A SIMULATION MODEL FOR BUS BASED SHARED MEMORY MULTIPROCESSOR SYSTEMS

by

İlker BEKMEZCİ

B.S. in CmpE, Boğaziçi University, 1994

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfilment of
the requirements for the degree of
Master of Science
In
Computer Engineering

Boğaziçi University
1998

A SIMULATION MODEL FOR BUS BASED SHARED MEMORY MULTIPROCESSOR SYSTEMS

APPROVED BY:

Prof. Dr. Oğuz TOSUN .....
(Thesis Supervisor)

Prof. Dr. Ömer CERİD .....

Asst. Prof. Dr. Can ÖZTURAN .....

DATE OF APPROVAL 05.05.1998

ACKNOWLEDGMENTS

I would like to thank to all who have helped during my study. I especially would like to thank my thesis supervisor Prof. Dr. Oğuz TOSUN for his help and guidance in this study. In addition to this, I want to thank Asst. Prof. Can ÖZTURAN for his carefull review of the thesis and valuable comments. I acknowledge the academic permission given to me by the Turkish Air Force.

This study is dedicated to my wife and parents for their encouragement throughout the study.

ABSTRACT

Simulation modelling is one of the most common methods to predict the performance of multiprocessor computer systems. In this study, a new simulation model is developed and a simulator based on this model is implemented to predict the performance of the bus based shared memory multiprocessor systems.

The main inputs of the simulator are representation of the architecture and workload parameters. There are several proposed algorithms to process these inputs, including path finding, mailbox location, and memory units clustering algorithms. A number of protocols are also devised and implemented in the simulator.

The verification and validation studies are very important for a simulator. The verification study is realised by the help of several analytical models, which are available in the literature. The validation is realised by the performance results of the TOMP prototype.

In the last part of the thesis, several sample runs are provided to analyse and compare certain architectures under various workload conditions.

ÖZET

Benzeşim modellemesi, çok işlemcili bilgisayar sistemlerinin başarım modellenmesinde kullanılan en yaygın metodlardan biridir. Bu çalışmada, yol tabanlı bellek paylaşımlı çok işlemcili sistemlerin başarım ölçümü amacıyla yeni bir benzeşim modeli geliştirilmiş ve bu modele dayalı bir simülatör gerçekleştirilmiştir.

Simülatörün ana girdileri, bilgisayar sisteminin mimarisi ve işyükü parametreleridir. Simülatörde, bu girdileri işleyen birçok algoritma vardır. Ana algoritmalar, yol bulma, posta kutusu yeri ve hafıza ünitelerini sınıflama algoritmalarıdır. Bunlara ek olarak, simülatörde birçok protokol mevcuttur.

Geçerlilik ve doğrulama çalışmaları da, benzeşim modeli geliştirmede çok önemli çalışmalardır. Benzeşimin doğrulanması, literatürde bulunan birçok analitik model ile gerçekleştirilmiştir. Geçerlilik ise TOMP prototipinin performans sonuçları ile gerçekleştirilmiştir.

Çalışmanın sonunda, birçok örnek bulunmaktadır. Bazı mimarilerin farklı işyükü koşullarındaki performans değerleri sunulmuş ve karşılaştırılmıştır.

TABLE OF CONTENTS

1. INTRODUCTION **1**

1.1. ORGANISATION OF THE THESIS 2

2. PARALLEL PROCESSING COMPUTER SYSTEMS **3**

2.1 THE NEXT REVOLUTION IN COMPUTING 3

2.1.1 Modern Prehistory 4

2.1.2 The Age of Dinosaurs 4

2.1.3 The Second Wave 4

2.1.4 The Third Wave 5

2.1.5. The Parallel Wave 6

2.2. FLYNN'S HARDWARE TAXONOMY 6

2.2.1 SISD 7

2.2.2 SIMD 7

2.2.3 MIMD 7

2.3. PARALLEL PROCESSING APPLICATIONS 8

3. PERFORMANCE EVALUATION OF MULTIPROCESSOR SYSTEM AND SIMULATION MODELS **10**

3.1. PERFORMANCE MEASUREMENT TECHNIQUES 11

3.2. PERFORMANCE EVALUATION MODELS 12

3.2.1.System Description 13

3.2.2. System Abstraction and Model Description 14

3.2.3. Data Collection 16

3.2.4. The Selection of Method of Analysis 16

3.2.4.1. Analytical Model 17

3.2.4.2. Simulation Model 18

3.2.5 Simulation Program Development 19

3.2.6. Verification 20

3.2.7. Validation 21

4. ARCHITECTURAL MODEL AND WORKLOAD MODEL OF THE SIMULATOR 22

4.1. WHY SIMULATION MODELLING? 22
4.2. THE ARCHITECTURES THAT CAN BE SIMULATED 22
4.3. GENERAL DESCRIPTION OF THE SIMULATOR 23
4.4. LEVEL OF ABSTRACTION 24
4.5. REPRESENTATION OF THE ARCHITECTURE 25
4.5.1. Supervisor Structure 25
4.5.2. Basic Elements of Multiprocessor Architecture 27
4.5.2.1. Processing Elements 27
4.5.2.2. Shared Memory Units 29
4.5.2.3. Interconnection Network 29
4.6. REPRESENTATION OF WORKLOAD 29
4.6.1. The Structure of the Tasks and Primitive Routines. 31
4.6.1.1. Completion Times of Each Sub-Task and System Load 32
4.6.1.2. Length of Tasks 34
4.6.2. Task Synchronisation Mechanism 34
4.7. PATH FINDING ALGORITHM 37
4.8. MAILBOX PROBLEM 41
4.9. MEMORY MODULE CLUSTERING AND DATA DISTRIBUTION 44
4.10. CACHE COHERENCE 48
4.11. THE STATES OF TASKS IN PROCESSORS 48
4.12. PROTOCOLS 50
4.12.1. Protocols for Bus Contention 50
4.12.2. Protocols for Task Selection 53

5. VERIFICATION OF SIMULATION MODEL 54

5.1. SINGLE BUS MULTIPROCESSOR WITH EXTERNAL COMMON MEMORY 55
5.2. MULTIPLE BUS MULTIPROCESSOR WITH EXTERNAL COMMON MEMORY 57
5.3. SINGLE BUS MULTIPROCESSOR WITH DISTRIBUTED MEMORY 58

6. VALIDATION OF SIMULATION MODEL 61

7. SIMULATION EXAMPLES 63

7.1. The Effect of Architecture and Task Graph 65
7.2. Protocols 66
7.3. The Place of Task Synchronization Mechanism 67
7.4. Data Distribution and Probability Distribution of the Memory Usage 67

8. CONCLUSION 70

REFERENCES 71

LIST OF FIGURES

Figure 3.1. Multiprocessor System performance evaluation techniques. 10

Figure 3.2. The modelling and analysis process. 12

Figure 4.1. The block diagram of the simulator. 23

Figure 4.2. Classification of supervisor structures. 26

Figure 4.3. The supervisor structure of the simulator. 27

Figure 4.4. A processing unit in a shared memory system. 28

Figure 4.5. The states of a parallel program. 30

Figure 4.6. An example of a task graph with node weights equal to 1. 30

Figure 4.7. An example of task. 32

Figure 4.8. An example of a task structure. MP or RM is added. 35

Figure 4.9. MP is at the end and RM is at the beginning of the task. 35

Figure 4.10. An example. MP and RM is in the task. 36

Figure 4.11. A complete example of tasks with two links. 36

Figure 4.12. AllPaths procedure. 38

Figure 4.13. LoadOnBus function. 39

Figure 4.14. LoadOnPath function. 39

Figure 4.15. Path Finding algorithm. 40

Figure 4.16. A part of a multiprocessor system. 40

Figure 4.17. Distance algorithm. 41

Figure 4.18. Mailbox finding algorithm. 42

Figure 4.19. An example of hierarchical bus based multiprocessor system. 44

Figure 4.20. Clustering algorithm. 45

Figure 4.21. An example of hierarchical bus based multiprocessor system. 45

Figure 4.22. State transition diagram of tasks. 49

Figure 4.23. Maximum Distance Protocol 51

Figure 4.24. Maximum Bus Efficiency Protocol. 52

Figure 4.25. Important Message Protocol. 52

Figure 5.1. An example of single bus multiprocessor with external common memory. 55

Figure 5.2. Processor utilization under different system loads 56

Figure 5.3. An example of multiple bus multiprocessor 57

Figure 5.4. Simulation and analytical results. 58

Figure 5.5. An example of single bus multiprocessor with distributed memory 59

Figure 5.6. Simulation and analytical results of single bus 59

Figure 6.1. Comparison of TOMP architecture and simulation model results. 62

Figure 7.1. Architectures used in the example analysis. 63

Figure 7.2. Task graphs used in the example work. 64

Figure 7.3. The performance data for Architecture 1 under TG1 and TG2. 65

Figure 7.4. The performance data for Architecture 2 under TG1 and TG2. 66

Figure 7.5.The performance data under the protocols of FIFO and Random. 67

Figure 7.6.The performance data under different task synchronisation mechanism. 68

Figure 7.7.The performance data under different probability distributions. 69

LIST OF SYMBOLS

G	: directed acyclic weighted task graph
V	: set of task nodes
E	: set of communication edges
C	: set of communication edges costs
$P(c_i)$	: probability of being chosen of c_i
T	: set of node computation costs
m	: number of memory module in a cluster
n_j	: j^{th} node in the task graph
c_i	: i^{th} cluster set
p_I	.: conditional probability of being chosen of c_i
$\varsigma_{i,j}$	: communication cost along n_i to n_j
γ	: number of nodes in a task graph
ε	: number of edges in a task graph
λ	: average completion time of CPU sub-tasks
μ	: average completion time of MEM sub-tasks
ρ	: system load

1. INTRODUCTION

Recent progress in VLSI technology has allowed the production of single-chip computing units with processing power comparable to that of mainframes of ten to twenty years ago. A consequence of this innovation has been the possibility of designing and implementing distributed computing systems inspired by the physical distribution and by the intrinsic parallelism of many applications.

Different goals often suggest the development of distributed computing systems with quite different structural characteristic. Multiprocessor systems are a special class of distributed computing systems that appear to represent the most promising way of obtaining the high-performance computers needed in many application fields, such as artificial intelligence, CAD, expert systems, and large-scale system simulation. Characteristic such as fault tolerance, flexibility, functional upgrading and cost effectiveness are other motivations that have spurred the realisation of multiprocessor systems. To achieve these goals, a variety of multiprocessor architectures with different design alternatives have been proposed, implemented, and made commercially available, but their relative merits are not yet fully understood. It is thus very important to develop methodologies and tools for the predictions of the performance of multiprocessor architectures, so that system designer can verify how well different alternatives suit certain given performance specifications.

There are several performance evaluation techniques for multiprocessor systems, and one of them is simulation. It is certain that developing a simulation model is a costly technique and it is harder than any alternative modelling strategies. Moreover, multiple runs are required to obtain accurate results and confidence intervals. Although simulation model has some disadvantages, because of its flexibility and representing power, many researches prefer simulation.

In this thesis, simulation model is used to evaluate the performance of "Bus Based Shared Memory Multiprocessor System". By the help of this tool, performance of many bus based systems can be investigated under different conditions.

1.1. Organisation of the Thesis

In the second chapter, the basic concepts of parallel processing are discussed. It covers classification of architectures and different applications of parallel systems. Chapter 3 is devoted to performance evaluation techniques. Especially, simulation models are explained and steps of developing a simulation model are presented. In Chapter 4, the simulation model developed in this thesis is described. Building blocks of simulation model, workload characterisation and parameters, different algorithms and protocols in this simulation model are in the scope of this chapter. Another important issue in simulation model is verification and it is provided in Chapter 5, comparing analytic models and simulation model for different architectures. Chapter 6 covers the validation of the model. TOMP prototype system is used to show the accuracy of the results of this model. In Chapter 7, multiple simulations are done varying the architecture and workload parameters and the results are compared. The last chapter, Chapter 8, states the conclusion of the thesis.

2. PARALLEL PROCESSING COMPUTER SYSTEMS

In the first computing wave, scientific and business computers were more or less identical -big and slow. This was the "prehistory of computing"; here computing had to be employed at any cost. And, even though earlier electronic computers were not very fast, they achieved speeds that easily exceeded human computers.

The second and third waves brought on mainframes, minis, and finally macros. This diversity of computing caused a number of niches to develop, which broadened and deepened the computer industry. Scientific and business computing went their separate ways, and there seemed to be a computer in just about everyone's price range.

But the original power users who pioneered computing continued to emphasize speed above all else. Single-processor supercomputers achieved unheard of speeds beyond 100 million instructions per second, and pushed hardware technology to the physical limits of chip manufacturing. But soon this trend will come to an end, because there are physical and architectural bounds, which limit the computational power that can be achieved with a single-processor system.

Nowadays, it is the time of parallel wave of computing, where performance is enhanced by using multiple processors. In this chapter, first, the history of parallel computing will be introduced. And then, taxonomy to classify different architecture will be explained.

2.1 The Next Revolution in Computing

Modern society is particularly susceptible to changes in computer technology. The insurance and banking industries were forever changed by the mainframe data processing computer; science and engineering will never be the same after the impact of minicomputers and workstations and our personal lives have been enriched by personal computers. Computers affect everyone.

To determine the next step in computing, it is a good idea to look to the past, because, like most progress in technology, computing evolves through time in an orderly fashion.

2.1.1 Modern Prehistory

The Alwac 3E computer was typical of the state of computing in 1963. It could store 32,000 numbers, each with 32 bits, and read punched paper cape at unheard of 100 Frames per second. The Alwac was less powerful than a 1980 personal computer, but it was operated by one person at a time much like a personal computer.

Early computers such as the Alwac had one major disadvantage compared which personal computers: they were expensive. Because of high hardware costs, the first generation of computers had to be shared by a lot of users to justify their cost. It would take 20 years before simple and easy-to-use machines were to reappear as inexpensive alternatives to centralized computing.

2.1.2 The Age of Dinosaurs

By 1965 the Alwac "personal computer" and its contemporaries had been pushed aside by the radically new IBM System/360 mainframe.

The IBM System/360 was the right computer in the right place at the right time. It was in harmony with the instincts of most programmers of the mid-1960s and early 1970s. It had a real operating system, multiple programming languages, and incredibly large disks capable of 10 megabytes of storage. This was the first wave of modern computing, and the world quickly jumped on the mainframe bandwagon.

The System/360 filled a room with boxes and people to run them. Its transistor circuits were reasonably fast. Power users could order magnetic core memories which up to one megabyte of 32-bit words. This machine was large enough to support many- programs in memory at the same time, even though the central processing unit had to switch from program to program.

2.1.3 The Second Wave

The mainframes of the first wave were firmly established by the late 1960s when advances in semiconductor technology made the solid state memory and integrated circuit

feasible. These advances in hardware technology spawned the minicomputer era. They were small, fast, and inexpensive enough to distribute throughout the company. Minicomputers made by DEC, Prime, and Data General led the way in defining a new kind of computing: departmental.

By the 1970s it was clear that there existed two kinds of commercial or business computing:

- centralized data processing mainframes, and
- decentralized transaction processing minicomputers.

The minis expanded the usefulness of computing into engineering, scientific and non-data processing applications.

2.1.4 The Third Wave

When personal computers were introduced in 1977 by Altair, Processor Technology, North Star, Tandy, Commodore, Apple and many others, they were largely ignored. But then the original "personal " computing idea of the 1960s was suddenly catapulted into orbit. By 1981, personal computing was becoming so pervasive that IBM entered the "billion dollar baby" market.

Personal computers enhanced the productivity of individuals, and in turn departments. Because big companies are made up of individuals, the productivity improvement of individuals using stand-alone computers was too compelling to ignore. PCs soon became pervasive.

Networks of powerful personal computers and workstations began to replace mainframes and minis by 1990. The power of the most capable "big" machine could be bought in a desktop model for one-tenth the cost. But, these individual desktop computers were soon to be connected into larger complexes of computing by networking.

One of the clear trends in computing is the gradual substitution of networks in place of central computers. These networks connect inexpensive, powerful desktop machines to one another to form unequaled computing power. Network is an early form of parallel computing.

Clearly, there is a limit to the power of a single computer. Even networking has limitations. Within the decade of the 1990s, the maximum switching speed of silicon will be reached and the rapid progress in achieving greater computing speed will level off [1].

2.1.5. The Parallel Wave

What is the next wave of computing? How can machines continue to operate faster and faster in the face of fundamental limits to the hardware? Parallel computing is the answer to both of these questions. The 1990 decade is to parallel computing what the 1980 decade was to personal computing.

Parallelism is the process of performing tasks concurrently. When many tasks can be executed in parallel, average execution time is decreased. This is the basic logic of parallel systems.

2.2. Flynn's Hardware Taxonomy

There are many different ways to organize computational structures to exploit the parallelism that exists in most current and future computer applications. Many research efforts around the world are being conducted with the purpose of determining those hardware and software organizations that are best suited for general purpose parallel processing. The availability of parallel processing is essential to achieve the wide acceptability and commercial success. At the same time, many other efforts have concentrated on speeding up the solutions of specific problems or classes of problems in special purpose systems [2].

As a result, the large number of proposed parallel processing architectures exhibit such a great diversity of combinations of common as well as unique characteristics that they are difficult to group into a neat classification scheme.

Flynn's taxonomy is one of the basic taxonomies in parallel systems[3]. This classification scheme was introduced by Michael J. Flynn [4]. It is originally proposed to classify hardware as SISD, SIMD or MIMD.

2.2.1 SISD

SISD (Single-Instruction-Single-Data) computing is the traditional single-processor. An application is run on a single processor under control of a single instruction scream (one instruction is taken from the program at a time), and each instruction operates on a single datum at a time.

SISD machines are often given the appearance of parallelism through operating system features for supporting multitasking. When equipped with time multiplexing of tasks, a fast SISD machine can support a form of concurrency, but true parallelism is not supportable. Therefore, SISD hardware is incapable of parallel computing.

2.2.2 SIMD

SIMD (Single-Instruction-Multiple-Data) seems restrictive at first, but is perhaps the most useful paradigm for massively parallel scientific computing.

In a SIMD computer, a single instruction stream is acted upon by many processing elements, in lockstep sequence. That is, one instruction counter is used to sequence through a single copy of the program. The data that is processed by each processing element differs from processor to processor. Therefore, a single program and a single control unit simultaneously act on many different collections of data.

Many scientific and engineering applications naturally fall into the SIMD paradigm, e.g., image processing, particle simulation, and finite element methods.

2.2.3 MIMD

MIMD is the most general model of parallelism. Synchronization is achieved explicitly and locally rather than through a global synchronization mechanism. This is flexible, but it also means the software is more difficult to control.

Because of the flexibility of MIMD, a variety of programming paradigms may be used. However, the overriding question is, "when should the MIMD paradigm be employed?". As a generalization, MIMD is useful when the problem allows multiple,

heterogeneous tasks to be performed at the same time. This is most likely to occur when either the number of tasks to be performed is not known ahead of time, the tasks perform different operations from one another, or both.

In fact, MIMD is general enough to encompass SIMD, because SIMD behaviour can be emulated by restricting MIMD through careful programming. However, there may be several performance penalties inherent in simulation of one form on a machine of a different form.

To complicate matters even more, MIMD machines are typically composed of SISD processors, and each processor is capable of supporting many at the "same" time. Indeed, most shared-memory multiprocessor systems such as Encore support multiple UNIX tasks on each processor, giving rise to a class of machines not covered by Flynn's taxonomy. Such hybrids of the concurrent and parallel processing worlds make very cost-effective transaction processing systems because they are able to dramatically improve response time in a multitasking operating system.

2.3. Parallel Processing Applications

Parallel processing provides many different high-speed architectures. Solving many complicated problems and implementing many algorithms are easy with the help of these architectures,. There are some specific areas that are very suitable for parallel architectures.

One of them is artificial intelligence. Artificial Intelligence is an area that can take advantage of Parallel Processing for new approaches to solve efficiently many important problems. One important question in AI applications is the actual speedup from parallelism that can be exploited in the underlying logical structure. It appears that at least in some cases, such as rule-based systems, the practically achievable speedup is quite limited, less than tenfold [5].

Another application area is computer vision. Computer vision is an application that can be implemented efficiently in different architectures, if the corresponding low level algorithms are matched to the architectures. There are many examples of this application in the literature [6].

Signal processing is another important application area in which a number of architectures have been shown to be very effective, including data flow machines, systolic arrays and the CHiP computer [7].

Parallel processing can also support discrete event simulation techniques efficiently. The main design issue is the partitioning of the problems among the processors of the system and the synchronization of the partitions, since each has its own simulation clock.

Parallel processing for Computer Aided Design (CAD) is an application area being offered by most vendors of parallel computers because of its big market and because most design automation algorithms partition well into parallel architectures.

Neural networks and optical computing studies are only some of the new technologies developed after parallel processing systems have been implemented. These new areas are also very suitable for parallel architectures.

3. PERFORMANCE EVALUATION OF MULTIPROCESSOR SYSTEM AND SIMULATION MODELS

The performance evaluation of computing system has been the object of extensive research studies since the early days of computer. Many different techniques are used to evaluate the performance of many computer systems, including multiprocessor systems.

Performance evaluation techniques can be classified into two main areas; these are measuring and modelling, respectively. Measurement techniques can be investigated in three branches. These are measurement, benchmarking, and prototyping. Simulation and analytic models are the most common modelling techniques [8]. All performance evaluation techniques are shown in a hierarchical diagram in Figure 3.1.

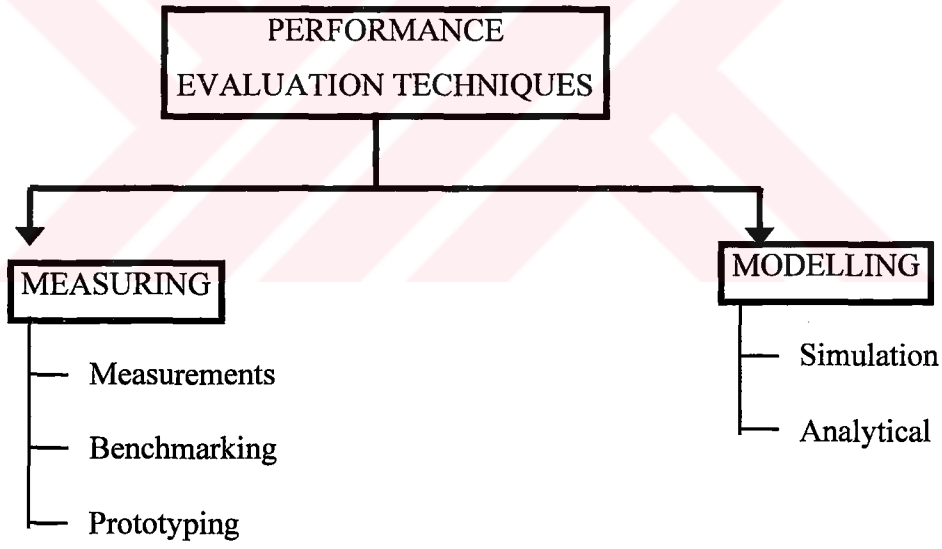


Figure 3.1. Multiprocessor System performance evaluation techniques.

3.1. Performance Measurement Techniques

Measurements are performed on a real system under real operating conditions, and thus provide very accurate performance figures with respect to the specific system under investigation, and to its workload. When the need to compare the performances of different computing systems arises, it is necessary to perform measurements, using the same workload on the different machines. The artificial workloads (selected application programs or specially built one) that are used in the case are called *benchmarks*, and the technique itself is called as *benchmarking*.

The Numerical Aerodynamic Simulation (NAS) Parallel benchmarks are good examples. One of the main aims of NAS program is to demonstrate by the year 2000 an operational computing system that can simulate an entire aerospace vehicle system within several hours of computing time. There are eight standard operations in such a program. NASA creates benchmarks to evaluate the performance of the candidate multiprocessor systems under these eight conditions. These benchmarks are embarrassingly parallel, multigrid, conjugate gradient, 3D FFT PDE, integer sort, lower-upper diagonal, scalar pentadiagonal, and block tridiagonal. The benchmarks are run on candidate multiprocessor systems and results are compared to select the best system [9].

The last technique for performance measurement is prototyping. When the performance evaluation study refers to a computing system that is not psychically available yet, it is necessary either to build a prototype or to use models. *Prototypes* are approximations of the real system (built in hardware or in software- in the latter case they are often referred to as emulators) that can be used to perform measurements, possibly with benchmark programs. TOPMP80 is an example of a prototype [10].

The design of computer systems whose specifications include performance objectives requires the preliminary evaluation of alternatives that may be only in a first stage of definition. In this case, the techniques of measurement and benchmarking must obviously discarded, since the object of measurement is not available yet. However, even prototyping is of little use, since the development of a prototype or of an emulator requires the definition of many details that may be far from being decided at this stage of the design. It is thus necessary to resort to modelling techniques that pertain to the second performance evaluation area, and that allow the designer to choose the level of abstraction of its representation of the real system.

3.2. Performance Evaluation Models

There are some certain steps and rules of developing and using a model for a multiprocessor system. The key step in this process is that of abstracting a system design into a model design. This process of abstraction is mostly based on past experience, and even the experienced modeller finds new challenges in each new system. To develop skills in abstracting from system design to model requires working with an actual system. It is best to begin with an existing system. In this way, workload characterisation and model validation data can be obtained easily.

The modelling and analysis method is outlined in Figure 3.2 [11]. The process can be divided into three phases: development, testing, and analysis.

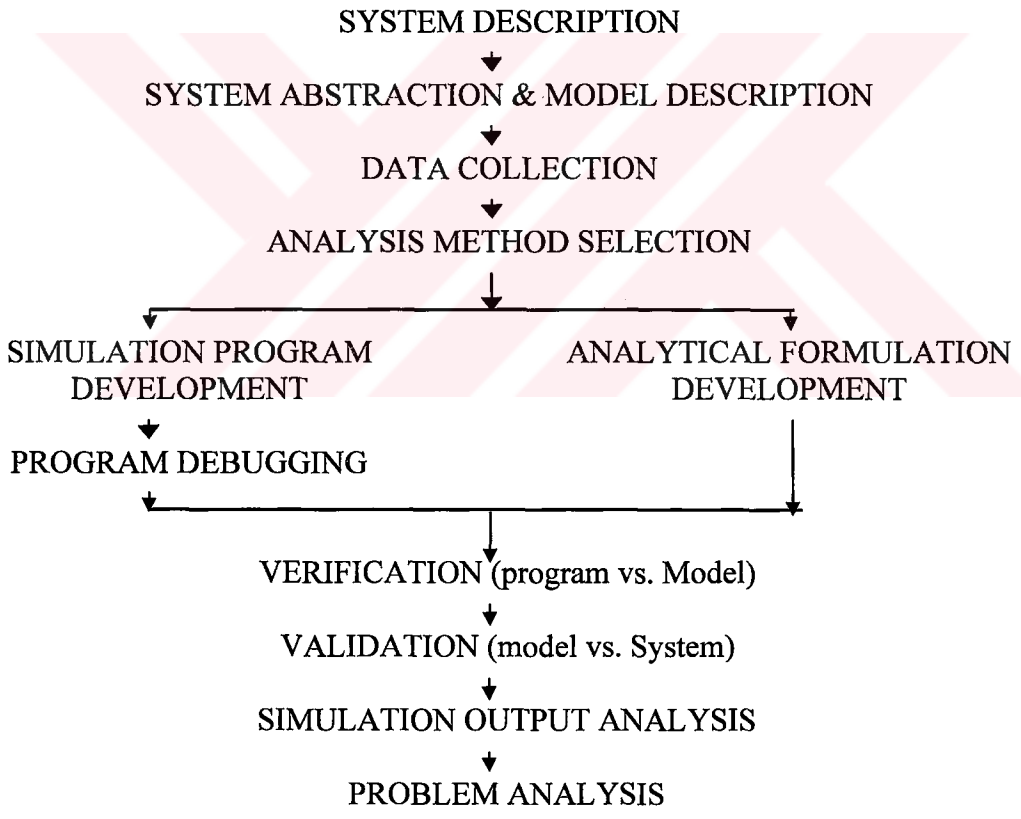


Figure 3.2. The modelling and analysis process.

The first step in development is to describe system operation from a performance viewpoint. This description then is abstracted, in accordance with the objectives of the analysis, into a model description. This specifies the facilities to be represented in the and

the operations that are involved in accomplishing this work. The level of abstraction determines the data to be collected. Next, the appropriate analysis method is chosen, and a model implementation is developed.

The testing phase comprises three steps: debugging, verification, and validation. If model gives accurate and reasonable results, it is debugged. In same cases, giving a result is a sufficient condition for a successful debugging. Verification insures that the system to evaluate the performance is indeed an implementation of a model. Validation insures that the model is a reasonable representation of the real system. The details of modelling and analysis process will be explained in the following sections.

3.2.1.System Description

It is generally assumed that the designer and modeller is the same person. When they are different persons, the modeller's first task is learning how the system works and describing its operation from a performance viewpoint; this description provides the basis for developing a model. The modeller relies on the designer to provide the knowledge needed. If the two fail to communicate, the analysis effort is, at best, a waste of time; at worst, it can result in bad design decisions. Communication problem can be both technical and inter-personal.

Effective technical communication places responsibilities on both designer and modeller. The designer has a broad view of the system: the modeller, a narrow one. However, the modeller has to learn enough about the design to determine what aspects are critical to its performance and must be included in the model. The designer and modeller are mutually responsible for the latter's education. The modeller needs to gain both a working knowledge of the overall design and a detailed understanding of the part of the system to be modelled. He has to understand this part in more detail than he plans to model it. The designer has a continuing responsibility for answering questions about design details; because the modeller's view differs from, the designer's, these questions may cut across design levels and modules. The modeller needs to explain the designer what analysis results can be expected, why particular questions are being asked, and how the answers will be used. The design may be incomplete (for reasons motivating the analysis

in the first place), and designer and modeller need to work together to develop the assumptions needed to carry out the analysis.

The knowledge the modeller gains in this learning process is an abstraction of the design, a model in its own way, and reflects a number of assumptions, some explicit, some implicit. The modeller's view of how the system operates should be documented, this system description is reviewed and any appropriate revisions are made by the designer. When the designer agrees with the description, it becomes an informal contract between designer and modeller. The designer usually will let the modeller know when design changes affect this description, and will readily accept results from models based on it. The form of this description depends on the type and scope of the system being modelled; it may be nothing more than a one-page flow diagram.

3.2.2. System Abstraction and Model Description

A model is an abstraction of a system, and represents a particular view of that system. Models frequently are described in terms of the method used to obtain performance measures: analytic model, simulation model. At this point in the modeling and analysis process, a representation of the system, which captures its essential performance-determining characteristics, is developed. This analysis should be developed systematically. Otherwise, some invalid assumptions may introduce unconsciously. All the study that should be done in this step is to describe the representation of the model.

A model description of a simple system typically takes the form of a diagram showing system resources (both hardware and software) and their interconnection, annotated to show the flow of work through the system and the operations involved, and accompanied by explanatory notes and descriptions of assumptions. It identifies decisions and timings dependent on attributes of work as well as timings dependent only on the system design. For complex systems, multiple levels of diagrams may be used to show the configuration, and flow charts or pseudo-programs used to describe processing operations. Its style depends on the design background of the modeller (hardware or software); the way in which it is developed depends on how the abstraction process is approached.

There are no formal rules for abstracting a system design into a model description. There is no simulation text which helps in this topic, and performance text which can not

offer more. Approaches differ from problem to problem (and person to person), but basically either employ synthesis or decomposition.

Synthesis. Synthesis begins at the level of the design description. To form a higher-level description, elements of the system are combined (or perhaps just ignored), and associated activities are correspondingly combined and simplified. In making each simplification, modeller needs to ask if and how the essential underlying characteristics have been preserved. In this step, some resources may be combined. It is important that each simplifying assumption should be recorded. The synthesis process may take several steps, each creating a higher level of description. When the desired level of detail is reached, all the assumptions made are reviewed and assessed their probable impact on the results of the analysis. If the net effect is in acceptable range, the assumptions are useful for the model.

Decomposition. Decomposition is the reverse of synthesis. The system initially is viewed as a single entity, its work viewed at the highest level (computer system, job; disk subsystem, request, LAN, message). Work is decomposed into its principal; this process is repeated through increasing levels of detail until the desired level is reached. In decomposition, the starting point is very general assumptions. Modeller should refine them in advancing from one level of detail to the next. Decomposition provides a better overall representation; to add details is very easy, when it is needed. In either approach, the strongest assumptions probably will be very much the same and will involve describing work.

The problem is to choose the appropriate method. Sometimes modeller do not have any chance to choose, because system allows only one of them. Large systems are best approached via decomposition. In most cases, although synthesis is available, decomposition is used. Because, it is better to begin at a high level of abstraction and add detail later than to begin with too much detail.

3.2.3. Data Collection

When development of the model description is complete, the next task is to list the model parameters that have to be specified numerically and determine their values. These parameters can be categorised as workload parameters (such as inter-arrival times, execution times, storage requirements, record types and lengths) and system parameters (usually timings for various operations, such as the memory cycle time). A parameter may have a single fixed value, or it may have to be specified in terms of the distribution of its values.

It is useful to start by determining values of system timing parameters. When modeling an existing system, it may be possible to measure parameters either directly (via hardware or software instrumentation) or indirectly (via regression analysis). When modelling a design, parameter values will have to be estimated.

Determining values of workload parameters and, in particular, specifying distributions, is the hardest part of the analysis process. Measurement and characterisation of actual system workloads can provide values directly to the analysis of existing system, and can provide a basis for estimating values for use in analysing new systems. There are several studies on this subject [12, 13].

It is difficult to carry out a workload characterisation study of a particular environment, and extremely difficult to study a range of environments. The difficulties increase with the level of detail with which the system is viewed. In undertaking a study, it is hard to find existing measurement tools. This may not be possible; even when it is the added overhead or added risk, which may limit.

While there is no substitute for the insights gained from studying actual system behaviour, blind use of measurement may create a false sense of confidence in the analysis and its results. In working with real systems, it is very hard, and frequently impossible, to demonstrate that a design performs as desired.

3.2.4. The Selection of Method of Analysis

Models can be divided into two classes: simulation models and analytical models. Analytical models describe system operations and workloads in mathematical terms.

Performance estimates are obtained by either analytical or numerical solution of the resulting mathematical model. Simulation models, instead, are computer programs in which system behaviour and workload are described by using proper algorithms. Special high-level programming languages are usually employed for the construction of these models, whose performance indices are obtained by monitoring program execution. In the following parts of this chapter these two models will be explained.

3.2.4.1. Analytical Model

In the development of an analytical model it is often necessary to use a high level of abstraction, since in order to be able to solve the model, some constraints on its structure must be accepted. In the simpler cases it is possible to obtain closed form solutions useful for studying the impact that different model parameters have on performance indices, ie. to perform a sensitivity analysis. In the more complex cases the model solution can be obtained only numerical, and the sensitivity analysis is possible only at the expense of a large number of numerical solutions, computed for different values of the model parameters. In extreme cases, the computational complexity, the storage requirements, and the numerical problems may make the solution of an analytical model more cumbersome and expensive than simulation.

A class of models that is widely used due to their limited mathematical complexity is based on the theory of stochastic processes named Markov chains [14].

A limit on the use of Markovian models of complex computer systems comes from the fact that their direct construction often requires some familiarity with the basic results of the underlying theory. Indeed, in these cases it is necessary to identify all the system states and the speeds or the probabilities with which the system moves from one state to another. This task may be particularly difficult, and ad hoc techniques may be required for its accomplishment.

A more convenient approach is that of using one of the high-level model description tools that have been proposed by the literature. The two best known such techniques are queuing networks and stochastic Petri nets [15]. These techniques allow model to be constructed in a natural way from the description of the system components and operation rules; the model is specified in a graphical form rather than in mathematical one. Petri nets

are then analysed by studying their underlying Markovian model, but the system designer need not be aware of the theory and of the methods that are necessary to obtain the model solution.

3.2.4.2. Simulation Model

Another method to develop a multiprocessor system model is simulation. In this thesis this kind of method is used to evaluate the performance of multiprocessor systems. The key step to develop and use a simulation model is that of abstracting a system design into a model.

Several simulation packages are available to develop a simulation. One of them is "smpl". This is the most common library for simulation developers [11]. However, to use a package is not a must for simulation. Many developers use a programming language to develop simulations for multiprocessor systems.

Analytic versus simulation methods. In doing performance analysis in a real-world design environment, the most important point is function, not form. The best method is arithmetic, but, beyond that, the choice depends on the knowledge of the modeller. If modeller is not aware of analytic methods, it is better to use simulator. Successful performance analysis uses both methods, and uses them together. Simulation models are used as submodels of analytic models, and conversely, in hybrid modeling [16], and analytic models can be used in simulation model verification.

Choosing a method. An analytic model is always preferable, if there is one that fits our model description, because of its solution speed. If an appropriate analytic model does not exist, it can be developed if time and skill permit, or perhaps a model can be adapted from technical literature [11].

However, it is not possible to develop an analytic model sometimes. If the problem complexity is much higher than the power of analytic methods, simulation is a must.

In this study, simulation model is used to evaluate the performance of a bus-based multiprocessor system. So, only simulation model is under our scope. The following sections of this chapter are based on this assumption.

3.2.5 Simulation Program Development

Developing a simulation program is very much like any other program development task. The main considerations are:

- simulation model design,
- program organisation,
- parameter management,
- debugging aids,
- instrumentation.

Simulation model design. After deciding upon simulation as the analysis method, the next step is to transform the model description into a simulation model design. This is the first point in the modeling process at which the simulation language's view is imposed on the model. If the method of simulation is event-oriented, the model design defines sequences of activities with their initiating and terminating events. However, if the model is complex, it should be tried to approximate a process-oriented view by developing separate definitions for different classes of processes. This essentially involves defining a separate event-oriented model for each class and specifying any inter-model coordination required.

Program organisation. The next transformation is from model design to program design. The organisation of the program depends on the complexity of the model. For simple models with few activities, the simulation program may be a single procedure with events identified by number. For somewhat larger models, separate function procedures will be used for each event routine. For complex model designs, the program is organised as a set of submodels, each of which may comprise a set of function procedures. There is a limit on complexity of model, when using smpl or any other event-oriented language. Some submodels may be combined at this point in which case their data structures have to be merged and modified.

Parameter management. When program is developed, one of the most important decisions is which parameters can be assigned by the user and which parameters will be

fixed. This effects the flexibility of simulator directly. When most of the parameters can be assigned by the user, complexity of the program increases. However, simulator can simulate different architectures under different conditions, so its flexibility also increases. It is clear that parameter management affect all the program development phase.

Simulation Program Debugging. Debugging is the task of getting the simulation program to the point where it runs without errors and the results it produces seem reasonable. This task can be done by the help of a special tool of the simulation languages. These tools are traces, dumps, reports, error messages. Those kinds of tools are available in any programming languages.

3.2.6. Verification

When the program produces reasonable results, verification study must be done. Verifying a simulation model means that the program is a valid implementation of the model [17]. For small models, this may be obvious from inspection; for larger models, some substantiating analysis is needed.

At a minimum, verification requires a comparative "walkthrough" of the model description and the simulation program. Sometimes this is all that is feasible, and success of the analysis effort depends on how diligently it is done. However, additional verification via comparison with analytic models often is possible. The simulation program is modified to represent a model for which analytic results can be obtained, and the simulation and analytic results are compared. This *analytic verification* does not, of course, guarantee that the program matches the model. However, it does provide a way to eliminate errors in at least part of the modeling process. If analytic verification is successful, then any remaining errors are either in transforming the model description to a model design or extending assumptions from the analytic model to the simulation model.

The results obtained from simulation rarely, if ever, will agree exactly with those obtained analytically. Simulation is a sampling process. Some difference between simulation and analytic results can result sampling variation. If this difference is small, it can be acceptable. Alternatively, analysis should be continued: collect additional data, compute confidence limits for the simulation estimate, and determine if these limits

include the analytic result. Another possible source of variation between analytic and simulation results is approximations used in the analytic model. So, the modeller should be aware of all assumptions.

3.2.7. Validation

Validation is the task of demonstrating that the simulation model is a reasonable representation of the actual system: it reproduces system behaviour with enough fidelity to satisfy analysis objectives. The question of how much is enough can be answered only in terms of these objectives and perhaps the results obtained in the current iteration of the analysis process. For example, demonstrating that model tracks real-system trends may be sufficient in a comparative analysis in which one alternative significantly outperforms the other. On the other hand, when a critical system parameter must be estimated within a few percent, the simulation model must be demonstrably capable of providing that accuracy. The simulation model usually is developed to analyse a particular problem and may represent different parts of the system at different levels of detail. The model does not have to be equally valid for all parts of the system over the full spectrum of system behaviour; it just has to meet the requirements of the problem.

There are two different cases of validation to consider. In the first case, the system being modelled exists and can be measured, and validation is based on comparison of model results with measurements. In the second case, the system being modelled exists only as a design, and the analysis objective is to estimate performance of the design or perhaps to evaluate alternative designs; little or no comparative data exists, and validation mostly is a matter of design-model comparison.

4. ARCHITECTURAL MODEL AND WORKLOAD MODEL OF THE SIMULATOR

In this project, a new simulation model is developed and implemented to predict the performance of bus based shared memory multiprocessor systems. This chapter is dedicated to explain all the details, algorithms and methods used in this model. However, there are some basic information, which must be explained. These are the reasons for choosing simulation modeling, the systems that can be simulated by this model, and general description of the simulator.

4.1. Why Simulation Modelling?

As it was explained in Chapter 3, modelling is one of the most important method to evaluate the performance of the computer systems. There are two main techniques to model a system: analytic and simulation.

Surely, there are many advantages and disadvantages of these methods. Analytic modelling is easier and cheaper to implement than simulation modelling. However, to represent complex systems is very hard for analytic models. Because the current mathematical theories used in analytic modeling, such as Petri nets and queuing theory are not enough powerful for complex systems. Although simulation is more expensive and requires more effort than analytic methods, it can model all multiprocessor systems.

The first aim of this project is to develop a general model that can represent all multiprocessor systems. This is possible only with simulation modelling.

4.2. The Architectures That Can Be Simulated

The first aim of the project was to develop a general simulator, which can evaluate the performance of all multiprocessor systems. In order to achieve this, a known model, EUCLID, was very suitable. EUCLID was developed by James Butler and Yavuz Oruç in 1986. According to this model, all multiprocessor systems are processing networks. They include processors and terminals (memory or I/O). The interconnection network (mapping)

between processors and terminals is the topology of the system [18]. The implementation of EUCLID was realised by Hüseyin Sepik and Tunç Yıldırım at Boğaziçi University as BUEUCLID. However, EUCLID has some disadvantages to simulate the multiprocessor systems. First, its simulation level is instruction level. It simulates all instruction execution in the system. However, simulating instruction execution for a large system would take prohibitively long. Moreover, user must write a real parallel program before simulation, and a wrong code in the program may break all the simulation. For these reasons, EUCLID (or BUEUCLID) was not found feasible to simulate especially large multiprocessor systems.

On the other hand, it is a fact that there are so many different topologies, protocols, and other parameters in multiprocessor systems it is hard to produce a general simulation model. So, the project is restricted and bus based shared memory multiprocessor system were selected for simulation.

4.3. General Description of the Simulator

A model of a multiprocessor system, and in general of any computing system usually consists of two parts: the representation of architecture, and the representation of workload. These are the main inputs of the simulator.

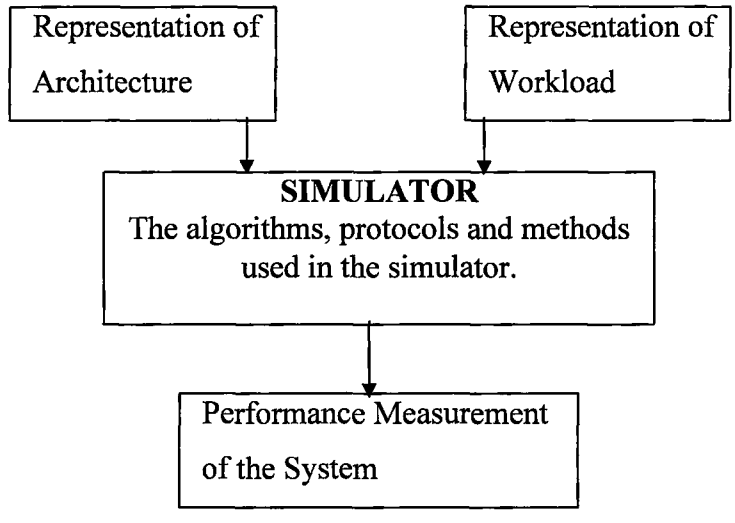


Figure 4.1. The block diagram of the simulator.

Undoubtedly, there are many algorithms, protocols, methods used in the simulator. The results of the simulation model are the performance values of the whole system. This is described in Figure 4.1.

In this chapter, representation of architecture, representation of workload, and algorithms and methods such as path finding, mailbox location, memory module clustering algorithms, will be explained.

However, first, another key element in the model development will be mentioned. It is the level of abstraction.

4.4. Level of Abstraction

The *level of abstraction* can be defined as the choice of the level of detail of the description of each subsystem, as well as the functional relationships and the rules of communication among building blocks. In order to define a proper system model, first it must be clearly defined the objectives of the analysis and then decide the level of abstraction of the representation. The level of abstraction should be chosen bearing in mind the parameters that significantly describe system performance. The evaluation (or the estimation) of such parameters is the actual goal of the analysis and must be performed as efficiently as possible. The model must contain all elements relevant to the analysis, whereas all the details that are not significant at that level of abstraction must be eliminated [8]. In the performance evaluation of a computing system, several possible levels of detail can be identified. Starting from the lowest level of abstraction (maximum level of detail):

1. Hardware level: In this level, all the details of the original system must be represented in the model. Even if, registers of the processor must be defined explicitly.

2. Functional level: This corresponds to the objective of evaluating the behaviour of basic hardware units, while they cooperate (or interfere) in performing basic operations. It includes less detail than hardware level.

3. System level: This corresponds to the objective of verifying the efficiency of the global system. It contains the least details.

The system level includes less information then the other levels. It is easy to implement a system level model, but it may not give enough information. Analytic methods for predicting the performance are suitable for system level. Because it includes less details and complexity about the system and analytic models can represent only simple systems. Hardware level is another extreme point in levels of details. It includes all the details, like the register of the processors. A model in hardware level is hard to use, because user must know and set all the details of the system. Moreover, it requires all the details of the program that will be run in the abstract machines. EUCLID is a good example for hardware level model.

The functional level covers more information than the system level, but includes less detail than the hardware level. This project is designed to be used in parallel computer courses. So, it can be considered as an educational tool. The possible users of this program will be students. If this model is in the hardware level of abstraction, users must write a parallel program and design a real detailed multiprocessor system. However, it may not be possible for a student, who has no detailed information about this kind of systems. If it is in the system level of abstraction, on the other hand, student can not research different types of architectures, protocols and topologies.

As a conclusion, it is clear that the most appropriate level for this simulator is the functional level.

4.5. Representation of the Architecture

Another important issue is the representation of the computer system architecture that includes the building blocks like supervisor structure, processors, memories and the interconnection network between shared memories and processors. These blocks will be mentioned in the following section.

First, the supervisor structure of the system will be introduced.

4.5.1. Supervisor Structure

When a program is run in a multiprocessor system, there are several issues must be done. For example, task to processor assignment, establishment of communication paths,

data distribution and alike. *Supervisor structure* manages all the system and tries to find the optimal solutions for these issues. One of the most important issues for a bus based system is supervisor.

There are two main supervisor structures in the multiprocessor systems. These are distributed supervisors and centralised supervisors. Centralised approach has also two main methods: Dedicated and floating supervisors. This classification is shown in Figure 4.2.

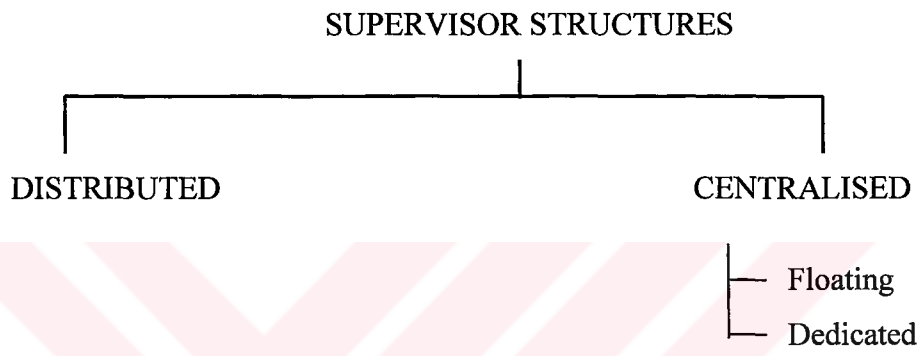


Figure 4.2. Classification of supervisor structures.

In *distributed supervisor* structure, there are some special supervision tasks in each processor. These tasks are responsible for the works of a supervisor. There is no specialised processor, which is responsible for the supervision of the system.

Another possibility is *centralised supervisor* approach. In this case, supervision of the system is done by a special processor. In other words, a processor is responsible to run the supervision tasks. If this special processor is not deterministic, if it may change, it is called a *floating* supervisor. In this situation, while system runs, supervisor tasks may be migrated to another processor. However, in *dedicated supervisor*, there is a special processor for supervision and it is static. The same processor must be supervisor from beginning to end of the run time.

The supervisor must communicate with the other processors. It may communicate with the other processor by the help of the existing interconnection network. However, in some cases, there is another interconnection network between supervisor and normal processor. All communications and processes between supervisor and processors are granted by this network.

In the simulator, it is assumed that, there is a dedicated supervisor and there is another interconnection network between supervisor and processors. The supervisor is directly connected to each processing element (Figure 4.3).

Moreover, all delays from supervisor and processor to supervisor communication time is assumed to be zero. For example, if a processing element tries to access a shared resource, this supervisor finds a path from processor to that resource. However, time to find that path is assumed as zero.

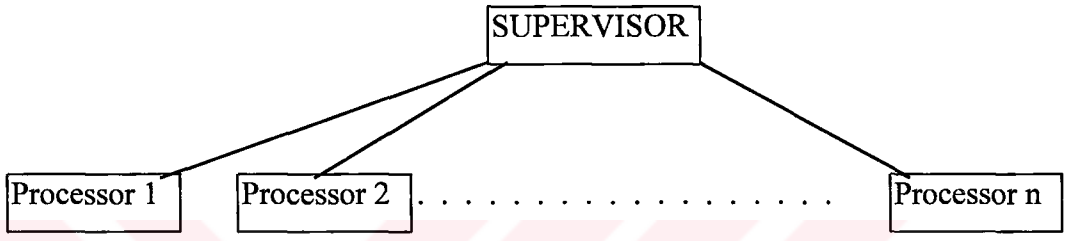


Figure 4.3. The supervisor structure of the simulator.

4.5.2. Basic Elements of Multiprocessor Architecture

The representation of multiprocessor architectures are based on three basic elements These are processors, memories and the interconnection network structure. First, processing elements will be explained.

4.5.2.1. Processing Elements

In this simulation model, a processing unit consists of a processor, a cache and a private memory. In this section the properties of these units will be introduced.

Processor. *Processors* are the core unit of a computer system. There are several kinds of processors, which are commercially available. The most commons are RISC based and CISC based ones. Surely there are many differences between them. However, in functional level, all differences and details are hided. These differences effect only the performance

of the processors. So, differences between internal architectures of processors are embedded in the speed of the processing unit.

Private Memory. In this simulator, it is assumed that all processing units have a *private memory*. This type of memory may be used by its related CPU. It includes program codes or frequently used data. The aim of this memory is to operate the CPU faster. A CPU can access its private memory with no handicap or contention, because only that CPU can reach its private memory.

Cache. In many cases, processors try to access to its private memory unit. However, private memory may be slow with respect to CPU. In this case, CPU needs a faster memory to operate faster. This faster memory is called as *cache*. Cache is placed between processor and private memory. Cache includes most frequently needed data. When processor tries to access data, most probably, it can find the desired data in the cache. In the simulation model it is assumed that there is a cache in each processing unit.

Figure 4.4. shows the processor, cache and private memory in a processing unit.

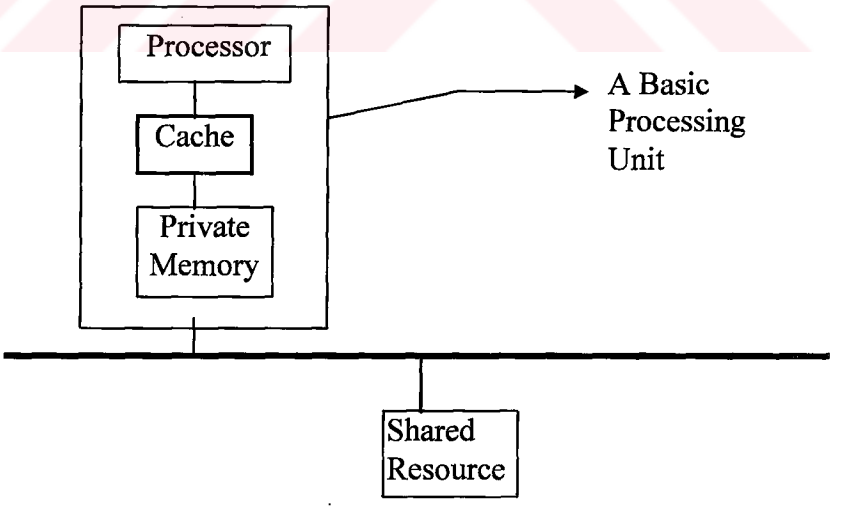


Figure 4.4. A processing unit in a shared memory system.

The only important parameter of a processing unit is the speed. The *speed of a processing unit* is the work done within a processing unit in one time unit. The work with a processing unit includes works within CPU, works with the interaction of its private memory or its cache. Default speed of a processor is 1000 unit work in one unit time, but

user may change the speed of each processor in the system. Simulator allows simulating processors with different speeds.

4.5.2.2. Shared Memory Units

A shared memory may accept and serve requests issued by several processors. In few architectures, memories are capable of serving several simultaneous access requests. Memories of this type are referred to as multiport memories. However, in this model all memories are assumed to have only one port. In this case, a memory can serve only to one request at a time. The only property for a memory that user can set is the speed. The default speed of a memory unit is 100 unit data in one unit time. However, as in processor, user can set the speed of each shared memory unit.

4.5.2.3. Interconnection Network

The interconnection network links processors to memories. It is very important for the performance of the system. In this model, only bus-based interconnection networks are covered. *Bus* is a shared communication link connecting all the system component.. Processors can access shared resources through buses. A bus can hold data for only one processor. In other words, only one processor can use a bus at a given time. When a processing element requires accessing a shared memory, then a *path*, which is a collection of buses that connects processor to memory, is requested for that processor. If that path is granted, processor can communicate with shared memory.

4.6. Representation of Workload

Program that runs on the system is as important as the architecture itself. It effects performance results directly. The task of describing the work performed by a system is called as *workload.*

In a normal multiprocessor system, a program is written in a parallel programming language. The compiler compiles this program and optimises the parallelism of the

program. When a programmer compiles the program via a parallel compiler, it partitions the program and data and identifies the parallelism. The parallelism inherent in a program can be modelled as a directed acyclic task graph (DAG). After that, task graph is scheduled on a parallel machine. At the end, the program is ready to run on a parallel computer. The fundamental steps of compiling a program in multiprocessor system are shown in Figure 4.5.

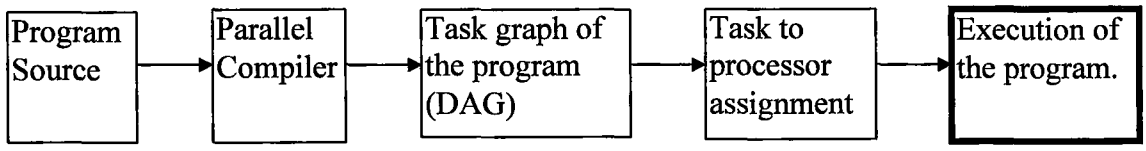


Figure 4.5. The states of a parallel program.

A directed acyclic weighted task graph (DAG) is defined by a tuple G = (V, E, C, T) where $V = \{n_j, j = 1 : \gamma\}$ is the set of task nodes, E is the set of communication edges and $\varepsilon = |E|$ is the number of edges, C is the set of edge communication costs and T is the set of node computation costs. The value $\zeta_{i,j} \in C$ is the communication cost incurred along the edge $\varepsilon_{i,j} = (n_i, n_j) \in E$, which is assumed to be zero if both nodes are mapped in the same processor. The value $\tau_i \in T$ is the execution time of node $n_i \in V$ [19]. An example of a complete task graph is shown in Figure 4.6.

A task is a unit of computation that may be an assignment statement, a subroutine or even an entire program. In the task computation, a task waits to receive all data in parallel before it starts its execution. As soon as the task completes its execution it sends the output data to all successors in parallel [19].

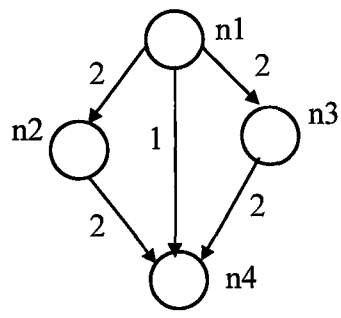


Figure 4.6. An example of a task graph with node weights equal to 1.

In this model, the task structures and communications between tasks are realised by a different mechanism. In the following sections, the details of this mechanism will be explained.

4.6.1. The Structure of the Tasks and Primitive Routines.

From time to time, tasks may need to access a shared resource because of the cache misses. So, tasks are interrupted and these interruptions affect the system performance dramatically. However, in a normal task graph, all tasks are convex, which means that once a task starts its execution it can run to completion without being interrupted for communication. Therefore, modelling these interruptions is not possible with a normal task graph. In order to model them, a different mechanism must be introduced. This mechanism can be realised by the *primitive routines*. These routines are the fundamental parts of a task. Another name of primary routines is *sub-task*. There are four basic sub-tasks to represent the interruption due to cache misses. These are:

- CPU Sub-Tasks. This represents the work with no interruption. In this kind of sub-task, processor tries to access to private memory location or its cache. Processor does not need any shared resources.

- MEM Sub-Tasks. This represents the accesses to shared memories. When a processor can not find the required data in its cache, it tries to access a shared memory. There are two conditions that must be satisfied to complete. First, the buses between the processor and shared memory must be captured. Secondly, that shared memory must be idle.

The other two sub-tasks will be explained in Section 4.6.2.

A task consists of consecutive structures of these routines. An example task is shown in Figure 4.7.

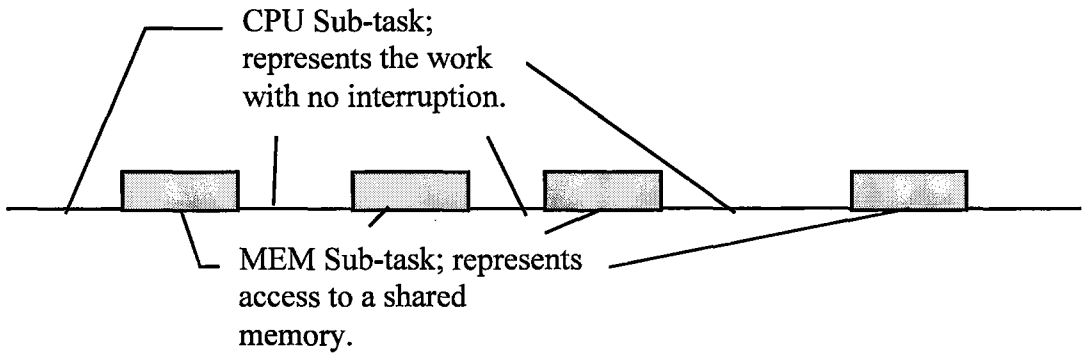


Figure 4.7. An example of task.

In order to understand the workload characterisation of this model, completion times of sub-tasks and the length of a task must be explained.

4.6.1.1. Completion Times of Each Sub-Task and System Load

Completion times of each sub-task may seriously affect the performance results of the simulator.

Completion time of a CPU sub-task is defined by the amount of work in the CPU sub-task over the speed of processor. The amount of work in a CPU sub-task is a random number that is uniquely distributed between 20000 and 100000 unit work. As the default speed of a processor is 1000 unit work in one time unit, the completion time of a CPU sub-task is between 20 and 100 unit time which is consistent with the recent studies [20]. Surely, when the processor speed is changed the completion times will be also changed.

Computing of MEM sub-task completion times is much likely in computing of CPU sub-task completion times. It is the volume of data, which will be transported on this sub-task over the speed of the memory. It is assumed that the volume of data is a random number between 2000 and 10000 unit data. It is also uniquely distributed. As the default speed of a memory is 100 unit data in one unit time, the completion time of a MEM sub-task is between 20 and 100 unit time again.

Many of the analytical models in the literature assume that duration times of sub-tasks are exponentially distributed and simulator must be compatible with the known models for

the verification study which is explained in Section 3.2.6. In order to realise the compatibility of the duration times of the sub-tasks in the models, simulator can generate exponentially distributed random number, instead of uniquely distributed random numbers. This utility is used in Chapter 5, especially.

Average completion time of CPU sub-tasks over average completion time of MEM sub-tasks has an important role on the performance results. In many analytical models, this ratio is called *system load* and considered as one of the most important parameters about the workload of the system. The formulation of system load is given in 4.1.

Let λ be average completion time of CPU sub-tasks.

Let μ be average completion time of MEM sub-tasks.

System Load, $\rho = \lambda / \mu$. Formula 4.1.

For example, user uses the default speeds of processors and memories. As it was calculated before, completion times of CPU and MEM subtasks are uniquely distributed random numbers between 20 and 100. In this case, the average completion times of CPU and MEM sub-tasks are (20+100)/2 = 60. According to Formula 4.1, system load will be 60/60 = 1.

When the user tries to simulate the system with the workload whose system load ratio is 0.5, there are two methods. One of them is to set the average completion time of MEM sub-tasks to 120. The other one is to set the average completion time of CPU sub-tasks to 30. This can be realised by decreasing the speeds of memory units to 50 unit data in one unit time or increase the speeds of processors to 2000 unit work in one unit time. User can set the system load in this way, and he/she can investigate the effects of different workload conditions. This method will be used in verification and validation studies in Chapter 5 and 6.

4.6.1.2. Length of Tasks

In real parallel programs, lengths of tasks may not be equal. Some tasks are longer than the others. This non-uniformity may affect the performance of the system. So, it is a very important topic in the simulator.

The length of a task depends on the number of sub-tasks in it. User can set the number of sub-tasks in a task. In this way, he/she can change the lengths of the tasks. So, simulating tasks with different lengths is possible.

4.6.2. Task Synchronisation Mechanism

This simulator can simulate the contention that is arisen from accessing a shared memory. However, there is another important issue that affects the system performance. It is task synchronisation.. Cooperating processes or threads in a multiprocessor environment often communicate and synchronise. Such interprocess communication employs one of two schemes: shared variables or message passing.

Multiprocessor operating systems have experimented with a large variety of different communication abstracts, including ports, mailboxes, links and others. From an implementation point of view, such abstractions are kernel-handled message buffers. However, only the mailbox mechanism is under the scope of this simulator.

Up to now, when a task is modelled by sub-tasks, it is assumed that all tasks are independent and there is no synchronisation between tasks. In many simulators, this issue is ignored and it is assumed that there is no delay due to task communication. In reality, one of the handicaps in a multiprocessor system is task synchronisation between tasks. In this simulation model, this constraint is one of the main parts of the workload model. The following primitives are introduced to model task synchronisation mechanism:

- MP Sub-task: This sub-task represents the message passing to another task. It writes the required data to the mailbox.

- RM Sub-task : This sub-task represents the receiving message from another task. It reads the required data from the mailbox. If task can not receive this message, it will be suspended, until it can access and get the message it needs.

In Figure 4.7, there is a deficiency about message passing or receiving message. MP or RM type sub-task must also be a part of a task. After the tasks are created from CPU and MEM sub-tasks, all the links between tasks are added. This means to add MP and RM primitives to the tasks. An example task with MP or RM sub-task is as in figure 4.8.

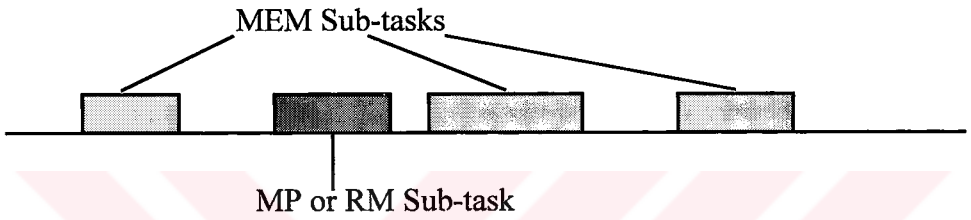


Figure 4.8. An example of a task structure. MP or RM is added.

There are two ways of inserting MP and RM primitives. One of them is the classic way. All MP's are at the end of the task and all RM's are at the beginning of the task as in Figure 4.9.

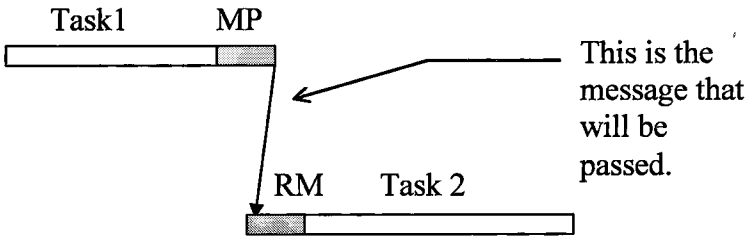


Figure 4.9. MP is at the end and RM is at the beginning of the task.

In this system, no task can be initiated, unless it gets all the messages that are required. This is the most common method in the simulation models. However, it is not the real case. In reality, MP sub-tasks does not have to be at the end of the task and RM sub-

task does not have to be at the beginning. They may be placed at different places of the tasks as in Figure 4.10.

In this case, RM may not be in the first part, and, passing message may not be at the end of the task. So, each task can be run immediately.

In this simulation, there are two options that support the situation in Figure 4.9. and Figure 4.10. User can select one of them.

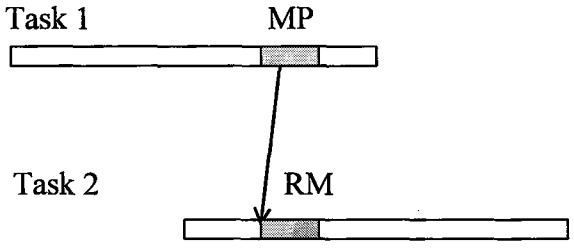


Figure 4.10. An example. MP and RM is in the task.

In more realistic case, there may be more than one message passing from one task to another. Simulating such cases is also possible in this simulator. Each link between tasks has a certain property for the number of messages between tasks. Its default value is 1. User can change that value. For example, if the value of this property is changed to 2, the sender task must include 2 MP sub-tasks and receiver task must include 2 RM sub-tasks. Figure 4.11. illustrates this situation.

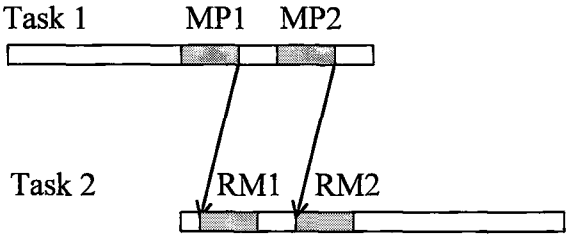


Figure 4.11. A complete example of tasks with two links.

The volume of data to be sent or received is another issue that should be discussed. It affects the performance of the system, and it is also placed in the model. This simulator can model different volumes of data. User can set the volume of data in the message. As a result, the effects of different volumes in different messages can be investigated.

4.7. Path Finding Algorithm

Processors sometimes need to access to shared memory modules with the help of the interconnection network. When a processor tries to access to a shared memory module, a *path* between processor and memory must be established. This path is needed to transport the data. The main questions for this problem are which buses will be used or which path will be established.

For some architecture, the answers of these questions are simple. If there is only one path from processors to memories, as in hierarchical bus based multiprocessor systems, there is no need for choosing the optimal path. In each path establishment, that path must be used. However, in some architecture, there may be more than one path, as in multiple bus architecture. When there are more than one path, system should choose the most efficient one.

There are two main approaches for path finding problem. One of them is static approach. All the paths are fixed. If a processor tries to access to a memory unit, path that will be used is fixed and is already known by the system. This path is the shortest path in most of the systems.

On the other hand, there are some other systems that allow dynamic path finding. In these kinds of systems, if there are more than one path, supervisor finds the most efficient path from processor to memory.

This simulation model includes dynamic path finding algorithm. So, a processor may access to a memory from different paths.

There are three helpful procedures that must be explained to understand path finding algorithm clearly. First, AllPath procedure will be explained.

AllPath algorithm finds all paths from a processor to a shared memory. These paths are stored into PEtoMEMPaths global variable. This procedure is run for all pairs of processor and memory for only once. In this way, all possible paths are stored into

PEtoMEMPaths. When simulator needs the possible paths from a processor to a shared memory, it can find them from PEtoMEMPaths variable. PEtoMEMPaths variable will be used in mailbox location algorithm in Section 4.8. The AllPath algorithm is shown in Figure 4.12.

The time complexity of AllPath procedure is $O(2^n)$, when there are n buses in the architecture.

```
procedure AllPath (E1 : Element, E2 : Element, P: Processor, APath : Path)
' Element is a processor or a memory or a bus
 if E2 is directly connected to E1 by a bus, ABus then
   APath.Add ABus
   PEtoMEMPaths(P, E2).Add APath
   Terminate
 else
   for each bus, ABus which is connected to E1
     if ABus is not in APath then
       APath.Add Abus
       AllPath(Abus, E2, P, APath)
       APath.Remove Abus
     endif
   next
 endif
end
```

Figure 4.12. AllPaths procedure.

The other procedures, which will be explained, are the load on a bus and load on a path algorithm. Load on a bus is shown in Figure 4.13.

```
function LoadOnBus (B : Bus) : integer
  for each task, T, waiting for B
    TotalVolume = TotalVolume + T.DataVolume
  next
  LoadOnBus = TotalVolume / B.Bandwith
end
```

Figure 4.13. LoadOnBus function.

When a processor requires a bus and it can not be granted, that processor is put in a bus request queue. Each processor requests the buses to read or write a certain volume data. *Load on a bus* is total volume of data, which are required by the waiting tasks on that bus over bus bandwidth. In other words, load on a bus is time to complete all waiting requests. The code for this load calculation of a bus is shown in Figure 4.13. When there are t tasks in the system, the time complexity of this function is O(t) in the worst case.

It is time to explain the load on a path. It is as in Figure 4.14.

Load on a path is the maximum of loads on the buses in that path. The time complexity of the algorithm is O(b*t), where b is the number of buses and t is the number of tasks.

```
function LoadOnPath( P : Path) : integer
  load=0
  for each bus, B, in P ' this loop finds the maximum load
    if load < LoadOnBus(B) then
      load = LoadOnBus(B)
    endif
  next
  LoadOnPath = load
end
```

Figure 4.14. LoadOnPath function.

The path finding algorithm that uses LoadOnPath and AllPath algorithms is in Figure 4.15. The complexity of this function is O(b*t*p), if b is the number of buses and, t is the number of tasks and, p is the number of paths from processor to memory. The aim of the algorithm is to balance the loads of the buses. If there is a high loaded bus in a certain path, algorithm will try to find alternative paths. The algorithm will be clarified by the help of an example.

```
function PathFinding(P : Processor, M : SharedMemory) : Path
  PossiblePaths = PEtoMEMPaths(P, M)  :  Min = + ∞
  for each path, Apath, in PossiblePaths ' this loops finds the minimum load
    if Min > LoadOnPath(Apath) then
      Min = LoadOnPath(Apath)
      PathFinding = Apath
  endif
  next
end
```

Figure 4.15. Path Finding algorithm.

A part of a multiprocessor system is shown in Figure 4.16. In this figure, the names and the loads of the buses are written. If a task in PE1 tries to access to MEM1, a path must be established.

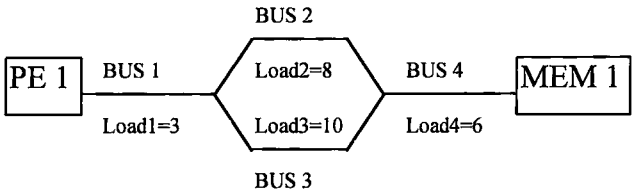


Figure 4.16. A part of a multiprocessor system.

First of all, according to path finding algorithm, all possible paths must be found. There are two alternative paths. One of them is a path with BUS1, BUS2, BUS4 and the other one is BUS1, BUS3, and BUS4.

The minimum of the path loads must be calculated. The load of the first path is the maximum of the bus loads of BUS1, BUS2, and BUS4. This is 8. The load of the second path is the maximum of the loads of BUS1, BUS3, and BUS4. It is 10.

The first path has the minimum load. At the end, the first path, BUS1, BUS2, BUS4, is chosen.

The traffic on the second path is higher than the first one. In order to balance the traffic, algorithm chooses the first path.

4.8. Mailbox Problem

In this simulation model, processors execute tasks that cooperate by passing messages through mailboxes established in shared memory. *Mailboxes* are the meeting places of communicating processes. When two tasks need to communicate, supervisor should establish a mailbox at an optimal location, it must communicate with the other task and, a mutual place must be found.

The place of mailbox may effect the whole performance of the system. A mailbox that is far away from sender and receiver causes to increase the delays.

Mailbox finding algorithm finds an optimal place for communication of tasks. In order to understand this algorithm, first distance function must be introduced.

```
function Distance(P: Processor; M: SharedMemory) : integer
   AllPaths = PEtoMEMPaths(P, M)
   for each path, Apath, in AllPaths
      NumberOfBus(P)= the number of bus in APath
   next
   Distance= minimum value in NumberOfBus
end
```

Figure 4.17. Distance algorithm.

The main aim of this algorithm is to find the number of buses used in the shortest path from processor to shared memory. In this procedure, all possible paths from processor to

shared memory are found by the help of PEtoMEMPaths list, which is explained in Section 4.7. The buses used in each path are counted in the next step. At the end, the minimum of them is selected. The code of the algorithm is given in Figure 4.17. The complexity of distance function is O(p), where p is the number of paths from processor to memory.

The algorithm, which is in Figure 4.18, is the mailbox algorithm that uses distance function.

```
function Mailbox (Task1, Task2: Task) : SharedMemory
  dim PE1, PE2 as Processor
  PE1 = the processor that runs Task1
  PE2 = the processor that runs Task2
  if PE1=PE2 then
    Mailbox = Null
    terminate
  else
    for each shared memory, M, in the whole system 'beginning of the first stage
      Formula1(M) = Distance(PE1,M)+ Distance(PE2,M)
    next
    MinFormula1 = minimum value in Formula1
    create a list, Candidates, keeps all memory modules
    that Formula1 value is MinFormula1 'ending of the first stage
    for each shared memory, M, in Candidate list 'beginning of the second stage
      Formula2(M)= Abs(Distance(PE1,M)-Distance(PE2,M))
    next
    MinFormula2= minimum value in Formula2
    MailBox= a memory module which formula2 value is MinFormula2
  endif 'ending of the second stage
end
```

Figure 4.18. Mailbox finding algorithm.

In the algorithm, first, the processors that run the tasks are found. If the tasks are in the same processor, they can communicate through the private memory of the processor. So, there is no need to find a mailbox.

However, if they are not in the same processor, the first formula will be calculated for each shared memory module in the system. The first formula is Distance(PE1,M)+ Distance(PE2,M). This formula represents total number of buses that will be busy when the message is sent and received, if M is selected as mailbox. This algorithm tries to minimise this value. Those memory modules that minimises the total number of buses which will be used for communication (i.e. send and receive) are added to the **Candidates** list. In the second stage, algorithm tests and selects memory modules from **Candidates** list. The aim of this stage is to balance the distance of the mailbox to sender and receiver processors. When M is far from PE1 and near to PE2, there may be a problem. To establish a long path is harder than a short path. Therefore, communication delay for TASK1 and TASK2 may be longer. This will degrade the performance. However, if distance(PE1,M) and distance(PE2, M) are nearly equal, it will be easier to establish the paths. The balance of the distances is achieved by Abs(Distance(PE1,M)-Distance(PE2,M)) formula.

This algorithm results in most efficient mailbox location, because combined send and receive hops are minimised. Moreover, the distance of chosen shared memory unit to each processor is balanced. The complexity of mailbox finding algorithm is simply O(m*p), if m is the number of memory modules and, p is the maximum number of paths from a processor to a memory.

Figure 4.19 may be considered as an example to understand the algorithm clearly. Assume that, two communicating tasks are assigned to PE1 and PE2. For M1 Distance(PE1,M1)+Distance(PE2,M1) is 6. That is the same for M2. It is 10 for GM and 16 for M3. Obviously, M1 and M2 are selected in the first stage.

The result of Formula2 for M1, Abs(Distance(PE1,M1)-Distance(PE2,M1)), is 2 and it is the same for M2. In this case one of them is selected randomly.

However, if a communication between PE1 and PE3 is needed, the result will be different. Total distance calculated in the first stage of the algorithm is 10 for M1, for GM and for M3 and it is 13 for M2. So, M1, GM and M3 are selected as candidates.

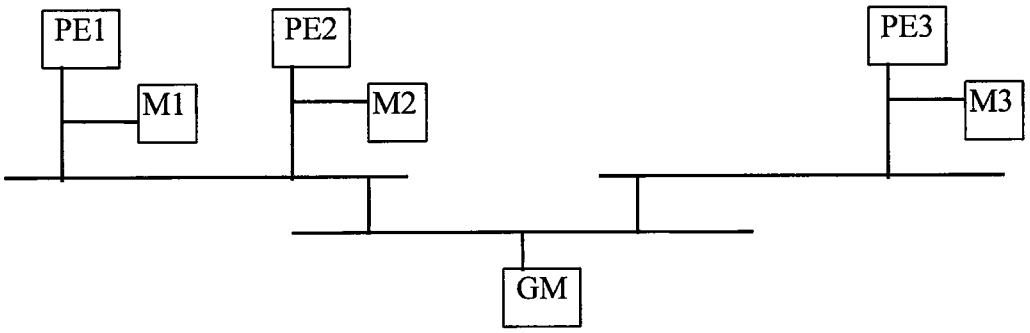


Figure 4.19. An example of hierarchical bus based multiprocessor system.

In the second stage, the absolute differential distances are calculated as 6 for M1 and M3; but it is 0 for GM. In this case, GM is selected as mailbox.

4.9. Memory Module Clustering and Data Distribution

One of the most important functions of a supervisor is to distribute the data over the shared memories, so that each task can find the required data in the nearest memory module as long as it is possible. This is called as *data distribution* and it affects the performance of a multiprocessor system seriously. In order to simulate a real data distribution case, the usage frequency of each shared memory unit must be formulated. This formulation can be realised by the *clustering algorithm.*

The clustering algorithm is based on the distance from processor to memory units which is explained in mailbox problem, Chapter 4.8. The details of the algorithm are shown in Figure 4.20.

The distance between a processor and a memory is the minimum number of buses that connect them. According to the clustering algorithm, The closest memory or memories are classified as the first cluster. Other closest memory or memories after the first cluster are classified as the second cluster. All memories are classified in this way, until all memory modules are clustered.

This algorithm must be run for each processor, because a specific memory module may be far from a processor and it may be near to another processor. So, the cluster sets of each processor may be different.

```
procedure Clustering (P : Processor)
  dim i, MinDistance as integer
  i = 1
  while there is still unclustered module do
    MinDistance = minimum distance between P and unclustered modules.
    Cluster(P, i)=a list contains the memories so that their distance is MinDistance
    Mark memory modules in Cluster(P, i) as clustered.
    i = i+1
  wend
end
```

Figure 4.20. Clustering algorithm.

All clusters are kept as a list. In this way, this algorithm is run only once. When simulator needs the cluster sets, it does not run the algorithm and takes this data from the list. The complexity of this algorithm is O(m*p), if m is the number of memory modules and p is the number of maximum paths from a processor to a memory.

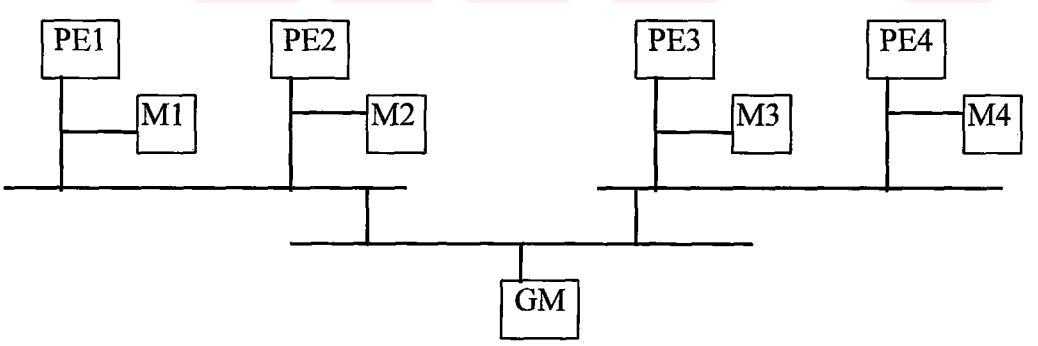


Figure 4.21. An example of hierarchical bus based multiprocessor system.

An example will help to understand the clustering algorithm. Let the system be a hierarchical bus based multiprocessor system as in Figure 4.21. First, clustering algorithm will be executed for PE1. Distances from PE1 to shared memories are as follows:

Distance (PE1, MEM1) = 2.

Distance (PE1, MEM2) = 4.

Distance (PE1, GM) = 5.

Distance (PE1, MEM3) = 8.

Distance (PE1, MEM4) = 8.

At the beginning, there is no shared memory module, which is clustered. It is clear that, the closest unit, in other words, the unit with minimum distance is MEM1. So, MEM1 is clustered as the first cluster. After that, the closest unit that is not clustered is MEM2. It is clustered as the second cluster. GM is clustered as third cluster in the same way. There are two modules, MEM3 and MEM4, which are not clustered and their distances are the same. So, these units are clustered as the fourth cluster.

However, the situation is different for each processor. For example, the first cluster of PE3 is M3. The second cluster is M4. The third cluster is GM. And the third cluster of PE3 is MEM1 and MEM2.

Clustering information is not enough to simulate the real data distribution case. There must also be a probability distribution that defines the frequency of accesses to cluster sets. This probability distribution can be defined as follows:

Assume that, i is the cluster number and there are n clusters. In this case, i =(1..n). Let c_i be i^{th} cluster, which is calculated in clustering algorithm, and p_i be the conditional probability of c_i to be chosen, given that clusters c_1 to c_{i-1}, are not chosen. For example, let p_3 be 0.9 in certain architecture. If c_1 and c_2 are not chosen, the probability of selection of c_3 is 0.9. It should be reminded that if there are n clusters, p_n must be 1. The probability of choosing cluster c_i, P (c_I), is as follows:

$$P(c_I) = \left[\prod_{k=1}^{i-1} (1-p_k) \right] * p_i \qquad \text{Formula 4.2.}$$

According to 4.2, the probability of being chosen of the first cluster is p_1. The probability of the second cluster is $(1-p_1)*p_2$. And it is $(1-p_1)*(1-p_2)*p_3$ for the third cluster. The others can be calculated in the same way.

A cluster is a set of shared memory modules. If there are more than one shared memory units in the selected cluster, the probability of selection of each module in that

cluster is uniformly distributed. If $Cluster_j$ is selected, and if there are m modules in $Cluster_j$ and if MEM_k is a member of $Cluster_j$, the probability of selection of MEM_k is 1/m. For example; if $Cluster_3$ is the selected cluster and if it contains three shared memory modules, MEM_1, MEM_2, MEM_3, the probability of selection of MEM_1 is 1/3. The probabilities for MEM_2 and MEM_3 are the same, 1/3.

As it has been explained before, in order to simulate a real data distribution, clustering information and a probability distribution must be defined. In the example that is shown in Figure 4.21, clustering information of PE1 was presented. The cluster sets was:

Cluster(1) = {MEM1}
Cluster(2) = {MEM2}
Cluster(3) = {GM}
Cluster(4) = {MEM3,MEM4}.

If usage probability of each cluster is defined, the data distribution can be simulated correctly. Let p_1, p_2, p_3 are 0.9. This is the default value in the simulator. But p_4 must be 1, because it is the last cluster. So, the probability of M1 to be chosen is 0.9. If M1 is not chosen, that resource will be M2 with the probability of 0.9. If it is not M2, the probability of GM will be 0.9. At the end if it is not GM, it will be M3 or M4. According to Formula 4.2, the probability distribution for PE1 will be as follows:

The probability of M1 is 0.9
The probability of M2 is (1-0.9)*0.9
The probability of GM is (1-0.9)* (1-0.9) *0.9
The probability of M3 is (1-0.9)* (1-0.9)* (1-0.9)*0.5
The probability of M4 is (1-0.9)* (1-0.9)* (1-0.9) *0.5
Total probability is 1.

Surely, these probabilities are different for each processor.

Another flexibility of the simulator is the conditional probabilities, p_i's. Although the default values of the pi's are 0.9, user can set these. If probability values are decreased, tasks can not find the required data in the closer modules. This implies a bad data distribution case. If the values are increased, tasks can find required data in the closest

modules. So, user can simulate good or bad data distribution conditions with the help of these probability values.

4.10. Cache Coherence

As it was explained, cache is a special memory associated with processor. Traffic through the bus network can be reduced as much as %95 by using a cache memory [2]. But while processor caches can significantly improve system performance, they introduce a coherence problem due to the presence of multiple cached copies of main memory locations. It is necessary to ensure that changes made to shared memory locations by any one processor are visible to all other processors.

There are two basic approaches for cache updating; write-back and write-through. Basically, write-back or write-through presents extra load on the system, when it updates the data.

In this simulation model, there is no explicit cache structure, there is no explicit cache simulation methods, and algorithms that simulates the cache coherence problem. Cache coherence algorithms brings extra load to the system. Accessing to shared memory was represented by memory sub-tasks. The traffic load in the system was determined by sub-task's completion times. So, the extra load which cache coherence brings to the system can be represented as the extensions of MEM sub-task completion times [20]. This can be regarded as a translation from a high-level workload model to a low-level workload model. How to calculate the MEM sub-task completion times was explained in 4.6.1.1.

This simulation model can not give results about the effect of cache size, cache speed; because these are not represented explicitly. However, at least, cache coherence traffic can be included.

4.11. The States of Tasks in Processors

Each task on the system is on a state in the processor. There are four main states for a task. These are READY state, ACTIVE state, CURRENT state, and PASSIVE state. The basic state transition diagram as in Figure 4.22. First, READY state will be explained.

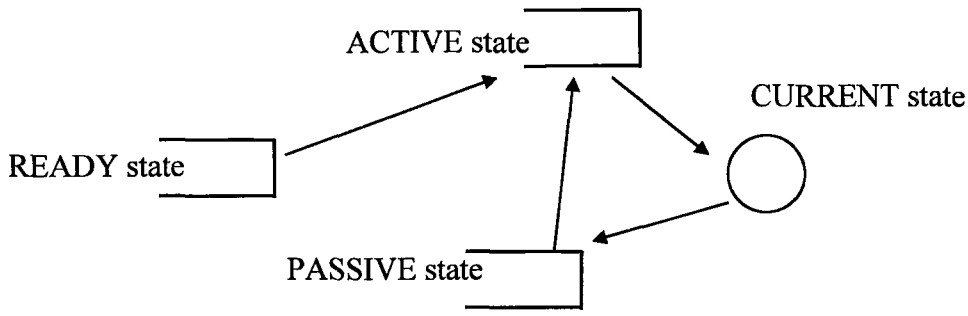


Figure 4.22. State transition diagram of tasks.

In this simulation model, there are some limitations of processors. One of these is maximum number of outstanding tasks. As default, three tasks can be executed in the same processor at the same time. If there are already three tasks in the processor, other tasks must wait and can not be activated. Those kinds of waiting tasks are in READY state. They are ready to be activated, but the capacity of the processor is full. When an activated task is finished, one task can be migrated from READY state to ACTIVE state.

It was expressed that the capacity of a processor is three as default. However, the user can set this capacity. This is another flexibility of the simulator.

ACTIVE state is another important state for tasks. The tasks in the ACTIVE state wait for central processing unit of the processing element. Central processing unit, which is in processor, can execute only one task at a time. When this unit is busy, the other tasks must wait. When the central processing unit in the processor is idle, one task, which is in ACTIVE state, will be migrated from ACTIVE state to CURRENT state.

CURRENT task is a task that is currently executed in the CPU. There can be only one task in the central processing unit, so it includes only one task. If a task is in CURRENT state, its sub-task must be CPU sub-task. If this CPU sub-task is executed, its state will be PASSIVE.

If the task in the CURRENT state is migrated to PASSIVE state, and there is no task in CURRENT state, one task is selected from ACTIVE state and it is put into CURRENT state. If there is no task in CURRENT state and ACTIVE state, processor will be idle.

PASSIVE state is the last state to be explained. If a task, which is in CURRENT state, requires a memory operation or waits for a message or tries to send a message, it is sent to

PASSIVE state. When the memory operation of a PASSIVE task is granted or its message is sent or received, task will be in ACTIVE state.

4.12. Protocols

Transition from READY state to ACTIVE state or transition from ACTIVE state to CURRENT state requires a selection process. Which task will be migrated? Or when two processors try to access to the same bus, which processor will gain access? There must be a rule for selection process. These selection rules are called as *protocols*. There are two main protocols in a multiprocessor system. One of them is for bus contentions and the other one is for selection of tasks.

4.12.1. Protocols for Bus Contention

When a bus is required by more than one task, one of them must be selected and that bus must be assigned to selected task. This selection rule is called as *bus contention protocol*. Protocols for bus contention are used as follows. Bus contention protocols that this model supports are the followings:

- Defined Priority. In this kind of priority, user gives a priority value for each task. This value is static and does not change during run time. A task with minimum defined priority captures the bus. Its complexity is O(n), because it is a minimum finding algorithm.

- Randomly. A task is chosen randomly. That task achieves to gain the bus. Complexity is simply O(1).

- FIFO. If a task wants the bus first, bus is used by that task i.e. first in first out. The complexity is O(1).

- LRU. In least recently used protocol, the task that used that bus least recently has the highest priority. The task that is the oldest in the waiting list gains the bus. The complexity is O(n).

- Maximum Distance Protocol. The aim of this protocol is to give priority the task that must establish longer paths. If the distance from processor to memory is longer, to establish that path is harder. So, there should be a priority for that kind of tasks. In this protocol, bus is assigned to a task that tries to access to a shared memory from maximum distance. The function is shown in Figure 4.23. When there are t tasks in the workload , the time complexity is O(t).

```
function MaxDistanceProtocol (Candidates : list of Task) : Task
  Max = 0
  for each task, T, in Candidates list
    if Max < number of buses in T.RequiredPath then
      Max = number of buses in T.RequiredPath
      MaxDistanceProtocol = T
    endif
  next
end
```

Figure 4.23. Maximum Distance Protocol

- Maximum Bus Efficiency Protocol. The aim of this protocol is to maximise the efficiency of the traffic through buses. The efficiency depends on the number of buses in the path and the volume, which will be transported. In this protocol, the priority of a task is volume of the data, which will be transported, times number of required buses. The task with maximum priority gains the bus. Figure 4.24. gives the code of this protocol. The complexity of this protocol is O(t) again, where t is the number of tasks.

```
function MaxBusEfficiency (Candidates : list of Task) : Task
  Max = 0
  for each task, T, in Candidates list
    if  Max < (number of buses in T.RequiredPath)*T.DataVolume then
      Max = (number of buses in T.RequiredPath)*T.DataVolume
      MaxBusEfficiency  = T
    endif
  next
end
```

Figure 4.24. Maximum Bus Efficiency Protocol.

- **Important Message.** When a task can not receive the required messages, it must wait. One of the most important factors, which reduce the system performance, is the delay due to the messages. If a task has a mechanism for sending a message (MP sub-task) in the following sub-tasks, this task should have the priority to be executed. If this task is given then priority, it can send the message earlier and the receiver task can be activated faster. Surely, this will improve the efficiency of the system. In this protocol, this kind of scenario is considered. The tasks which may activate another tasks has the highest priority. The program code of this protocol is given in Figure 4.25. When there are t tasks, the complexity is O(t).

```
function ImportantMessage(Candidates : list of Task) : Task
  Max = 0
  for each task, T, in Candidates list
    if  there is a message passing , MP, operation in the following 5 sub-task then
      ImportantMessages = T
      Terminate
    endif
  next
end
```

Figure 4.25. Important Message Protocol.

4.12.2. Protocols for Task Selection

When a transition from READY state to ACTIVE state or a transition from ACTIVE state to CURRENT state must be done, a task must be selected from ReadyTask list or from ActiveTask list. The selection rules are called as *task selection protocols*. The followings are the protocols that is included into this simulation model.:

- Defined Priority. It is the same in bus protocols. User defines all the protocols statically. These priority numbers are used to select a task.

- Randomly. A random task is selected from list, and that task is chosen.

- FIFO. First in first out for task selection. It selects the first entry of ReadyTask or ActiveTask list.

- Important Messages. This protocol works with same logic as in bus contention important message protocol which is explained in Figure 4.25.

User is allowed to select protocol algorithm and to observe the effects of the different algorithms. This simulation tool gives an opportunity to research about protocols.

5. VERIFICATION OF SIMULATION MODEL

There are many types of verification methods. Some of them are explained in Chapter 4. The most practical and reliable method for verification is analytical method. It produces cross checking. If the analytical results match with simulation results, simulation model is assumed to be verified successfully.

This simulator can simulate most of the bus based shared memory multiprocessor systems. In contrast, an analytical method can predict only certain architecture with strict assumptions. It is hard to find an analytical method to predict the performance of all architectures. For this reason, verification is tested on some specific architectures.

There are some assumptions on these architectures. All analytical results are based on these assumptions. These are as follows:

- A processor executes a sequence of accesses to its private memory or to cache and an access to a shared memory.
- Each processor can execute only one task. (i.e. no multitasking)
- All sub-task completion times are exponentially distributed.
- All memory modules have single-port. They can serve only one task at a time.

All the systems are analysed on different system load parameters. System load, which was explained in Section 4.6.1.1, is the ratio of average completion time of CPU sub-tasks over average completion time of MEM sub-tasks.

The performance index is another problem. In this example, average percentage of active processors, *the processor power*, is used as a performance evaluation index; because, many other indices may be evaluated from processor power.

There are three main architectures that are considered for verification. The first one is single bus multiprocessor with external common memory. The second one is more detailed. It is a multiple bus multiprocessor with multiple common memory. The last one is single bus multiprocessor with distributed common memory.

5.1. Single Bus Multiprocessor with External Common Memory

This may be the simplest form of the multiprocessor systems. There is only one global bus. This bus connects the processors to common memory. An example of such a system is shown in Figure 5.1.

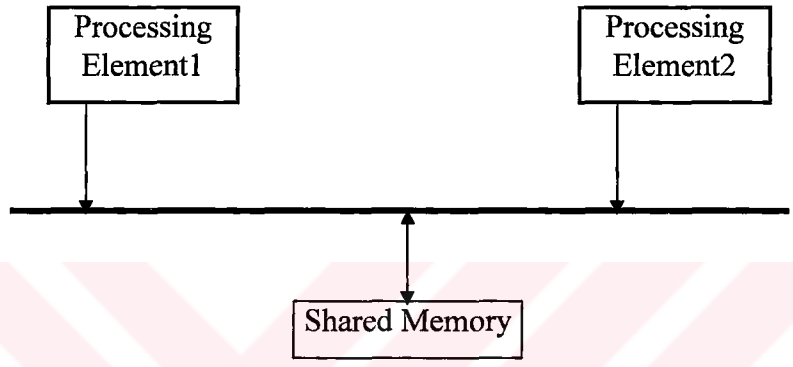
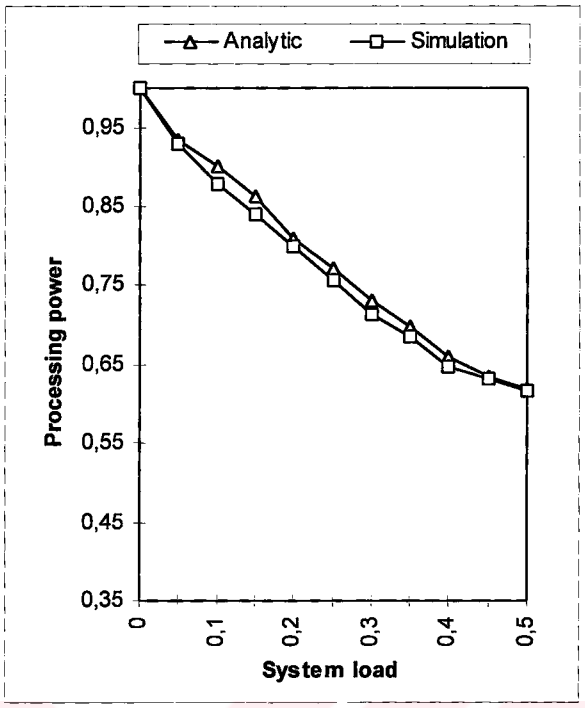


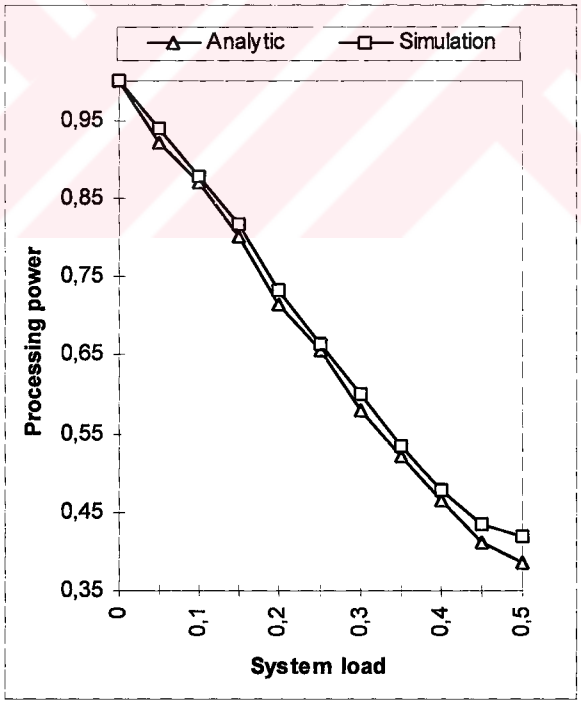
Figure 5.1. An example of single bus multiprocessor with external common memory.

The only bottleneck of the system is bus and memory. When more than one request for shared memory from processors only one of them is served. The others must wait in queuing state.

System with two processors and system with two processors are analysed. In Figure 5.2.(a), the comparison between analytic and simulation results in a system with two processors are shown. The results of system with five processors are in Figure 5.2(b).



(a)



(b)

Figure 5.2. Processor utilization under different system loads.(a) System with two processors, (b) System with five processors.

It is clear that there is no important difference between results of the models. The results of analytical model are highly correlated with simulation results.

5.2. Multiple Bus Multiprocessor with External Common Memory

Single bus multiprocessor with common memory system is a base system. It can be extended by some additional features. One of them is bus structure. There was only one bus in the first system. If system has more than one global bus, contention through the buses may be decreased. Figure 5.3. shows an example of such a system with three processors, two common memories and two global buses.

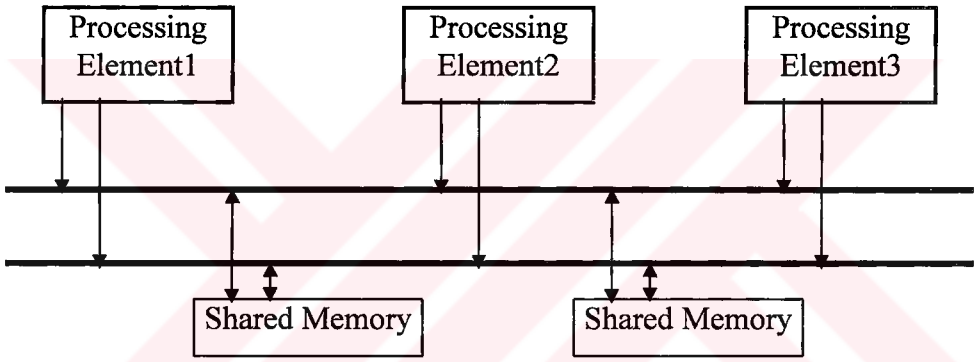


Figure 5.3. An example of multiple bus multiprocessor with external common memories and multiple buses.

While a processor accesses to a common memory, another processor can also access to another common memory. This opportunity is limited with the number of global buses and number of common memories.

System with 12 processors and two global buses is used to verify the simulator. Results are presented in Figure 5.4.

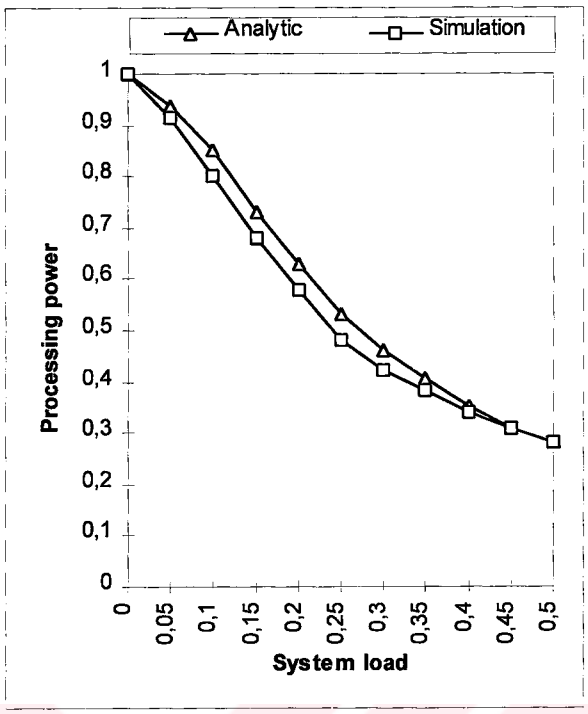


Figure 5.4. Simulation and analytical results.

Simulator finds lower processing power, while system load at 0.2-0.3. However, the main structure of the results is similar and close to each other.

5.3. Single Bus Multiprocessor with Distributed Memory

In multiprocessor systems that are analyzed, all common memories are at the same hierarchy with respect to processors. If common memories are split into separate modules that can be accessed with different rights, the distribution of memories provides less contention rate. In this way, utilization of the whole system may be increased.

There are many types of multiprocessor with distributed memory. One of them is shown in Figure 5.5.

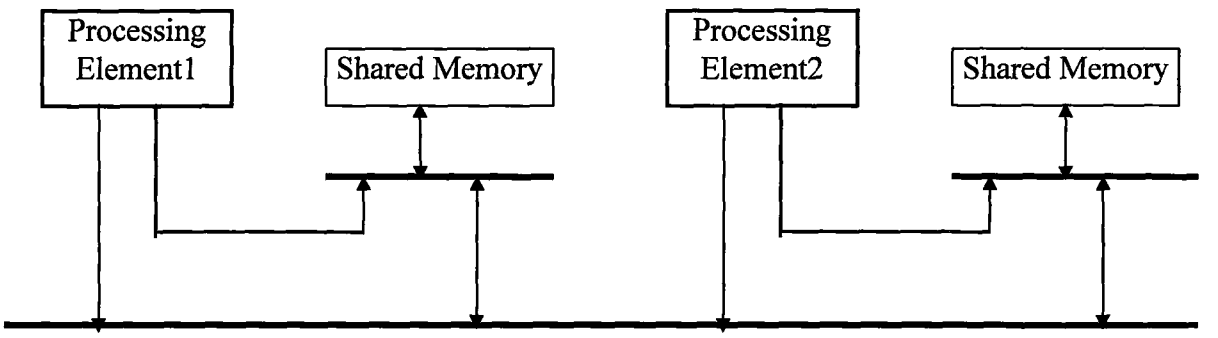


Figure 5.5. An example of single bus multiprocessor with distributed memory

Each processor accesses closer to common memory more frequently. For example, in Figure 5.5, P1 accesses to CM1 and P2 accesses to CM2 frequently.

In order to verify the simulator, multiprocessor system with three processors is used. The same architecture will be used for validation in the next chapter. The results are in Figure 5.6.

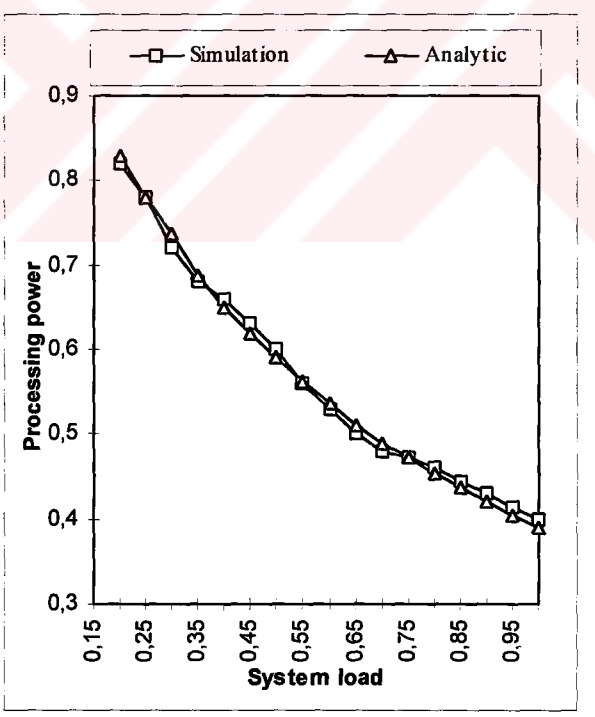


Figure 5.6. Simulation and analytical results of single bus multiprocessor system with distributed memory.

The results are very close and they provide the verification of simulator. The main reason for differences may be run length and replication number.

In this chapter, the verification of simulation model is provided. All results from analytical models and simulation model are very close to each other. It can be said that this simulator can simulate the desired models.

6. VALIDATION OF SIMULATION MODEL

Validation is the comparison of the model with the real world. If a simulator is valid, it is certain that it can produce reasonable results with respect to real world. In order to test the validation of a simulation model, its results must be compared with an actual system.

Unfortunately, validation is not a hot topic in multiprocessor system simulation. There are only a few studies about validation. In many times, when a new simulation model is developed, the real data may not be available, especially either if the simulated system is a new system, or if it is not implemented.

In this study, a general bus based multiprocessor system is simulated, and there are a number of bus based systems realised. One of them is TOMP prototype. TOMP is an implementation of multiprocessor system with distributed memory that is used for verification study [8].

All assumptions in Chapter 5 are also used for validation. There is an important additional assumption in this system. The probabilities of accessing to either local or external common memory modules are:

P(accessing local common memory module) = 0.5

and

P(accessing each external common memory) = 0.25.

Comparison between real data and simulation results are in Figure 6.1. This comparison shows that although simulation generally overestimates the real world, the correlation between them is sufficient. The differences between the data may be generated by the differences between the model and actual prototype. Another reason may be the differences of distribution between actual system and simulation model. However, it is clear that the predictions of simulator are very accurate.

There is a problem with the results of simulation. The results of simulation do not fit in a curve. In other word, it is fluctuating. The main reasons are the simulation run length and number of replication. If it runs longer, it is expected to fit in a curve as in TOMP's data.

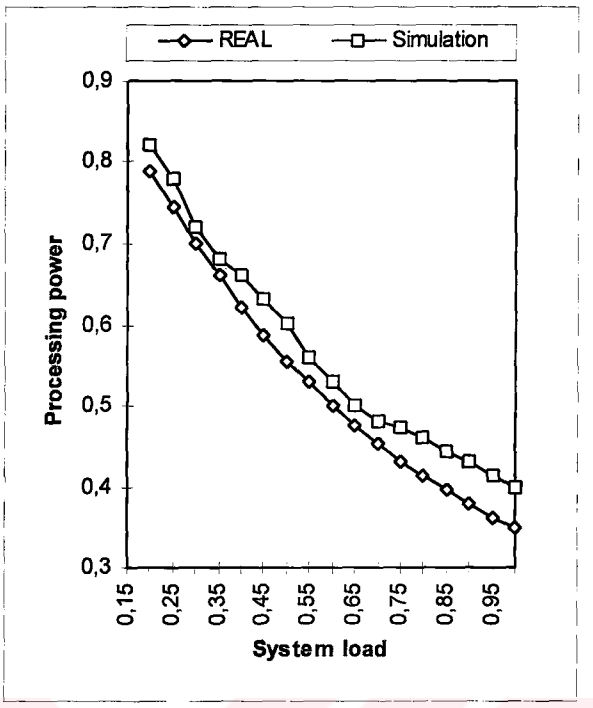
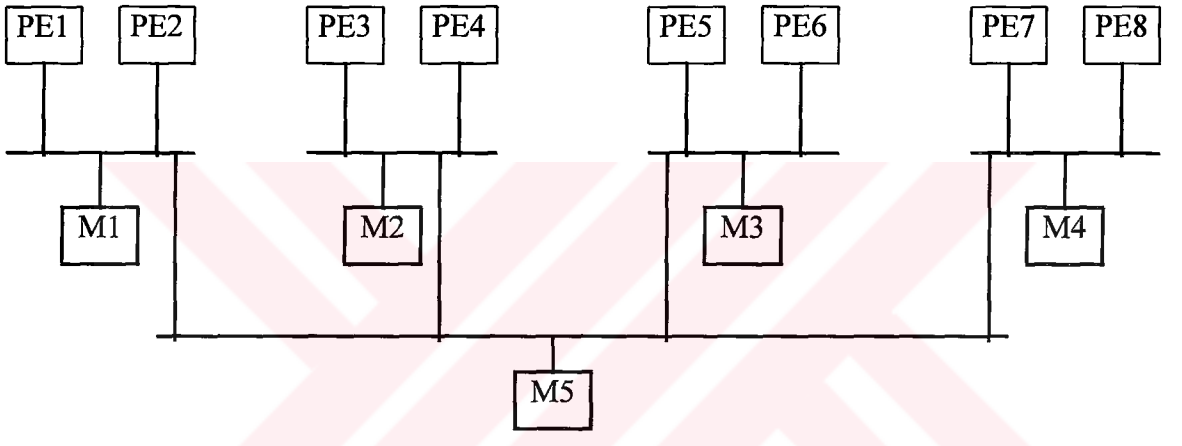


Figure 6.1. Comparison of TOMP architecture and simulation model results.

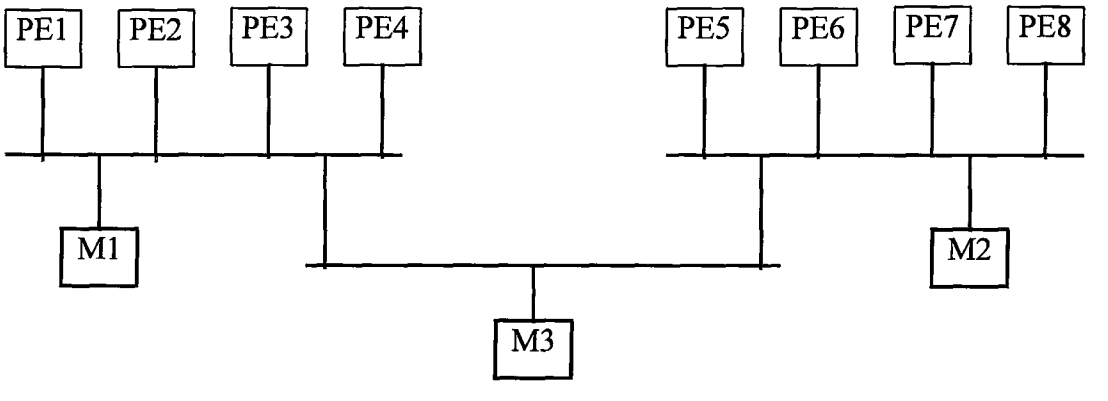
7. SIMULATION EXAMPLES

A number of simulations are conducted using the developed simulator to show how selected task graphs perform on various architectures. Furthermore the effect of changing parameters like probability distribution on shared memory access and use of alternative protocols developed in the thesis is analysed via multiple simulations.

Two hypothetical architectures and task graphs are selected for performance analysis. These are shown in Figure 7.1. and Figure 7.2., respectively.

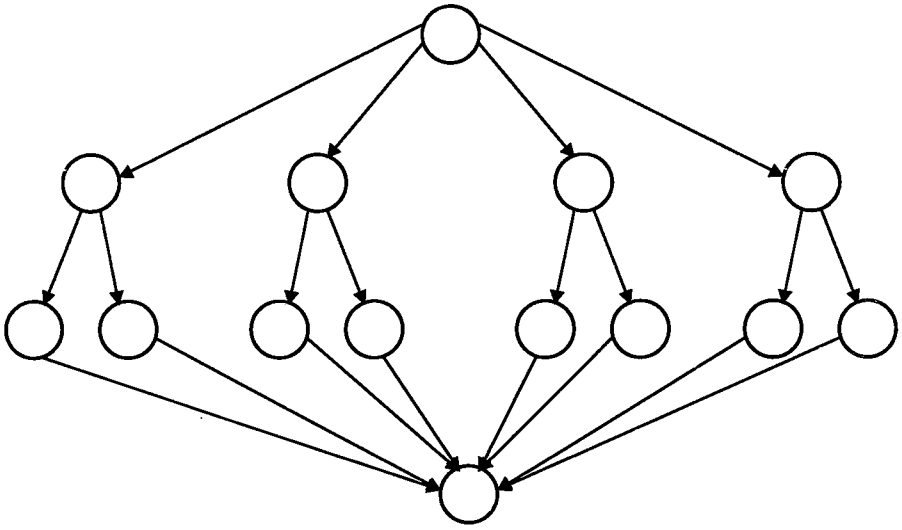


(a) Architecture1

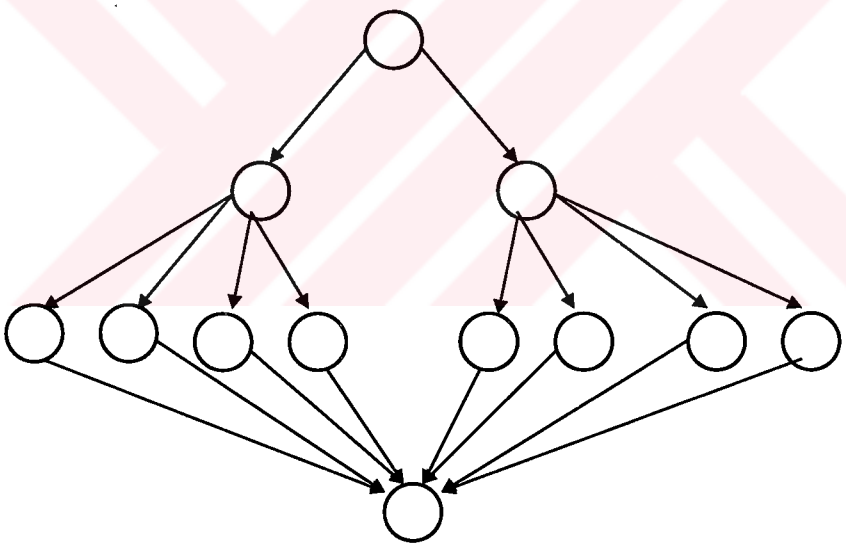


(b) Architecture 2.

Figure 7.1. Architectures used in the example analysis.



(a) Task graph 1 (TG1).



(b) Task Graph 2 (TG2).

Figure 7.2. Task graphs used in the example work.

The task graphs TG1 and TG2 are simulated on each of the architectures under various system loads. As the first approach, the default values of the simulator are used as the parameters. In other words, bus protocol and task selection protocol are FIFO. MP subtasks are at the beginning of the tasks and RM subtasks are at the end of the tasks. The conditional probability of memory usage, which is explained in Section 4.9, are assumed to

be 0.9. The probability used to determine the amount of work in each subtask is exponentially distributed.

7.1. The Effect of Architecture and Task Graph

Figure 7.3. shows the performance results of Architecture 1 under the workload of TG1 and TG2.

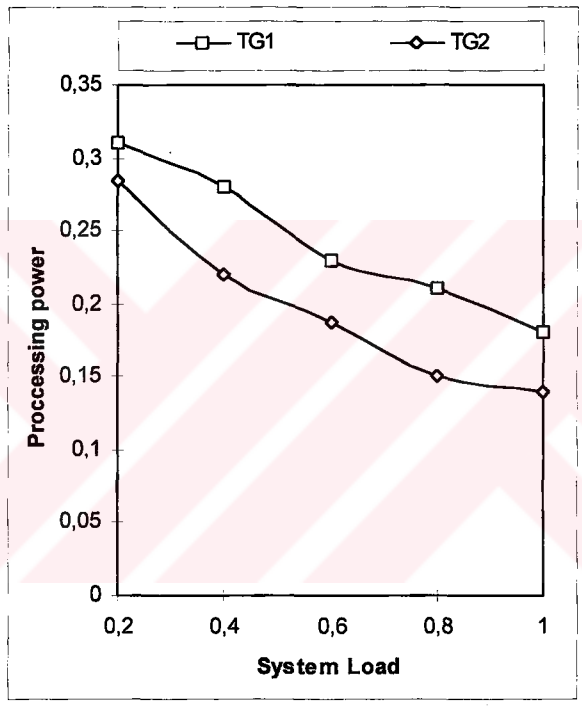


Figure 7.3. The performance data for Architecture 1 under TG1 and TG2.

As it can be seen from Figure 7.3. the task graph TG1 runs on the Architecture 1 with a better performance. On the contrary, Figure 7.4 shows that Architecture 2 fits to TG2 better.

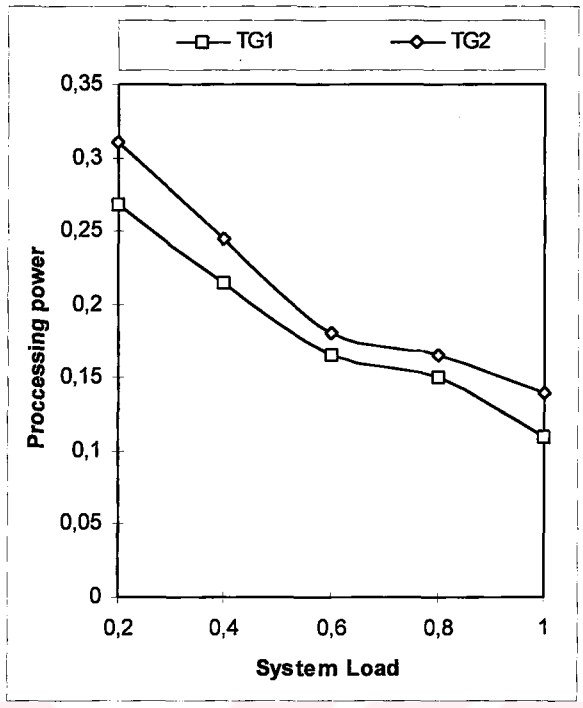


Figure 7.4. The performance data for Architecture 2 under TG1 and TG2.

7.2. Protocols

Up to now, the analysis is about the different cases, in architectures and task graphs. No default parameter value is changed. The effects of these parameters will be introduced in the further analysis. In the following, the architecture and task graph are fixed as Architecture 1 and TG1 and performance values are varied in conducting the simulation.

Figure 7.5. shows the performance, when the protocols are FIFO and random. The performance differences are considerable when the system load is larger. Because when the shared memory operation times are longer, the protocols will be more important and it will effect the performance of the system.

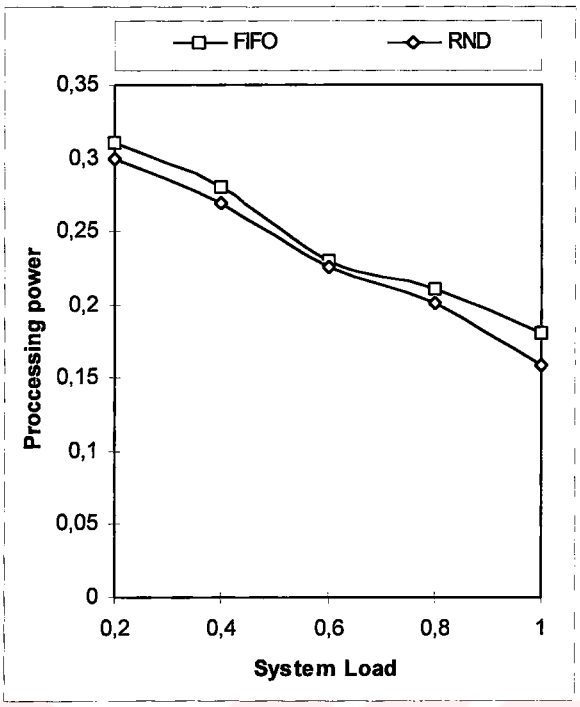


Figure 7.5.The performance data under the protocols of FIFO and Random.

7.3. The Place of Task Synchronization Mechanism

As it was explained in Section 4.6.1., there are two option for the place of the task synchronization subtasks. In the first option, tasks receive messages at the beginning and send messages at the end. In the second option, message passing may be in anywhere. The simulator chooses the first option as default. The performance may be different, when the second option is selected. The performance data varying the place of the task synchronization subtasks are given in Figure 7.6.

7.4. Data Distribution and Probability Distribution of the Memory Usage

The frequency of accessing a shared memory unit is determined by the help of a probability distribution, which is explained in Section 4.9. All conditional probabilities of choosing a cluster is 0.9 as default. It represents a good data distribution scheme. When data distribution is not so good, this probability value must be smaller. In this section, different probability values will be tested. The system performance is found when it is 0.7

and 0.5. The results are in Figure 7.7. As it can be seen, the data distribution effects the performance dramatically.

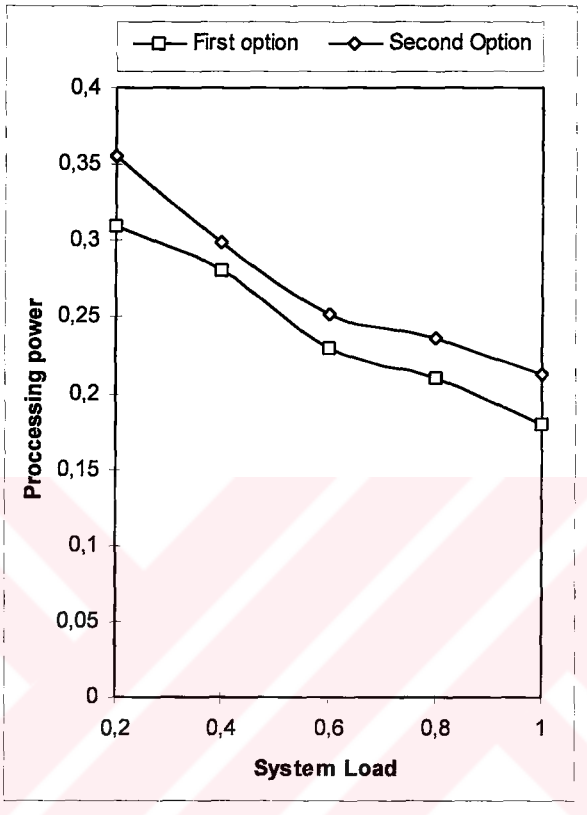


Figure 7.6.The performance data under different places of task synchronisation mechanism.

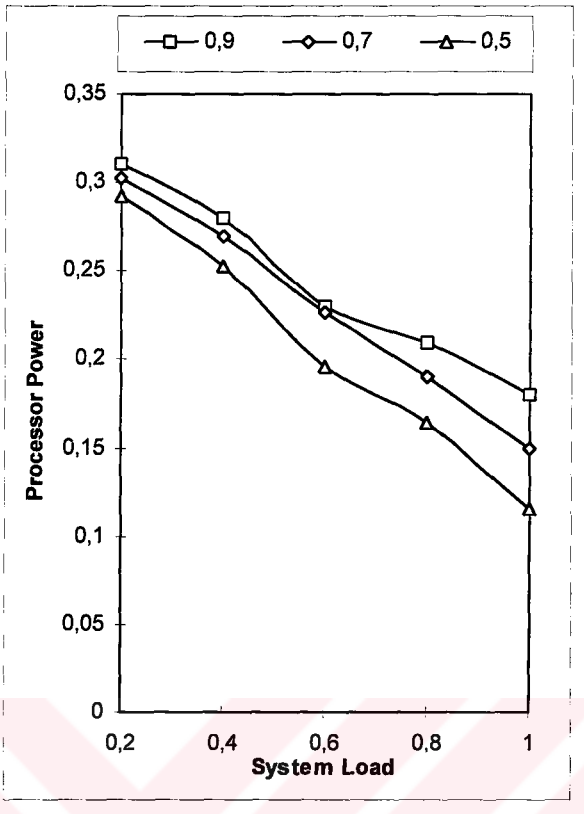


Figure 7.7.The performance data under different probability distributions.

8. CONCLUSION

Performance evaluation is one of the most important topics in computer world and simulation modeling is a powerful method to predict the performance of multiprocessing computer systems. In this study, a new simulation model is developed to predict the performance of bus based shared memory multiprocessing systems.

There are two main inputs of the simulator. One of them is the representation of the architecture that consists of processing elements, shared memory units and interconnection network. The second input is the workload, which is the representation of the work performed on the system. The workload is a task graph, so that each task consists of consecutive structure of some primitive routines. These routines represent the internal work in CPU, an access to a shared memory, or synchronisation of the tasks.

There are many algorithms to simulate the real multiprocessor systems in the simulator. Path finding, mailbox location, memory clustering are the most important algorithms. The details of these algorithms are explained in Chapter 4.

The flexibility is the most important characteristic of this simulator. It supports the elements with different speeds. It also runs the simulation under different probability distributions. There are many different protocols which user can choose. Moreover, many workload parameters can be set by the user.

When a simulator is developed, verification and validation are very important steps. The verification of the simulator is done by the help of analytic models in the literature. Validation is realized by the performance results of TOMP prototype.

Finally, the sample runs are given in Chapter 7. It includes the performance results of different architectures with different parameters used in the simulator.

This tool may be a base model for new researchers. It is easy to develop alternative algorithms or to add new protocols. Many new features can be inserted in this simulator.

REFERENCES

1. Duncan, R., "A Survey of Parallel Computer Architectures", *Computer*, Vol 23, pp.5-16, February 1990
2. DeCegama, A. L., *The Technology of Parallel Processing*, Prentice Hall, 1989
3. Lewis, T. G., and El,Rewini, H., *Introduction to Parallel Computing*, Prentice Hall, 1989.
4. Flynn, M. J., "Very High Speed Computing Systems", *Proc. IEEE*, Vol 54, pp.1901-1909, 1966
5. Gottlieb, A., "An Overview of the NYU Ultracomputer Project", *Ultracomputer Note*, Vol. 21, pp.18-33, July 1986.
6. Annavatone, M., "Applications and Algorithm Partitioning", *IEEE Conference Proceedings on Computer Architecture*, May 91, pp. 272-275.
7. Gaudiot, J.L., "Structure Handling in Data Flow Systems", *IEEE Transactions on Computers*, Vol 33, pp. 489-501, June 1986.
8. Marsan, M. A., and Balbo, G., *Performance Models of Multiprocessor Systems*, The MIT Press, 1986.
9. Dagum, L., and Simon, H. D., "NAS Benchmark Results", *IEEE Parallel and Distributed Technology*, Vol 6, pp. 40-55, February 1993.
10. Conte, G., DelCorso, D., and Gregorotti, F., "TOMP80-A Multiprocessor Prototype", *Proc. EUROMICRO 81*, September 1981.
11. MacDougall, M. H., *Simulating Computer Systems*, The MIT Press, 1987.
12. Ferrari, D., *Computer Systems Performance Evaluation*, Prentice Hall, 1978.
13. Serazzi, G., "Workload Modelling Techniques", *Proc. Modelling Techniques and Tools for Performance Analysis 85*, June 1985.
14. Parzen, E., *Stochastic Processes*, Holden Day, 1962.
15. Takacs, L., *Introduction To the Theory of Queues*, Oxford University Press, 1962.
16. Schwetman, H. D., "Hybrid Simulation Model of Computer Systems", *Communications of the ACM*, Vol21, pp. 718-723, September 1978.
17. Law, A. M., and Kelton, W. D., *Simulation Modelling and Analysis*, McGraw-Hill, 1982.

18. Butler, J. M., and Oruc, A. Y., "A Facility for Simulating Mulatiprocessors", *IEEE Micro*, pp. 32-44, October 1986.

19. Jonkers, H., and Reijns, G. L., "Predicting the performance of general task graphs with Underlying queuing model", *Proc. 1st Annual Conf. of the Advanced School for Computing and Imaging*, pp. 293-302, May 1995.

20. Holliday, M., and Stumm, M., "Performance Evaluation of Hierarchical Ring-Based Shared Memory Multiprocessors", *IEEE Transactions on Computers*, Vol.43, pp.52-67, January 1994.