,first,newfirst,y,new_y:DAR;
var cpushate,newcpushate: CPUstate;
var busstate,newbusstate: BUSstate;
var diskstate,newdiskstate: DISKstate;
var dar_list,dar_lists,data_list,data_lists,element_list,first_list : DAR_LIST;
var x,x1,x2 : X; (* random number holders *)

var Tr : TR; (*rotational latency holder *)
var xd,hlimit : XD; (* CPU think time holder *)
var Tib,Tu : TT; (* time to transfer a block of data across the bus *)
var Tab,Tat,Tad : TT; (* time to read/write a block, track,disk worth of data from the disk and send to the I/O controller < CPU *)
var A0,A1,A2,A3,A4,A5 : SERVICE; (* variables to hold the seek and rotation and read/write times for the various update schemes *)
var process,proc : PROCESS;
(* identifies what type of I/O system is being modeled: 1 = nonredundant, 2=mirror, 3=RAID5, 4= parity logging *)
var access : TVD; (* random number representing which access of TVD accesses *)

(* this function determines the size of the data requested *)
fun GenSize (limit,proc)=

```

Figure A.4 Declaration page of CPN Model

```

let
  val h = ranXD0;
  val GenSize = (if h > limit then 0 else proc);
in (GenSize)
end;

(* this function generates the seek time associated with a disk access *)
fun SeekTime ()=
let
  val x1 = ranX0;
  val x2 = ranX0;
  val x = abs(x1-x2);
  val Tr = ranTR0;
  val SeekTime= (0.4761*sqrt(x)+(0.0088*x)+2.0+Tr);
(* the above line generates the Service Time for a read and write access of a disk drive *)
in (SeekTime)
end;

(* this function generates the Service time for a disk access: including seek time, data access time and overhead for each RAID level *)
fun ServiceTime(process)=
let
  val Ttb = 512.0 / 1800.0;
  val Tt = 48.0 * Ttb;
  val Tab = 13.9/48.0;
  val Tat = 13.9;
  val Tad = 949.0 * 13.9;
  val access = ranTVD0;
  val A0 = SeekTime() + Tat + Tt; (* service time if large block *)
  val A1 = SeekTime() + SeekTime() + (2.0*Tab); (* read/write time of block: All types have this delay *)
  val A2 = (if process = 2 then (SeekTime() + Tab) else 0.0); (* mirror overhead *)
  val A3 = (if process = 3 then (SeekTime() + SeekTime() + Tab + Tab) else 0.0); (* RAID 5 overhead *)
  val A4 = (if process = 4 then (if (access mod 48 = 0) then (SeekTime() + Tat) else 0.0) else 0.0);
(* parity logging track write overhead *)
  val A5 = (if process = 4 then (if (access = 23322624) then (3.0*(SeekTime() + Tad)) else 0.0) else 0.0);
(* parity logging disk (2read + write) overhead *)
  val ServiceTime= (if process = 0 then A0 else (A1 + A2 + A3 + A4 + A5 + Ttb)); (* total service time for an access *)
in (ServiceTime)
end;

(* this function generates all the disk access tokens for each array request token it receives *)
fun gen_DARs(y) =
let
  val new_y = {
    CPUid = #CPUid y,
    DARId = #DARId y,
    Element = #Element y + 1,
    Disk = ((#Disk y + 1) mod NumDISK),
    N = #N y,
    Process = #Process y};
in
  if (#Element new_y) < ((#N new_y))
  then 1'y + gen_DARs(new_y)
  else 1'y
end;

(* this function calculates the think time associated with a processor based on an exponential distribution *)
fun Exp_Dist ()=
let
  val xd = ranXD0;

```

Figure A.4 Declaration page of CPN Model (cont.)

Appendix B:

Colored Petri Nets

This Appendix contains 2 parts, the first section describes the main concepts behind Colored Petri Nets and the second section presents a short overview of the functionality of Colored Petri Nets (CPNs).

Petri Nets have been used in performance studies of systems in many cases, the results of which show that Petri Nets are useful in systems that are not too complex [5,6,7]. A traditional Petri Net (PN) is a graphical and mathematical model which can be used to describe and study information processing systems that can be characterized as being distributed, concurrent, asynchronous, time varying, nondeterministic and/or stochastic. As a graphical tool, PNs can be applied to almost any application which can be described graphically like a flow diagram or state diagrams. In addition, when simulating the user can observe tokens flow through the model as they simulate the dynamic and concurrent activities of the system. As a mathematical tool, there is a mathematical formalism associated with PNs which completely defines what a PN is and how it behaves. Although PNs are generally represented as a directed graph, a PN is actually a mathematical object that exists independently of any physical representation. The actual implementation of a PN model is a state matrix which describes the set of possible states in that model. As a mathematical tool it is also possible to set up state equations, algebraic expression and other mathematical models governing the behavior of the system.[5].

A Colored Petri Net (CPN) is the type of PN used in this study. The CPN tool's main strength is its ability to study applications of higher complexity than is generally possible with traditional PN tools. A CPN differs from traditional PNs in the following ways: A CPN has the added ability to declare data types, hereafter called colors; it provides many modeling capabilities which simplify the modeling process; and most drastically, it does not offer the ability to perform the mathematical operations on the state matrix of the system that a mathematically formal PN tool would. In traditional PNs only a single data type can be handled by a node. Thus additional nodes would be required to handle each different data type. In CPNs multiple data types can flow through a single node which reduces the number of nodes in the system. In addition, Meta Software's CPN tool also provides many additional features, like simulated time and code segments which allow functionality to be entered into a model while keeping it understandable. The reason that the mathematical manipulations have not been offered for CPN is that additional functionality such as color declarations and other features makes the state space associated with a CPN is too large for matrix reduction techniques.

The main disadvantage of a CPN is that if the modeler is not careful the model can get too complex to be analyzed. This is due to the direct relationship between the CPN model's complexity and the size of the state matrix related to the model. In addition, as the state matrix gets larger the simulation model executes more slowly. Also, if a host computer platform is used which has limited RAM

available, then a model can get too large to execute. For example, even a fairly high performance personal computer, such as the Macintosh IIci with 32Mb of RAM used for this study, can quickly be overwhelmed by a model of moderate complexity. Even when the model's complexity is adequately controlled to make simulation possible, the amount of time required to simulate most models is quite large. When the model was executed on a higher performance computer, such as a Quadra 750 with 40 Mb of RAM, the simulation times were reduced by about one half. Therefore, the modeler must balance the amount of detail in the model and the host computer's ability to handle the complexity contained in the model.

Another disadvantage of the CPN tool is that the built-in charting tools, which are meant to extract data from a model, impose too much overhead to operate with this paper's model on either of the computer platforms described above. Therefore, the state of the system was saved in a text report. This report recorded any changes to the state of the system. Because this report contained much information which was not pertinent to this study, a program was written in C which extracted the relevant information. It gathered information about disk array's response time and the utilization of the bus and disk arrays. The program is included for reference in Appendix C. An example of the raw CPN data and the output of the data extraction program are presented in Appendix D.

Together, the features provided by Meta Software's Colored Petri Nets allow great flexibility for the modeler. A model can be easily

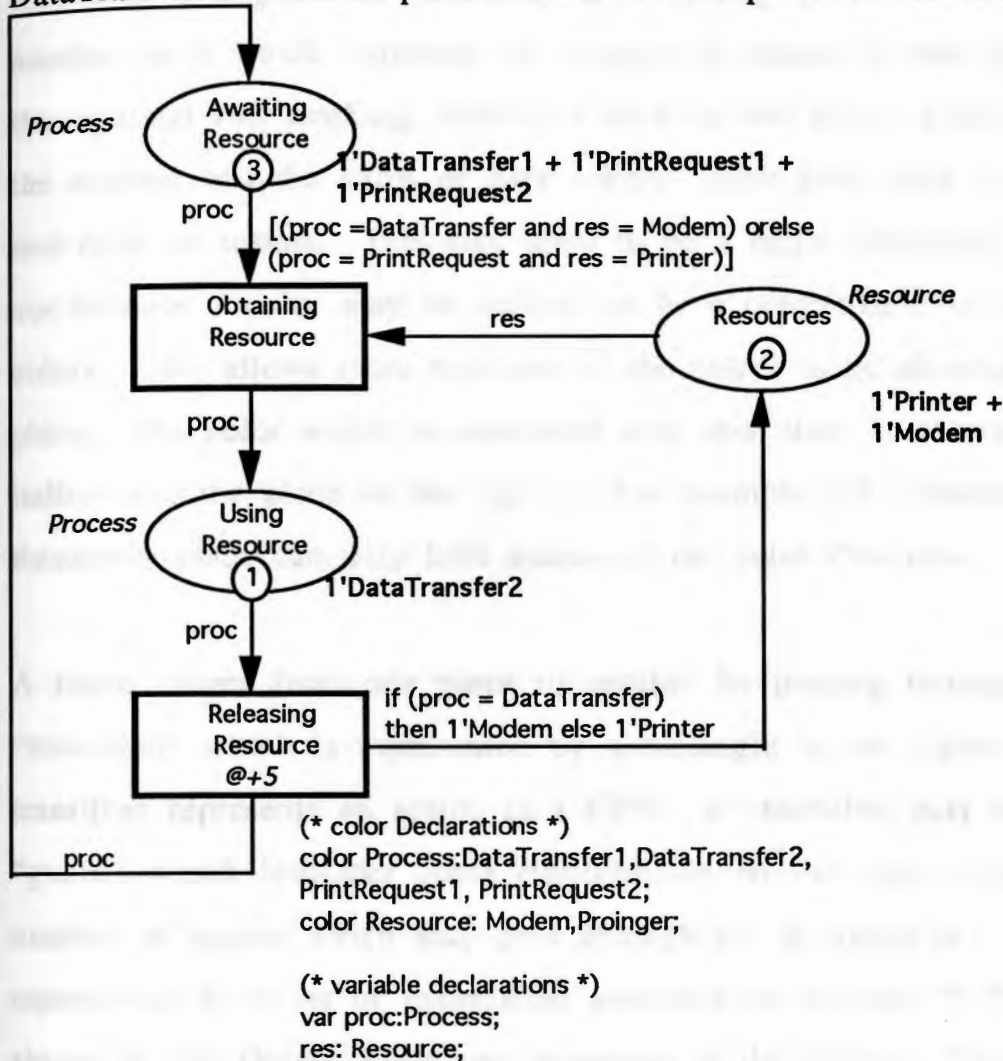
created and detail can be added to any area of the model it is required. The graphical nature of a CPN can make it easier to understand the functionality of the model and therefore does not require the audience to have much background using this tool. Because a CPN is a simulation model, it does not inherently require simplifying assumptions to be made to create a model although one has to be careful to limit the complexity of the model. The ability to declare colors and encode functionality can help limit the complexity of a model and make the model more understandable. In addition, since CPN is a mature, commercial modeling tool which has been available for several years and used on many diverse models, it is believed that the results produced by the tool are reliable.

Colored Petri Net Functionality: An Overview

In figure B.1, a simple resource contention model is shown. CPNs are made of three types of objects: A token, a place and a transition. The role that each of these play in the CPN model will be described in the following paragraphs.

A token which represents the data flowing through the model is represented by the small circle with a number inside it. These tokens are defined by data type or "color". In the figure the tokens have two possible colors which are specified in the color declaration section of the diagram: A Resource color which can have values Printer or Modem and a Process color which can have a value

DataTransfer or PrintRequest. Colors can represent more complex



A Simple Resource Contention CPN Model

Figure B.1

data types such as records or combinations of previously declared colors.

Tokens are held in “places” which are represented by ovals in the figure. The set of tokens in all the places represents the state of the model at any point in a simulation run. The number and value of

tokens in each place or "marking" is shown by the circle with the number in it which indicates the number of tokens in that place and the optional full marking, shown in bold by the place, which shows the number and the value of each token. Each place may hold only one color of tokens. This may seem to be a major limitation but it is not because a color may be defined to be a combination of other colors. This allows more than one of the colors to be allowed in a place. The color which is associated with this place is shown in italics near the place in the figure. For example the Awaiting Resources place can only hold tokens of the color *Process*.

A token moves from one place to another by passing through a "transition" which is represented by a rectangle in the figure. A transition represents an action in a CPN. A transition may have a "guard" which indicates some requirements on the type, value or number of tokens which may pass through it. A guard is represented by a set of expressions enclosed in brackets "[]" as shown in the Obtain Resources transition in the figure. The guard for this transition requires that the values of tokens coming into the transition match before allowing them to pass. Note that the "res" token is consumed by this transition. A guard may also determine what the output of the transition will be. For example a token could be generated and assigned a value based upon the value of the token entering the transition. Thus a transition may change the value or type of a token as a token leaves it.

A transition can fire only when all the input requirements and guard requirements are met. An example of an input requirement is that all places which go to the transition must have tokens which enable the transition. A transition which is enabled to fire is drawn with a thicker border as both are in the figure. If more than one combination of input tokens have enabled a transition to fire, then either a set is chosen at random or the guard determines which are selected. Thus in the Obtain Resources transition, there are three possible markings which fire this transition: (DataTransfer1, Modem), (PrintRequest1, Printer) or (PrintRequest2, Printer).

A CPN transition can pass more than one marking through a transition at a time if there are enough resources to allow it. For example, the Obtain Resources transition could allow both the (DataTransfer1, Modem), (PrintRequest1, Printer) tokens to pass through it at the same simulation step. This allows the simulation to advance using fewer simulation steps which reduces the overhead which is incurred by each simulation step.

An arc, represented by an arrow, is the connection between a place and transition. It can have a set of requirements which are similar to the guard associated with a transition. These requirements could specify a token color or a required number of tokens which may pass across it. For example in the arc leading from the Release Resources transition to the Resource Pool place the value of the token which will go across it is determined by the value of the token which enters the transition.

A CPN also has the optional ability to simulate time. The ability to represent time allows quantitative results to be produced by the model. The method for implementing time is that a delay can be associated with any transition or arc. Thus some transitions could be required to take time and others would take no time. This allows functionality to be included, such as data extraction, which does not have an effect on time associated with a token. The format for a this is: $@ + \textit{delaytime}$ where @ indicates the current time and *delaytime* could either be a constant or conditional numeric assignment. In figure B.1, the Release Resource transition has a delay time associated with it. Therefore when a token passes through this transition, it is not available for use until the time advances to (Current Time + 5 time units).

A code segment is a function which can be associated with an arc or transition which can be much like a procedure in a computer program. A code segment allows more complex operations to be performed than would easily be possible using a guard or arc inscription. A code segment is written in the CPN variant of the ML language. An example of a code segment can be found in Appendix A associated with the Generate Array Request transition on the processor page. This code segment builds a token of the "record" color which represents an array request.

Functions can also be defined to perform operations which are done repeatedly. A function is written in the CPN variant of the ML

language. A function could be associated an arc inscription, a time delay, a guard or a code segment. The functions are generally defined in the Declaration Node page. One example is the Exp_Dist function which calculates the exponential distribution for the processor think time. This function is on the arc between the CPU Processing Data place and the Generate Start Address and Stride transition. The body of the function is located on the Declaration Node page.

```

[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
[28]
[29]
[30]
[31]
[32]
[33]
[34]
[35]
[36]
[37]
[38]
[39]
[40]
[41]
[42]
[43]
[44]
[45]
[46]
[47]
[48]
[49]
[50]
[51]
[52]
[53]
[54]
[55]
[56]
[57]
[58]
[59]
[60]
[61]
[62]
[63]
[64]
[65]
[66]
[67]
[68]
[69]
[70]
[71]
[72]
[73]
[74]
[75]
[76]
[77]
[78]
[79]
[80]
[81]
[82]
[83]
[84]
[85]
[86]
[87]
[88]
[89]
[90]
[91]
[92]
[93]
[94]
[95]
[96]
[97]
[98]
[99]
[100]

```

Appendix C:

(* The program to extract the chart data from raw CPN data *)

```
#include <stdio.h>
#define MAXLINE 300 /*defines the maximum line length */
main ()

{
    double GetNumFl();
    FILE *in, *out, *out2, *out3, *out4;
    char infile[20];
    int i,j;
    char ch;
    char Gen;
    char Rec;
    char Join;
    int line;
    int NumInt;
    int numspace;
    double TimeNow;
    double TimeStart;
    int CPU;
    int N;
    int Disk;
    int Element;
    int Process;
    double CPUStart[11][100]; /* holds the amount of time a CPU was
idle */
    char text[300];
    double CPUWait; /* time a CPU has waited */
    double CPUWaitPerEl; /* time that CPU waited per element */
    double TotalCPUWait; /* time that all CPUs have waited */
    int TotalN; /* total number of elements sent */
    int ReqNum; /* number of data requests performed for a CPU thus
far */
    char Direction; /* which direction data went on bus */
    float Busy; /* how long Bus was held busy */
    char Bus; /* whether a Bus data line was read */
    float BusWait; /* idle time for bus */
    int BusId; /* which bus it is */
    int DiskId; /* which memory bank is it */
```

```

float DiskBusy; /* what the total access time of a bank was for a
certain access */
float DiskIdle[50]; /* holds the amount of time a bank was idle for
an access */
char DiskA,DiskB; /* whether the current line is 'Access' or
'Make'(Disk), respectively */

```

```

printf ("Enter the input file name. \n");
scanf ("%s",infile);
if (((in = fopen(infile,"r")) != NULL) && ((out = fopen("CPU.txt","a"))
!= NULL) &&((out2 = fopen("BUS.txt","a")) != NULL) && ((out3 =
fopen("Mem.txt","a")) != NULL)&& ((out4 = fopen("Join.txt","a")) !=
NULL));
{
line =0;
Gen= 'f';
Rec='f';
Bus='f';
DiskA='f';
DiskB='f';
Join='f';
Direction=' ';
Busy=0.0;
EraseArray(text,MAXLINE);
i=0;
j=0;
TotalCPUWait=0;
TotalN=0;
while ((ch=getc(in)) != EOF)
{
text[i] = ch;
if (line==0)
{
if (ch == '\n')
{
line =1;
EraseArray(text,MAXLINE);
numspace = 0;
i=-1;
}
}
if (line ==1)

```

```

{
if (ch == '\n' && numspace != 0)
{
EraseArray(text,MAXLINE);
numspace = 0;
i=-1;
line=2;
ch='\0';
}
if (ch == ' ')
{
numspace = numspace +1;
if (numspace == 4)
j=i+1;
}
if (numspace == 4 && text[j] == 'G' && text[j+2] == 'N' && Gen !=
't')
{
TimeNow = GetNumFl(3,text);
Gen = 't';
}
if (numspace == 4 && text[j] == 'R' &&Rec != 't')
{
TimeNow = GetNumFl(3,text);
Rec = 't';
}
if (numspace == 4 && text[j] == 'J' &&Join != 't')
{
TimeNow = GetNumFl(3,text);
Join = 't';
}
if (numspace == 4 && text[j] == 'd' &&Bus != 't')
{
TimeNow = GetNumFl(3,text);
Bus = 't';
Busy = 0.4; /* data going to CPU*/
}
if (numspace == 4 && text[j] == 'T' &&Bus != 't')
{
TimeNow = GetNumFl(3,text);
Bus = 't';
Busy = 0.0001; /* DAR going to memory bank*/
}
}

```

```

if (numspace == 4 && text[j] == 'A' && DiskA != 't')
{
    TimeNow = GetNumFl(3,text);
    DiskA = 't';
}
if (numspace == 4 && text[j] == 'M' && text[j+8] == 'M' && DiskA
!= 't')
{
    TimeNow = GetNumFl(3,text);
    DiskB = 't';
}
} /*end of line =1 processing */
if (line == 2 && Gen == 't' && ch == '\n')
{
    CPU = GetNumInt(9,text);
    ReqNum = GetNumInt(11,text);
    CPUStart[CPU][ReqNum] = TimeNow;
    Gen = 'f';
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;
    numspace=0;
}
if (line == 2 && Rec == 't' && ch == '\n')
{
    CPU = GetNumInt(9,text);
    ReqNum = GetNumInt(11,text);
    Disk = GetNumInt(15,text);
    N = GetNumInt(17,text);
    Process = GetNumInt(19,text);
    CPUWait = TimeNow - CPUStart[CPU][ReqNum];
    CPUWaitPerEl = CPUWait / N;
    TotalCPUWait= TotalCPUWait + CPUWait;
    TotalN= TotalN + N;
    fprintf(out,"%f %d %f %d %d %d %d
\n",TimeNow,CPU,CPUWait,
    Process,N,Disk,ReqNum);
    Rec = 'f';
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;

```

```

    numspace=0;
}
if (line == 2 && Join == 't' && ch == '\n')
{
    CPU = GetNumInt(9,text);
    ReqNum = GetNumInt(11,text);
    Element = GetNumInt(13,text);
    Disk = GetNumInt(15,text);
    N = GetNumInt(17,text) +1;
    CPUWait = TimeNow - CPUStart[CPU][ReqNum];
    fprintf(out4,"%f %d %f %d %d %d %d
\n",TimeNow,CPU,CPUWait,Element,N,Disk,ReqNum);
    Join ='f';
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;
    numspace=0;
}
if (line == 2 && Bus == 't' && ch == '\n')
{
    BusId = GetNumInt(9,text);
    TimeStart = GetNumFl(11,text);
    BusWait = TimeNow - TimeStart;
    fprintf(out2,"%f %d %f %f \n",TimeNow,BusId,BusWait,Busy);
    Bus ='f';
    Busy = 0.0;
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;
    numspace=0;
}
if (line == 2 && DiskA == 't' && ch == '\n')
{
    DiskId = GetNumInt(15,text);
    TimeStart = GetNumFl(39,text);
    DiskIdle[DiskId] = TimeNow - TimeStart;
    DiskA ='f';
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;

```



```

    numspace=0;
}
if (line == 2 && DiskB == 't' && ch == '\n')
{
    DiskId = GetNumInt(9,text);
    TimeStart = GetNumFl(11,text);
    DiskBusy = TimeNow - TimeStart - DiskIdle[DiskId];
    fprintf(out3,"%f %d %f %f
\n",TimeNow,DiskId,DiskIdle[DiskId],DiskBusy);
    DiskB ='f';
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    line =1;
    numspace=0;
}
if (line ==2 && Rec != 't' && ch == '\n')
{
    line =1;
    EraseArray(text,MAXLINE);
    i=-1;
    j=0;
    numspace=0;
}
i++;
}
    fclose(in);
    fclose(out);
    fclose(out2);
    fclose(out3);
    fclose(out4);
}
}

```

/* GetNumFl function */

```

double GetNumFl(spaces,arrIn)
int spaces;
char arrIn[];
{
    int m,n,o;
    double NumFl;
    char greater; /* greater than zero flag */

```

```

double tens; /* power of 10 holder */

/* enter in to the beginning of the number */
m=0; /* 'arrIn' array pointer */
n=0; /* number of spaces counter */
tens = 10.0;
greater = 't';
NumFl = 0.0;
while (n < spaces)
{
    if (arrIn[m] == ' ')
    {
        n++;
    }
    m++;
}
/* convert the array to a number */
while (arrIn[m] != ' ' && arrIn[m] >= 48 && arrIn[m] <= 57)
{
    if (arrIn[m] != '.' && greater == 't')
    {
        NumFl = (NumFl * 10.0) +(arrIn[m]-48.0);
        m++;
    }
    if (arrIn[m] == '.')
    {
        greater='f';
        m++;
        o=1;
    }
    if (greater == 'f' && o<7)
    {
        NumFl = NumFl + ((arrIn[m]-48.0) / tens);
        tens = tens * 10.0;
        m++;
        o++;
    }
    if (greater == 'f' && o >= 7)
        m++;
}
return(NumFl);
}

```

```
/* GetNumInt function */
```

```
GetNumInt(spaces, arrIn)
```

```
int spaces;  
char arrIn[];  
{  
    int m,n,o;  
    int NumInt;  
  
    /* enter in to the beginning of the number */  
    m=0; /* 'arrIn' array pointer */  
    n=0; /* number of spaces counter */  
    NumInt=0;  
    while (n < spaces)  
    {  
        if (arrIn[m] == ' ')  
        {  
            n++;  
        }  
        m++;  
    }  
    /* convert the array to a number */  
    while (arrIn[m] != ' ' && arrIn[m] >= 48 && arrIn[m] <= 57)  
    {  
        NumInt = (NumInt * 10) +(arrIn[m]-48);  
        m++;  
    }  
    return(NumInt);  
}
```

```
/* EraseArray function */
```

```
EraseArray(arr,length)
```

```
char arr[];  
int length;  
{  
    int k;  
    for(k=0;k<length;k++)  
    {  
        arr[k]='\0';  
    }  
}
```

Appendix D:

Raw CPN Data (excerps)

Simulation Report

1 A @ 0.0 GENERATE@(1:CPU#1)

{ cpushate = {Cpu = 10,Id = 1,StartIdle = 0.0},mar = {CPUId = 10,DARId = 1,Element = 0,Disk = 45,N = 10,Process = 1},newcpushate = {Cpu = 10,Id = 2,StartIdle = 0.0}}

2 A @ 0.0 GENERATE@(1:CPU#1)

{ cpushate = {Cpu = 9,Id = 1,StartIdle = 0.0},mar = {CPUId = 9,DARId = 1,Element = 0,Disk = 59,N = 10,Process = 1},newcpushate = {Cpu = 9,Id = 2,StartIdle = 0.0}}

3 A @ 0.0 MAKE@(1:CPU#1)

{ mar = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},mar_lists = []}

...

7 A @ 0.0 TRANSFER@(1:BUS#3)

{ busstate = {BusId = 9,StartIdle = 0.0},mar = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},mar_list = [{CPUId = 10,DARId = 1,Element = 8,Disk = 53,N = 10,Process = 1}],mar_lists = []}

...

14 A @ 0.0 ACCESS@(1:MEMORY#5)

{ data = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},mar = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},diskstate = {DiskId = 54,StartIdle = 0.0}}

...

726 A @ 20.3440527255964 Join@(1:MEMORY#5)

{ data = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},data_lists = []}

727 A @ 20.3440527255964 MAKE@(1:MEMORY#5)

{ diskstate = {DiskId = 54,StartIdle = 0.0},newdiskstate = {DiskId = 54,StartIdle = 20.3440527255964}}

728 A @ 20.3440527255964 ACCESS@(1:MEMORY#5)

{ data = {CPUId = 2,DARId = 1,Element = 1,Disk = 54,N = 10,Process = 1},mar = {CPUId = 2,DARId = 1,Element = 1,Disk = 54,N = 10,Process = 1},diskstate = {DiskId = 54,StartIdle = 20.3440527255964}}

729 A @ 20.3440527255964 d@(1:BUS#3)

{ busstate = {BusId = 3,StartIdle = 17.2029297094067},data = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N = 10,Process = 1},data_list = [],data_lists = []}

```

730 A @ 20.3440527255964 MAKE@(1:BUS#3)
    { busstate = {BusId = 3,StartIdle =
17.2029297094067},newbusstate = {BusId = 3,StartIdle =
20.3440527255964} }
731 A @ 20.3440527255964 q@(1:CPU#1)
    { data = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N =
10,Process = 1},data_list = []}
...
872 A @ 26.7412681559373 h@(1:CPU#1)
    { first = {CPUId = 10,DARId = 1,Element = 0,Disk = 45,N =
10,Process = 1} }
873 A @ 26.7412681559373 GET@(1:CPU#1)
    { element = {CPUId = 10,DARId = 1,Element = 1,Disk = 46,N =
10,Process = 1},first = {CPUId = 10,DARId = 1,Element = 0,Disk =
45,N = 10,Process = 1},newfirst = {CPUId = 10,DARId = 1,Element =
1,Disk = 45,N = 10,Process = 1} }
874 A @ 26.7412681559373 GET@(1:CPU#1)
    { element = {CPUId = 10,DARId = 1,Element = 9,Disk = 54,N =
10,Process = 1},first = {CPUId = 10,DARId = 1,Element = 1,Disk =
45,N = 10,Process = 1},newfirst = {CPUId = 10,DARId = 1,Element =
2,Disk = 45,N = 10,Process = 1} }
...
888 A @ 26.8354260254611 RECEIVE@(1:CPU#1)
    { first = {CPUId = 10,DARId = 1,Element = 9,Disk = 45,N =
10,Process = 1} }

```

When this raw CPN data is run through Strip.c it results in four files: CPU.txt which contains the response times for each array request (from processor back to processor), Bus.txt which contains the utilization data of each bus in the interconnection network and Mem.txt which contains the utilization of each disk in the disk array.

The data produced is best shown in chart. The following is the data produced from the complete file above from the CPU.txt file:

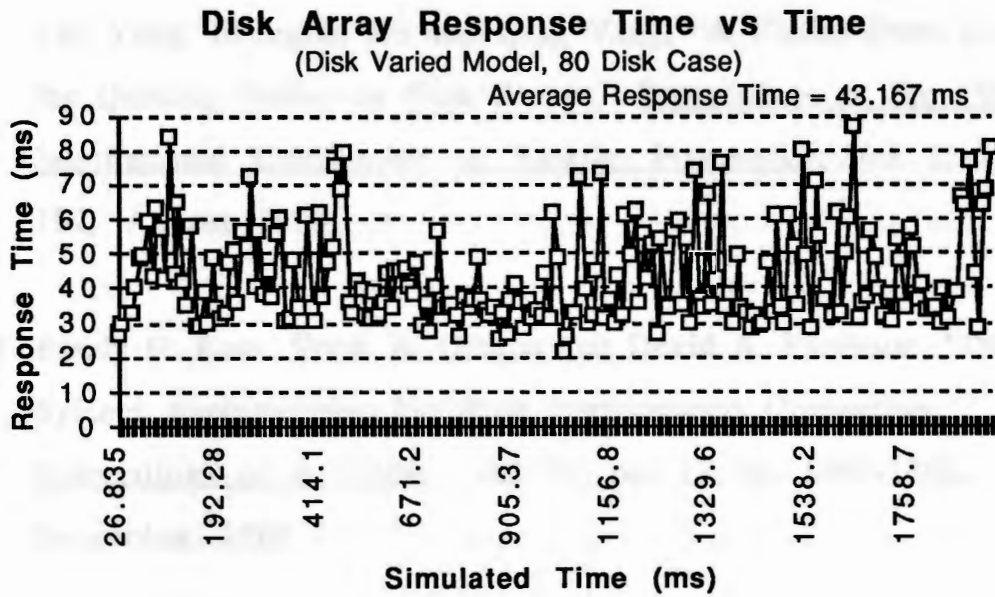


Figure D.1 Typical Results From a CPN Simulation Run

The average value from this chart is then used as a data point on one of charts used to characterize the system's performance over a range of system parameters. For example, the above chart's data produces the D=80 data point in Figure 4 of this report.

References

- [1] Tao Yang, Shengbin Hu and Qing Yang, "A Closed-Form Formula for Queuing Delays in Disk Arrays," Proceedings of the 1994 International Conference on Parallel Processing, Vol. 2, pp. 189-192, August 1994.
- [2] Randy H. Katz, Garth A. Gibson and David A. Patterson, "Disk System Architectures for High Performance Computing," Proceedings of the IEEE, Vol. 77, No. 12, pp. 1842-1858, December 1989.
- [3] Edward K. Lee and Randy H. Katz, "An Analytic Performance Model of Disk Arrays," Performance Evaluation Review: Proceeding of ACM SIGMETRICS, Vol. 21, No. 1, pp. 98-102, June 1993.
- [4] Daniel Stodolsky, Garth Gibson and Mark Holland, "Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays," IEEE Computer Architecture News, Vol. 21, No. 2, pp. 64-75, May 1993.
- [5] Tadao Murata, "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, Vol. 77, No. 4, pp. 541-580, April 1989.
- [6] Mark A. Holliday and Mary K. Vernon, "Exact Performance Estimates for Multiprocessor Memory and Bus Interference,"

IEEE Transactions on Computers, Vol. C-36, No. 1, pp. 76-85,
January 1987.

- [7] Mary K. Vernon and Mark A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," ACM Performance Evaluation Review: Proceedings of ACM SIGMETRICS, pp. 9-17, 1986.
- [8] Prithviraj Banerjee and A. L. Narasimha Reddy, "An Evaluation of Multiple-Disk I/O Systems", IEEE Transactions on Computers, Vol. 38, No. 12, pp. 1680-1690, December 1989.

Bibliography

Banerjee, Prithviraj and A. L. Narasimha Reddy and , “An Evaluation of Multiple-Disk I/O Systems”, IEEE Transactions on Computers, Vol. 38, No. 12, pp. 1680-1690, December 1989.

Holliday, Mark A. and Mary K. Vernon, “Exact Performance Estimates for Multiprocessor Memory and Bus Interference,” IEEE Transactions on Computers, Vol. C-36, No. 1, pp. 76-85, January 1987.

Katz, Randy H., Garth A. Gibson and David A. Patterson, “Disk System Architectures for High Performance Computing,” Proceedings of the IEEE, Vol. 77, No. 12, pp. 1842-1858, December 1989.

Lee, Edward K. and Randy H. Katz, “An Analytic Performance Model of Disk Arrays,” ACM Performance Evaluation Review: Proceedings of ACM SIGMETRICS, Vol. 21, No. 1, pp. 98 - 109, June 1993.

Murata, Tadao, “Petri Nets: Properties, Analysis and Applications,” Proceedings of the IEEE, Vol. 77, No. 4, pp. 541-580, April 1989.

Stodolsky, Daniel, Garth Gibson and Mark Holland, “Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays,” Computer Architecture News, Vol. 21, No. 2, pp. 64-75, May 1993.

Vernon, Mary K. and Mark A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," ACM Performance Evaluation Review: Proceedings of ACM SIGMETRICS, pp. 9-17, 1986.

Yang, Tao, Shengbin Hu and Qing Yang, "A Closed-Form Formula for Queuing Delays in Disk Arrays," Proceedings of the 1994 International Conference on Parallel Processing, Vol. 2, pp. 189 - 192, August 1994.